

THE C LANGUAGE REFERENCE AND RUNTIME MANUAL

(AOS/VS)

093-000264-01

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085 series) supplied with the software.

Ordering No. 093-000264

©Data General Corporation, 1982, 1983

All Rights Reserved

Printed in the United States of America

Revision 01, April 1983

Licensed Material - Property of Data General Corporation

NOTICE

DATA GENERAL CORPORATION (DGC) HAS PREPARED THIS DOCUMENT FOR USE BY DGC PERSONNEL, LICENSEES, AND CUSTOMERS. THE INFORMATION CONTAINED HEREIN IS THE PROPERTY OF DGC AND SHALL NOT BE REPRODUCED IN WHOLE OR IN PART WITHOUT DGC PRIOR WRITTEN APPROVAL.

DGC reserves the right to make changes in the specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT, INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN, SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

CEO, DASHER, DATAPREP, ECLIPSE, ECLIPSE MV/4000, ECLIPSE MV/6000, ECLIPSE MV/8000, ENTERPRISE, INFOS, MANAP, microNOVA, NOVA, PRESENT, PROXI, SUPERNOVA, and TRENDVIEW are U.S. registered trademarks of Data General Corporation, and AZ-TEXT, DG/L, ECLIPSE MV/10000, GENAP, REV-UP, SWAT, XODIAC, DEFINE, SLATE, microECLIPSE, GW/4000, GDC/1000, BusipEN, BusiGEN, and BusiTEXT are U.S. trademarks of Data General Corporation.

THE C LANGUAGE REFERENCE AND RUNTIME MANUAL (AOS/VS)

093-000264

Revision History:

Original Release - November 1982
First Revision - April 1983

Effective with:

C Rev. 1.10

A vertical bar in the margin of a page indicates a substantive change from the previous version.

PREFACE

The C language is a concise, structured programming language. It is sufficiently low-level that it deals with objects of the sort that most computers do: characters, numbers, addresses. But it is sufficiently general that it can serve a wide variety of applications. Its nonsequential control flow, strongly-typed data structures, rich set of operators, and relatively few restrictions allow the convenient, effective performance of complex tasks. A C program consists essentially of a set of functions defined on specified data types and returning results of those types. C compilers are available for a variety of machines. With some care, you can transport programs to and from Data General's C language implementation.

The Reader of This Manual

=====

This manual is written for people who have some experience programming computers, either in assembly language or a high-level language, and who may already be familiar with C. Since this manual is primarily a reference book, not a tutorial, we suggest that people with no experience in C consult the primary reference for C, "The C Programming Language," by Kernighan and Ritchie, Prentice Hall, New York, 1978.

Organization of this Manual

=====

Chapter 1, "An Overview of the C Language," introduces the general properties of the C language, and gives simple procedures for compiling, linking, and executing C programs.

Chapter 2, "Tokens," describes the symbols permitted in C and explains C's tokens: identifiers, keywords, constants, variables, operators, and token separators.

Chapter 3, "Declarations, Data Types, Data Type Conversions, and Data Initializations," describes C's data types and storage class types and explains data declarations, initializations, and conversions.

Chapter 4, "Expressions," describes how expressions in C are built up from identifiers and operators, and it gives the precedence of the C operators.

Chapter 5, "Statements," gives the form and use of C statements.

Chapter 6, "Block Structure and Functions," explains how to create blocks of C code, how to create and use functions in C, and how to call functions in a C program that are written in other languages.

Chapter 7, "Dictionary of C Keywords," describes the C keywords; they are listed in alphabetical order.

Chapter 8, "Dictionary of C Operators," details the C operators.

Chapter 9, "Preprocessor Statements," outlines compilation-time options for file inclusion, macro definition, conditional compilation, and listing control.

Chapter 10, "The Runtime Environment," describes the use of certain C functions. It serves as a library of standard functions and macros.

Chapter 11, "Operating Procedures," shows how to install the C language in the AOS/VS environment, and how to compile, link, and execute C programs. It also describes error handling.

Appendix A, "Error Codes," lists compiler error messages.

Appendix B, "A Graph of C Syntax," summarizes the structure of the C language.

Appendix C, "Implementation Characteristics," outlines hardware representations, differences from other implementations, and added features of Data General's C language implementation.

Appendix D lists the ASCII symbol set.

Appendix E gives conversions to and from decimal, octal, and hexadecimal numbers.

Additional References

=====

This manual assumes as a de facto standard the C language described in

Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language (Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978). Copyright (c) 1978 by Bell Laboratories, Incorporated.

The user of this manual may find these related manuals helpful:

"Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)" (093-000122)

"AOS/VS Programmer's Manual" (093-000241)

"AOS/VS Link and Library File Editor User's Manual" (093-000245)

"AOS/VS Macroassembler Reference Manual" (093-000242)

"Advanced Operating System/Virtual Storage (AOS/VS) Debugger
and File Editor User's Manual" (093-000246)

"SWAT Debugger User's Manual" (093-000258)

"INFOS II System User's Manual (AOS/VS)" (093-299000)

"AOS/RT32 Command Line Interpreter" (069-400028)

"AOS/RT32 System Programmer's Reference" (093-400016)

"AOS/RT32 Debugger" (069-400029)

"The ECLIPSE MV/8000 Principles of Operation" (014-000648)

Reader, Please Note:

=====

We use these conventions in talking about the syntax of the C language:

Convention	Means
-----	-----

keyword	You must enter the keyword as shown.

<required>	You must specify an identifier.

{opt1 opt2}	You must enter one of the members of the list. Don't enter the braces; they only set off the choice.

[optional]	You have the option of entering this argument. Don't enter the brackets; they only set off what's optional.

...	You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

In a few examples, we use

) to show the CLI prompt.

Contacting Data General

=====

- * If you have comments on this manual, please use the prepaid Remarks Form that appears after the Index. We want to know what you like and dislike about this manual.
- * If you need additional manuals, please use the enclosed TIPS order form (USA only) or contact your Data General sales representative.
- * If you experience software problems, please notify Data General Systems Engineering.

End of Preface

CONTENTS

Chapter 1 - An Overview of the C Language

Data General C	1-1
Operating System	1-2
Using the C Language	1-2
Sample C Program	1-7
Programming Style in C	1-7
The Rest of This Manual	1-8

Chapter 2 - Tokens

The C Character Set	2-1
Upper- and Lowercase	2-2
C Tokens	2-3
C Identifiers (Names)	2-4
Rules for All C Identifiers	2-4
Reserved Words (Keywords)	2-4
C Runtime Library Routines: Are They Reserved?	2-5
Constants	2-6
How to Specify the Radix of a Constant	2-6
Character Constants	2-7
String Constants	2-8
Exponents and Scientific Notation	2-8
Long Integer Constants	2-8
Symbolic Constants	2-9
Operators	2-10
Token Separators	2-11
Preprocessor Delimiter	2-11
Statement Delimiters	2-12
Comments	2-12

Chapter 3 - Declarations, Data Types, Data Type Conversions, and Initializations

Declarations	3-1
Syntax of Declarations	3-2
Where Can Declarations Occur?	3-2
Data Types	3-2
Fundamental Data Types	3-4
The Adjectives short, long, and unsigned	3-4
int	3-5
char	3-6
float	3-7
double	3-7

Derived Data Types	3-8
Arrays	3-8
Pointers	3-10
structs	3-11
unions	3-13
enums	3-13
Scope of Variables	3-14
The typedef Keyword	3-15
Data Type Conversions	3-16
The Usual Arithmetic Conversions	3-16
Conversion by Assignment	3-17
Passing Arguments to Functions	3-18
Casts	3-18
Sign Extension	3-18
Storage Classes	3-19
auto	3-19
static	3-20
register	3-20
extern	3-21
typedef	3-21
\$builtin	3-22
Some Tips on Storage Classes	3-22
Initializations and Defaults	3-22
Default Data Types and Storage Classes	3-23

Chapter 4 - Expressions

C Operators	4-1
Precedence	4-1
Grouping	4-2
C Expressions	4-3
Lvalues	4-3
Primary Expressions	4-4
Kinds of Expressions	4-4
Brackets, Parentheses, Structure Pointers, and Structure-Member Operators	4-4
Increment and Decrement Expressions	4-5
Arithmetic Expressions	4-6
A Note on Side Effects in C	4-7
Relational Expressions, Equality Expressions, and Some Logical Expressions	4-7
Bitwise Expressions	4-8
Assignment Expressions	4-9
Compound Assignment Operators	4-10
The Conditional Operator	4-10
Punctuation for Expressions: the Comma	4-11

Chapter 5 - Statements

Simple Statements	5-1
Expression Statements	5-2
Conditional Statements	5-2
if-else	5-3
switch and case	5-5
Loops	5-6
while	5-6
for	5-7
do	5-8
Getting Out of Loops	5-9
break	5-9
continue	5-9
goto and Labels	5-10
Flow of Control	5-10
Compound Statements	5-11
The Null Statement	5-11

Chapter 6 - Block Structure and Functions

Blocks of C Code	6-1
How to Create a Block of C Code	6-2
How to Create a Function	6-3
How to declare functions	6-4
Data Type	6-5
Storage Class	6-6
The main() Function	6-6
How to Call Functions	6-7
How to Pass Data To and From Functions	6-8
The return Statement	6-8
Recursive Calling	6-9
How to Call Functions Written in Other Languages	6-10
Calling Sequences	6-10
VC/ECS Calling Sequence	6-11
Making the Call	6-12
When the Call Returns	6-13
C Calling Sequence	6-14
Making the Call	6-14
When the Call Returns	6-15
Assembly Language Calling Sequence	6-16
Declarations for Other-Language Calling	6-16
Storage Class	6-17
Data Types	6-17

Chapter 7 - Dictionary of C Keywords

\$builtin	7-2
auto	7-3

break	7-4
case	7-5
char	7-7
continue	7-9
default	7-10
do	7-11
double	7-12
else	7-13
entry	7-15
enum	7-16
extern	7-18
float	7-20
for	7-22
goto	7-24
if	7-25
int	7-27
long	7-29
register	7-30
return	7-31
short	7-33
sizeof	7-34
static	7-35
struct	7-37
switch	7-39
typedef	7-41
union	7-43
unsigned	7-44
void	7-46
while	7-47

Chapter 8 - Dictionary of C Operators

+ Addition Operator	8-5
& Address Operator	8-6
= Assignment Operator	8-7
& Bitwise AND Operator	8-9
^ Bitwise Exclusive OR Operator	8-10
Bitwise Inclusive O Operator	8-11
?: Conditional Operator	8-12
-- Decrement Operator	8-13
/ Division Operator	8-14
== Equality Operator	8-15
> Greater-than Operator	8-16
>= or => Greater-than-or-equal-to Operator	8-17
++ Increment Operator	8-18
* Indirection Operator	8-19
!= Inequality Operator	8-21
< Less-than Operator	8-22
<= Less-than-or-equal-to Operator	8-23
&& Logical AND Operator	8-24
! Logical Negation Operator (Logical NOT)	8-25
Logical OR Operator	8-26

<% Modulus Operator	8-27
* Multiplication Operator	8-28
~ Ones Complement Operator	8-29
<< Shift Left Operator	8-30
>> Shift Right Operator	8-31
sizeof	8-32
. Structure Member Operator	8-33
-> Structure Pointer Operator	8-34
- Subtraction Operator	8-36
- Unary Minus Operator	8-37
, Comma Operator	8-38

Chapter 9 - Preprocessor Statements

Preprocessor Syntax	9-1
Macros	9-2
Simple Macros	9-2
Macros with Arguments	9-3
Invalidating Macros	9-4
Include Files	9-5
Conditional Compilation	9-5
Test an Expression	9-6
Test for a Definition	9-6
Specify an Alternative	9-6
End a Conditional Compilation	9-7
Listing Control	9-7
Line Control	9-7
Suspension of Listing	9-7
Cross Referencing	9-8
Testing Assertions	9-11

Chapter 10 - The Runtime Environment

Arithmetic Functions	10-1
Conversion Functions	10-2
File-Handling Functions	10-2
Input/Output Functions	10-2
Memory Management Functions	10-3
Miscellaneous Functions	10-3
Program Environment Functions	10-4
Queue-Handling Functions	10-5
Character-Handling Macros	10-5
String-Handling Functions	10-5
Standard \$builtin Functions	10-6
Standard Include Files	10-6
Library of Standard Functions and Macros	10-10

Chapter 11 - Operating Procedures

Compiling C Program Modules	11-1
AOS/VS Compilation Command Line	11-1
Conditional Compilation	11-2
Macros	11-3
Include Files	11-4
Compilation Switches	11-4
Global Switches	11-7
Local Switches	11-15
CC Macro	11-16
Linking C Programs	11-19
Terminology	11-19
Use of the Switches	11-20
.RESERVE	11-20
.STKSIZE	11-20
.HPSIZE	11-20
.STORAGE	11-21
Link Example	11-21
Combinable Options	11-21
<none>	11-21
.RESERVE	11-21
.STKSIZE	11-21
.STORAGE	11-22
.HPSIZE	11-22
.RESERVE, .STKSIZE	11-22
.RESERVE, .HPSIZE	11-22
.STKSIZE, .HPSIZE	11-22
CCL LINK Macro	11-23
Executing C Programs	11-24
File Redirection	11-25
Furnishing Arguments to Programs	11-26
Debugging Aids for C Programs	11-26
SWAT Native Language Debugger	11-27
AOS/VS Assembly Language Debugger	11-28
Error Handling for C Programs	11-28

Appendix A - Compilation Error Messages

Severity Levels	A-1
Compiler Error Messages	A-1
Command Line Errors	A-1
Preprocessor Errors	A-3
Token Errors	A-5
Code Generation Errors	A-6
Parser Errors	A-7
Initialization Errors	A-12
Conversion Errors	A-13
Global Errors	A-16

Appendix B - A Syntax Graph for the C Language

Statement Syntax	B-1
Expression Syntax	B-2
Declaration Syntax	B-4
Module Syntax	B-6
Preprocessor Syntax	B-6

Appendix C - Implementation Characteristics

Hardware Representations	C-1
Data Sizes and Machine Dependencies	C-1
Integer Types	C-1
Character Types	C-1
Real Types	C-2
Character Constants	C-2
Pointers	C-2
The struct Type	C-3
Implementation Differences	C-3
Line Length	C-3
Identifiers	C-3
Lengths	C-3
Number of Arguments	C-3
Legal Characters	C-4
Externals	C-4
Keywords	C-4
Additional Keywords	C-4
Reserved Keywords	C-4
String Constants	C-5
Initializations	C-5
unions	C-5
Storage Type auto structs and unions	C-5
Expressions	C-5
Operators	C-6
Labels	C-6
Additional Features	C-6
Identifiers	C-6
structs	C-6
Moving structs	C-6
Reusing the Tag Field	C-7
Forward Referencing	C-7
Conversions	C-7
Additional Data Types	C-8
The unsigned Types	C-8
Enumerations	C-8
The void Type	C-9
Initializations	C-9
Functions with extern Variables	C-9
Constants in auto Arrays and Structures	C-9
Characters	C-10
Vertical Tab	C-10

Uppercase Escapes	C-10
Escape in Arguments	C-10
Format Padding	C-10
Constant Folding	C-11
Calling Procedures in Other Languages From C	C-11
Preprocessor Statements	C-11
Shared Page I/O	C-12
AOS/RT32	C-12

Appendix D - ASCII Character Set

Appendix E - Octal and Hexadecimal Conversion

ILLUSTRATIONS

Figure Caption

11-1 Memory Configurations 11-23

TABLES

Table Caption

2-1 C Tokens 2-3

6-1 Comparison of Data Types in C, FORTRAN 77,
 Pascal, and PL/I 6-18

8-1 Precedence of Operators in C 8-3

10-1 Standard Functions and Macros 10-11

10-2 System Calls from C 10-307

11-1 Global Compilation Switches 11-5

11-2 Local Compilation Switches 11-7

11-3 Switches for Program Execution 11-25

✓

✓

✓

CHAPTER 1

AN OVERVIEW OF C

C is a versatile general-purpose computer programming language. It is designed for both systems-level programming and applications programming.

The C language was originally created as part of the UNIXTM operating system, which was developed at the Bell Laboratories in the 1970s. The C language is small compared to other languages; in particular, I/O statements are not part of the C language proper. Instead, C relies on a rich runtime library, which traditionally comes with every C compiler. The Data General C runtime library contains all the routines your programs need to perform I/O, as well as many other useful and traditional C-related functions.

In addition to C's brevity and its interface with the UNIX operating system, C includes the usual features one associates with a high-level programming language: flexible and easily-convertible data types, including user-defined arrays and structures; conditional branching; loops; user-defined functions; and the ability to interface smoothly with other languages.

The standard reference for the C language, which contains the specification of C currently in use in the C user community, is "The C Programming Language", by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1978. We urge you to read this book for a thorough understanding of the C language.

Data General C

=====

Data General's C compiler supports all the functionality described in the specification of the C language in Appendix A of Kernighan and Ritchie.

The runtime routines that come with the Data General implementation of C are described in detail in Chapter 10.

The Data General implementation of C also offers some extensions to the language, which are summarized in Appendix C of this book.

To provide the UNIX-related functionality that C programs are accustomed to, Data General C comes with a library of UNIX-emulating system calls and routines. These emulation routines are described in the C Release Notice.

UNIX is a trademark of Bell Laboratories.

Operating System

=====

The Data General C compiler, the Data General C runtime routines, and the Data General UNIX emulation routines all run under the Advanced Operating System/Virtual Storage (AOS/VS).

A subset of the Data General C runtime routines will let your C programs run under the Advanced Operating System/Real Time 32-Bit (AOS/RT32). For more information on running C programs under AOS/RT32, see Chapter 10.

Using the C Language

=====

This section contains questions and answers about the fundamentals of programming in C.

- * What does the minimum C program look like? This very short C program fetches a character from the input device and sends it to the output device:

```
main()
int c;
{
    while((c=getchar()) != EOF) /* get char from input */
        putchar(c);           /* send char to output */
}
```

There's a larger sample C program below.

- * What does the minimum command line to run the C compiler look like?

) CC program-name

 CC.CLI is actually the name of a CLI macro that runs the C compiler;
 CC.CLI comes with Data General C.
- * What does the minimum command line to link a C program look like?

) CCL program-name

 CCL.CLI is actually the name of a CLI macro that runs the link
 utility program; CCL.CLI comes with Data General C.
- * What is the minimum command line to compile, link and execute
 a C program?

) CC/LINK program-name; XEQ program-name

- * How do I define a block of code in C? With curly brackets; they work the same way as begin and end in Pascal.

```

{
    statements
}

```

- * How do I make a statement in C? You build statements from expressions; you build expressions from identifiers and operators. Simple C statements end with a semicolon.

```

i++;                /* increment i by 1.                */

sign = s[i++];      /* set sign equal to the ith      */
                   /* element of s, then increment i */

sign = (s[i++] == 'thing'); /* set sign equal to the truth   */
                           /* value of the expression; i.e., */
                           /* sign will be either 1 (true)   */
                           /* or 0 (false)                  */

```

You can also make statements in C from the various looping, flow control, and compound constructs.

- * How do I create an object? How do I name an object? You can name pieces of data and pieces of code; all names in C have 32 significant characters; the total length is switch-selectable, down to 8 characters, total. Names must begin with A through Z, a through z, underscore, dollar sign, or commercial at sign; names can include these characters and numbers. Sample names:

sign, _sign, \$sign, @sign	names of identifiers
sign()	a function named sign.
*sign	a pointer, possibly to the beginning of an array
sign[]	an array named sign.

You can declare and initialize things in the same statement:

```
char chiffre = 'z';    /* declare chiffre, init. it to "z". */
```

- * How do I create an array in C? You must declare the array in terms of what it contains. Does it contain integers? Then it is an array of integers. For an array of characters called command, use this declaration:

```
char command[];        /* declare char. array named command */
```

Inside the brackets, you can put an expression to show how large the array is. And as with other variables, you can initialize the array in the same statement you declare it:

```
char command[] = "F/S/AS";
```

- * How do I create a record, like those in Pascal? In C, records are called structures. To declare such a structure involves two steps. First, you create the data type:

```
struct music {                /* data type named music. */
    char datastring[SINGER];  /* members of music: */
    int datacount;            /* datastring, datacount */
}
```

This defines a structure type named music, which has two parts: a character array and an integer. When you mention music after this, C requires that you use the keyword struct with it.

Second, you declare variables to be of this type, music:

```
struct music show[3]; /* create an array, called show, */
                      /* of music structures. */
```

Finally, you can initialize the members of the structure. In this example, we set the entire array, show:

```
show =                      /* init. the 3 members of show. */
{ "Sinatra", 3,
  "Ellington", 1,
  "Basie", 2,
};
```

- * How do I create a pointer in C? Use the & operator to determine an address, and the * operator to declare a pointer. Some examples:

```
char lady; /* declare a character variable named lady. */

char *p; /* declare a pointer, named p, to a charac- */
          /* ter. The character is not specified yet.*/

p = &lady; /* set p equal to the address of lady; */
          /* p now points to the char. variable, lady.*/
```

- * How do I make a comment in C?

```
/* like this */
```


- * What operators does C support? In addition to the familiar arithmetic, logical, relational, bitwise, and assignment operators, C has some special operators, including:

++	increment
--	decrement
?:	conditional
->	structure pointer
.	structure member
~	one's complement
sizeof	size, in bytes, of operand

The C operators are described in detail in Chapter 8.

- * What are the data types in C? There are really only two data types in C: integers and real numbers. In fact, there are four data type specifiers in C:

char	specifies a character, which is a subset of integer
int	specifies an integer
float	specifies a floating-point (real) number
double	specifies a double-precision floating-point number

The data types take adjectives, which are:

short	16 bits
long	64 bits
unsigned	allocate the sign bit to the mantissa

You can also declare storage classes, which govern the permanence and scope of variables:

auto	variable goes away when the function goes away
static	variable stays as is even when the function is not active
register	implemented as auto in Data General C
extern	variable is declared in another module

All the data type and storage class keywords can be combined according to common-sense rules.

- * How do I use a string in C? First, put the string into an array of characters:

```
int i;
char string1[];          /* declare an array of characters */

i = 0;
while ((c=getchar()) != EOF ) /* use the getchar function */
    string[i++] = c;         /* to put each typed-in      */
                             /* char into the array, c.    */
```

You can also put the string into the array automatically, when you first declare the array:

```
char string[] = "Hello, string here.";
```

Then, sift through the array for what you want:

```
for (i=0; d=string[i]; i++)
    if d == 'sought'
        ...                /* you found it */
    else
        ...                /* keep looking */
```

- * What functions come with C? There are many. Some frequently-used functions are: putchar() and getchar() and printf() and scanf() which get and give characters and strings respectively on the standard input/output device (your terminal/keyboard).
- * How do I pass arguments to and from functions? C permits passing arguments by value only; passing pointers, however, is an alternative. The Data General implementation of C offers the added feature of allowing your program to pass a whole structure to a function. Functions are described in Chapter 6; structures are described in detail in Chapter 7.
- * How do I do input and output in C? C programs do I/O to and from a standard input and a standard output device. You can set these devices to be anything from a terminal to a disk to another program. Normally, they are set to your terminal and keyboard.

There are many I/O functions. There are also many formatting options in the functions, as with printf(), for example. The basic C I/O routine calls the putchar and getchar functions:

```
#include <stdio.h>                /* permits calls to getchar */
                                   /* and putchar          */
main() {
    int ch;
    while ((c=getchar() != EOF) /* get a char; end of file? */
        putchar();             /* no; put the char      */
    }
```

- * Does C let me create programs that exist in two or more modules? Does C allow include files? Does C have external variables? Can programs written in Data General C call programs written in other languages? Yes, Data General C provides all these features. The instructions for using the C compiler, in Chapter 11, explain how to combine several C modules into a single C program. Chapter 6 contains a section on interlanguage calling, between C and FORTRAN 77, C and PL/I, C and Pascal, and C and COBOL.

Sample C Program

```
=====

/* COUNTCHAR4.C  Counts characters and lines; also loops  */
/* and asks user: "Go again?"  6.15.82                      */

#define ESC '\033'

main ()
{
    int c, nc, nl;
    char wish;

    nc = 0;
    nl = 0;
    wish = 'y';
    printf ("COUNTCHAR4.  Type in some text.  Type ESC-NL to end.\n");

    while (wish == 'y')
    {
        while ((c = getchar()) != ESC)          /* read from keyboard  */
        {
            ++nc;
            if (c == '\n')
                ++nl;
        }
        printf ("Number of characters: ");      /* on ESC, give report  */
        printf ("%d\n", nc);
        nc = 0;
        printf ("Number of lines: ");
        printf ("%d\n", nl);
        nl = 0;
        printf ("Again? (y,n): \n");
        scanf ("%s", &wish);
    }                                           /* end of while (wish  */
    printf ("COUNTCHAR4 terminating.\n");
}
=====
```

Programming Style in C

=====

In the examples in this book, we sometimes use extra spaces and tabs to make the examples easier to read. We also include extra parentheses for clarity, and sometimes we use the braces ({ and }) like Pascal begin and end statements. Other C programmers may use these tokens differently.

We chose our style simply to make the examples easier on the eye and the mind; C is flexible enough to let us do so. It's true that there are some C conventions; we simply suggest you pick the ones you like and then stick to them.

The Rest of This Manual

The next four chapters, Chapters 2 through 6, give brief explanations of how to use the C features. Chapter 7 is a dictionary of the C keywords, and Chapter 8 is a dictionary of the C operators. Chapter 9 describes the statements for the C compiler's preprocessor. Chapter 10 lists all the runtime routines for Data General C. Chapter 11 explains how to operate the C compiler to compile your C programs and how to link your C programs.

End of Chapter

CHAPTER 2

TOKENS

Tokens are the simplest building blocks in a C program.

Speaking very generally, there is a hierarchy in C. The C character set makes tokens; the tokens combine into expressions; the expressions combine to form statements; and statements group together into blocks of code.

In this chapter we'll look at the C character set, and describe the C tokens and show how to make them.

The C Character Set

C uses most of the standard graphic characters. This section describes how C uses them.

When you make up identifiers in C, you can use the following characters:

- a through z
- A through Z
- 0 through 9
- underscore

The DG implementation of C permits the use of the following characters in identifiers:

- \$ dollar sign
- @ commercial at sign

CAUTION: You may want to avoid identifiers that begin with a dollar sign or underscore. Certain internal routines in the C library begin with these characters, and your program could call them unexpectedly. Also, Data General reserves the right to create new reserved words out of identifiers that begin with \$.

The C keywords, which are a subset of all the possible identifiers in C, use only the letters a through z.

The C operators use the following symbols:

+ & ^ | ? : - = / < > * ! . % ~

The C operators are described in more detail below; and they are described in complete detail in Chapter 8.

Other symbols that are necessary in C programs are:

()	parentheses	delimit an expression, delimit argument lists of functions, contain the control statements that for, while, if, use
[]	square brackets	show size of an array or an index into an array
{ }	braces	delimit a compound statement or a function
l L	letter L	specifies a constant as long
e E	letter e	shows the exponent part of a constant
,	comma	gives order of evaluation for expressions, separates function arguments
' '	single quotes	delimit a character constant
" "	double quotes	delimit a string constant

For other purposes, C uses the following characters and symbols:

\	backslash	begins an escape sequence
#	number sign	delimits a preprocessor statement

C does NOT use the following characters:

`	accent grave

Upper- and Lowercase

C is a lowercase language. The C compiler recognizes keywords only if they are in lowercase; thus, it does not recognize IF as the keyword if.

The /LOWERCASE switch on the C compiler will map all identifiers in a source file to lowercase, and in that case, C would recognize IF as the keyword if.

C Tokens

=====

For the purposes of this discussion, we'll divide the legal tokens in C into five classes:

- * identifiers, i.e., names of things, including functions and variables
- * C keywords (as we said, these are all also identifiers)
- * constants, including strings
- * operators
- * separators, including spaces, tabs, vertical tabs, NEW LINES, form feeds, and comments

The sections below describe each of these classes.

The only other items than can appear in your program file are preprocessor statements, which must begin with a number sign; for example,

```
#define EOF -1
```

For complete information on the C preprocessor, see Chapter 9.

Table 2-1. C Tokens

Token	You can use these characters	Useful For
identifiers	A through Z, a through z, 0 through 9, @, _, and \$	names of variables, functions, keywords, etc.
C keywords	must be one of the C keywords; see Chapter 7	basic C actions
constants	numbers, strings	global constants, strings, error messages
operators	must be a valid C operator; see Chapter 8	forming expressions
separators	space, tab, vertical tab, NEW LINE, form feed, comment	separating tokens

C Identifiers (Names)

=====

Identifiers are names, and as such they're useful for distinguishing pieces of data from each other and from sections of your program code.

C identifiers can name a variety of things:

- * If they name data, they're just called identifiers;
- * If they name code, they're called function calls;
- * If they name locations in the code, they're called labels; and
- * If they name locations in the data, they're called pointers.

The most common use for identifiers in C is probably to identify variables, like numbers, character arrays, and so on.

As with all other tokens in C, to make an identifier, you simply type it into your program code, following the rules for the particular kind of identifier you want. The next section describes the rules that apply to all C identifiers; for the rules specific to function calls, see Chapter 6; for the rules specific to labels, see Chapter 5; for the rules specific to pointers, see Chapter 3.

Rules for All C Identifiers

A C identifier is simply a sequence of letters and digits; in other words, it can include the characters A through Z, a through z, 0 through 9, underscore (_), dollar sign (\$), and commercial at sign (@). A C identifier must begin with a letter, \$, _, or @.

The following are all legal identifiers in C:

```
jonb
jonbl
JONBl
JONBl@
JON_Bl$
```

A C identifier can have up to 32 significant characters; any characters after that will be ignored by the compiler. The /UNIQUE switch on the C compiler lets you specify the length after which the compiler ignores characters; the range for /UNIQUE=nn is 8 through 32 characters. The compiler switches are described in Chapter 11.

Reserved Words (Keywords)

=====

Some identifiers in C are reserved. You can use them for only one thing, and you can't redefine them for your own use. The table below shows all the keywords in the C language. Chapter 7 describes each C keyword in detail.

The C keywords let your program do some of the most basic things in C, like declare data, manipulate data, and direct the flow of the program.

The following list does NOT represent the whole C language; it doesn't include any operators or punctuation, and it doesn't include any of the C runtime library routines, which are an important part of C.

C keywords:

auto	else	int	switch
break	entry	long	union
case	enum	register	unsigned
char	extern	return	void
continue	float	short	while
default	for	sizeof	
do	goto	static	
double	if	struct	

Keywords added by the Data General C Compiler:

\$asm
\$builtin
\$fortran
\$pascal
\$pll

C Runtime Library Routines: Are They Reserved?

The C runtime library is full of useful routines. Their names are not reserved words, but it's still possible to unintentionally confuse them.

For example, a C program normally gets characters from the standard input device, which is your terminal, by issuing a call to the `getchar()` routine. The program doesn't need to declare this routine or define it, because it's part of the C library.

However, if the program provides a `getchar()` routine of its own, and then tries to call `getchar()`, intending to use the one from the runtime library, the C compiler will use the program's own routine. Furthermore, the compiler usually won't post any error messages, unless the routine happens to be implemented by means of a C macro. In fact, `getchar()` is implemented as a C macro.

So: to make sure you're using the function you want, check all your function calls to see that they don't conflict with the C runtime routines.

The runtime routines included with the Data General C compiler are documented in Chapter 10.

Constants

=====

In C, you can create four kinds of constants:

Kind of Constant	Example
-----	-----
strings	"i am a string"
character constants	'a', '\033'
integer constants	100
floating-point constants	2.71828

You can assign constants to variables:

```
char testchar;  
testchar = 'b';  
  
int buffer;  
buffer = 10;  
  
double limit;  
limit = .30103;  
  
char string[];  
string = "i am a string"
```

And you can send constants to functions:

```
printf ("Filename not found.\n");
```

Of course, it's possible to bury the constants in your program. To avoid the problem of having to change them all later, you can use symbolic constants, described in the "Symbolic Constants" section below.

The remaining parts of this section describe the rules for constants in C.

How to Specify the Radix of a Constant

Constants can be decimal, octal, or hexadecimal.

To make a constant's radix decimal, leave the constant as is.
For example,

```
65      is the decimal number 65
```

To make a constant's radix octal, put a leading zero on an int constant.
For example,

```
065     is the octal number 65
```

To make a constant's radix hexadecimal, put a leading 0x or 0X on an int constant. For example,

0XA is the hex number A (decimal equivalent is 10).

Octal and hex constants can also be long; long constants are explained in the next section.

Character Constants

Define character constants by surrounding them with single quotes, as in the following examples:

```
'A'           the uppercase A character
'8'           the numeral eight

letter1 = 'a'; /* declare the character variable, */
               /* letter1; initialize it to 'a'. */
```

The backslash stands for the escape character; in other words, backslash lets you produce escape sequences:

```
'\014'        the form feed character
'\0'          the null character
'\\'          the backslash character
'\307'        a bit pattern
```

Other possibilities for escape sequences in C are:

Escape Sequence	Character	Mnemonic

\n	NEW LINE	NL
\t	horizontal tab	HT
\b	backspace	BS
\r	carriage return	CR
\f	form feed	FF
\\	backslash	\ (backslash)
\'	single quote	' (single quote)
\"	double quote	" (double quote)

You can also set a character constant to an octal number:

ddd bit pattern, in octal

For example,

```
#define FLAG1 '177'
```

sets FLAG1 equal to the octal number 177.

String Constants

Use the double quotes to define string constants. For example:

```
string1 = "i am a string"; /* set string1 to be "i am ...". */
printf ("Filename not found.\n");
                        /* print "Filename ..." on output. */
```

The `\n` sequence is a NEW LINE character.

The C compiler sees all strings as distinct, even when the letters between their quotes are identical.

C puts a null (`\0`) at the end of every string, so that a program can find the end of a string.

To put a tab into a string, use the backslash-t sequence. For example, in this string:

```
"Daily Balance\tWeekly Balance\tMonthly Balance"
```

the tabs will occur between Balance and Weekly and between Balance and Monthly.

Exponents and Scientific Notation

C makes all constants that are real numbers double-precision.

To show the exponent part of a real number constant, use the letter `e` or `E`. `E` is for use with double-precision constants only. Examples:

```
626.262e-3          is 0.626262
0.5454e6            is 545400.0
```

You can omit either the integer part or the fractional part, but not both.

Long Integer Constants

Using the letter `L`, either uppercase or lowercase, you can specify an integer constant as long:

```
20L, 20l
```

In 16-bit implementations of C, the arguments to certain functions like `fseek()` must be long integers.

Symbolic Constants

Instead of burying explicit constants in your code, you can bury symbols for them. To create such a symbol and set it to the number -- or string, or whatever you want -- you use the `#define` preprocessor statement; for example:

```
#define BUFSIZE 1000
#define USERNAME "Nitram"
#define LOGTWO .30103
```

The names that `#define` can use must follow the same rules as other C identifiers: see the "Rules For All C Identifiers" section above; and the constants that `#define` can create must follow the same rules as other constants.

Operators

=====

Like the operators in other languages, the operators in C perform the functions you would expect: addition, subtraction, and so on.

C Operator	Example	Function
-----	-----	-----
+	a + b	add a to b
++	a++	increment a by 1
=	a = b	set a equal to b
+=	a += b	set a equal to a plus b

C also offers some special operators, like sizeof, and ?: .

sizeof	sizeof(tnode)	give the size in bytes of the data type tnode
?:	expression-1 ? expression-2 : expression-3	if expression-1 is true, perform expression-2; else, perform expression-3.

By grouping identifiers together, the C operators form expressions.

For the purposes of our discussion of C tokens, it's useful to think of the C operators as signals to the C compiler to start grouping identifiers into expressions according to each operator's rules. Chapter 4 describes expressions in C and the rules for forming expressions.

For example, in this string of C tokens

```
while (c=getchar() != EOF)
```

the operators are = and !=. Since != has higher precedence than =, the compiler groups identifiers around it first, as

```
while (c = (getchar() != EOF))
```

The C operators are:

+	&	^		?:	--	/	=
==	>	>=	++	*	!=	<<	<
<=	&&	!		%	*	~	>>
.	->	-	-	sizeof	(type-name)		

Chapter 8 is a dictionary of the C operators; it explains in detail how each C operator works.

Token Separators

=====

The token separators in C are:

```
space
tab
vertical tab
NEW LINE
form feed
comment      /* like this */
```

The C compiler is not fussy about spaces tabs and NEW LINES; you can use them as you would in other languages to make your code clean and readable. In most cases the C compiler can tell where one token ends and the next begins. However in some cases, a token separator is necessary; for example, following a keyword, if the next character is alphabetic, a separator is necessary to delimit the identifiers.

A rule of thumb for C token separators is: wherever you can put a space, you can also put a tab, NEW LINE, form feed, or comment.

Preprocessor Delimiter

The preprocessor delimiter is `#`. Any statement beginning with a `#` and ending with a NEW LINE talks to the preprocessor and thus only indirectly to the compiler. Examples:

```
#define TABLESIZE 100    define a constant called TABLESIZE
```

A preprocessor `#define` has scope through the remainder of the program file in which it appears, until an `#undef` revokes it.

```
#include <stdio.h>        include the contents of the file stdio.h
                           into the program code
```

```
#nolist                   don't print out these statements when
( statements )             listing the program
#list
```

Statement Delimiters

There are two kinds of statement delimiters in C: one kind delimits simple statements and the other delimits compound statements.

To delimit a simple statement, put a semicolon at the end. Examples:

```
c = getchar();

s[i] = c;

while ((c=getchar()) != EOF)
    s[i] = c;
```

To delimit a compound statement (i.e., two or more simple statements, use the braces, { and } . Example:

```
while ((c=getchar()) != EOF)
{
    s[i] = c;
    i++;
}
```

You need the braces there because the while loop encompasses more than one statement.

Comments

The comment delimiters in C are /* and */. To put a comment into the code, surround it with stars and slashes:

```
a = b + c;                /* Comment goes here. */
```

Comments can act as token separators.

End of Chapter

CHAPTER 3

DECLARATIONS, DATA TYPES, DATA TYPE CONVERSIONS, AND DATA INITIALIZATIONS

In C, as in many other programming languages, you must declare objects before you can use them. The declarable objects in C are data, data types, and functions (code). When you declare functions, you simply identify the kind of data the function returns: an integer, an array, etc. We explain how to declare functions in Chapter 6. This chapter explains how to declare objects in C, how to use the C data types, how to convert data from one type to another, and how data is initialized in C.

Declarations

=====

C looks at declarable objects along three dimensions: a data type, a storage class, and a language type.

- * The data type shows the limits of the data. It answers the usual questions about data: Are the data numbers? Letters? A combination thereof? How large is the data? How precise is the data? And so on. In C, the default data type is integer, or int.

```
int a;          /* declares an integer named a.    */
```

Specifying the data type of an object also limits its use. A function that accepts only integers won't be able to tamper with floating-point numbers. However, C allows many data type conversions, which are described later.

- * The storage class of an object determines its permanence and how widely it can be used; in other words, it determines the object's scope.

Storage class also limits the permanence of an object. For example, an auto object will cease to exist between executions of the function it's in; a static object will continue to exist.

```
static int a;   /* declares a static integer named a. */
```

- * The language type of an object shows the AOS/VS language with which the object is to interface. The language type keywords are Data General extensions to C; they are described in Chapter 6.

When you declare variables, you can also give them initial values in the same declaration statement:

```
int a=1, b=1, c=2, d=3; /* declare and init. a, b, c, d */
```

Later, you can convert your data into data of other types. For example, when dividing two integers, you may need a floating-point number as a result.

Syntax of Declarations

All C declarations have the form:

```
specifier-list declarator-1, declarator-2 ... declarator-n
```

where specifier-list is a data type and/or a storage class and/or language type, and declarator-1 through declarator-n are identifiers that are to be of that type and class. For example,

```
static long int a, b, henry, henry2;
```

declares a, b, henry, and henry2 all to be long integers with static storage class.

Where Can Declarations Occur?

Declarations must occur either

- * at the beginning of blocks of code, including blocks that constitute functions, or
- * anywhere outside all functions.

The following sections in this chapter describe the data types and storage classes in C, how data type conversion proceeds in C, and how to initialize data in declaration statements.

Data Types

C has a small number of data types: char, int, float, and double. Using these four fundamental data types, however, it is possible to make up an infinite variety of combined or derived types.

The C keywords for specifying data type are:

char	character
int	integer
float	floating-point (i.e., real) number
double	double-precision (i.e., floating-point) number
short	an adjective, for int, float, and double
long	an adjective, for int, float, and double
unsigned	an adjective, for char, int, float, and double

The derived data types in C include:

arrays	May contain the other types, including other arrays, structures, pointers, etc.
pointers	May point to anything, including other pointers and functions. Anything that has a type can have a pointer to it.
structs	structs are like records in Pascal, like structures in PL/I. In C, you can use structs to build linked lists; structs are useful for grouping data that belongs together.
unions	unions are like pure memory; you can store data of mixed types in a union.
enums	enums are like scalars in Pascal; they permit a type that consists of a range of constant values.
functions	Functions, which are declared in terms of the kind of data they return, can return any data type, but not arrays or other functions.
typedef types	typedef creates a synonym for an already-declared type or standard C data type.

The following sections describe these types; Chapter 6 describes functions.

Remember, you can also declare an object's storage class; storage classes are described in the next major section.

Fundamental Data Types

In effect, there are only two kinds of data in C: integers and floating point numbers:

- * integers include the types `int` and `char`. (Remember that arrays of `int` and `char`, and pointers to `int` and `char` are part of the integer world.)
- * floating-point numbers include the types `float` and `double`.

The Adjectives `short`, `long`, and `unsigned`

You can modify the fundamental data types using the keywords `short`, `long`, and `unsigned`. As their names imply, `short` and `long` determine the amount of storage that C allocates for such objects. `unsigned` determines whether the object can accept signed data or not.

For complete details on each of these adjectives, please see the section for that adjective in Chapter 7.

int

Integers are the workhorses of C. And int, which means integer, is the default type for all data and functions in C. You can use all three data type adjectives with int: short, long, and unsigned.

In C, integers are useful for counting things, pointing to things, and storing characters, especially negative ones.

The following list shows the size limits that integers, floats, doubles, and chars enjoy in C:

int	32 bits
short int	16 bits
long int	32 bits
unsigned int	32 bits
unsigned short int	16 bits
unsigned long int	32 bits
float	32 bits: 23 mantissa + 8 exponent + 1 sign
short float	same as float
long float	same as double
double	64 bits: 55 mantissa + 8 exponent + 1 sign
short double	same as float
long double	same as double
char	8 bits

Examples:

```
int thing;          /* declares an integer named thing.      */
int *pthing;        /* declares a pointer named pthing to an      */
                   /* integer not specified yet.                 */
pthing = &thing;     /* sets the integer-pointer pthing to the      */
                   /* address of the integer thing.              */
int *athing[3];      /* declares an array, named atthing, of        */
                   /* three pointers to integers.                */
int (*pathing)[];    /* declares a pointer, named pathing, to      */
                   /* an array, not specified here, of          */
                   /* integers.                                  */
pathing = atthing;   /* sets the array-pointer, pathing, to        */
                   /* point to the array atthing.                */
```

```

int *fthing();      /* declares a function, named fthing, that */
                   /* returns a pointer to an integer.      */

int (*pfthing)();   /* declares a pointer, named pfthing, that */
                   /* points to a function, not specified here, */
                   /* that returns an integer.                */

```

char

char gives a variable a single byte of storage, 8 bits long. Your program can use a char variable to store any one of the standard ASCII characters. You can use one of the data type adjectives with char: unsigned.

The value of a char variable is the integer code for the character it contains. With Data General C, the default range of possible values is 0 to +255. If you use the /SIGNEDCHAR switch on the C compiler, char variables, but not unsigned char variables, will have a range of -128 to +127.

char is useful for declaring character arrays, pointers to characters, and functions that return characters. Since it's possible to get the integer code for a character, you can do relational operations (a is less than b, c is greater than d, etc.) with char variables, and this is useful for comparing and alphabetizing arrays of characters.

Examples:

```

char thing;         /* declares a one-character variable named */
                   /* thing.                                */

char *pthing;       /* declares a pointer, named pthing, to a */
                   /* character.                            */

pthing = thing;     /* sets the character-pointer pthing to */
                   /* point to the character thing.        */

char athing[];      /* declares an array, named athing, of */
                   /* characters.                          */

char fthing();      /* declares a function, named fthing, that */
                   /* returns a character.                  */

char *fpthing();    /* declares a function, named fpthing, that */
                   /* returns a pointer to a character.      */

char thing = '\n';  /* declares thing as a character and */
                   /* initializes it to NEW LINE character. */

```

float

float means a floating-point number; that is, a number that has a fractional part. You can use the data type adjectives short and long with float.

The following list shows the size limits that integers, floats, and doubles (described in the next section) enjoy in C:

int	32 bits
short int	16 bits
long int	32 bits
float	32 bits: 23 mantissa + 8 exponent + 1 sign
short float	same as float
long float	64 bits: 55 mantissa + 8 exponent + 1 sign
double	same as long float
short double	same as float
long double	same as long float

float is useful for making your program use less memory; it is also useful for fitting numbers to existing conventions.

NOTE: C carries out all floating-point arithmetic in double-precision. C lengthens all floats that appear in expressions by padding the fraction on the right with zeros. This means that you get the full benefit of C's accuracy during computations. After the computation, however, C will store the result back into whatever variable is called for, and this may result in rounding of floats or truncation of ints.

Rules for converting float data to double-precision data and vice versa are listed below in the section "Data Type Conversions."

double

double makes an object a double-precision floating-point number. You can use the data type adjectives short and long with double. double is useful for large floating-point numbers. The limits of double variables are shown in the list in the previous section, "float".

Derived Data Types

The derived data types are those composed of more than one object or more than one kind of object. These objects can be any of the data types or storage classes. The derived types include:

- * arrays
- * pointers
- * structs
- * unions
- * enums

The following sections describe how to declare them.

Arrays

For any data type, you can create an array of data of that type:

<code>int a;</code>	an integer variable named a
<code>int a[];</code>	an array, named a, of integers
<code>float b;</code>	a floating-point variable named b
<code>float b[LIMIT];</code>	an array, named b, of LIMIT floating-point numbers
<code>float b[10][20];</code>	a two-dimensional array, named b, of 200 floating-point numbers
<code>struct c { int c1; int c2; };</code>	a structure type named c, containing two members: c1 and c2
<code>struct c carray[10];</code>	an array, named carray, of 10 c-type structures

Exception: You can't have an array of functions, although you can have an array of pointers to functions.

In C, the elements of an array are numbered starting with 0. To refer to a particular element of an array, fill in the brackets:

<code>necklace[21]</code>	gets the twenty-second item in the array necklace.

You can also use constants, variables, and expressions in the brackets, as long as they resolve to integers.

In C, the name of an array is also a pointer to the first element of that array. This is very useful; you can use the array name as a base for offsets into the array:

```
v = marquee[3];           /* set v equal to the fourth element */
                           /* in the array marquee.          */

v = *(marquee + 3);       /* marquee points to 1st element,   */
                           /* marquee + 3 points to 4th element, */
                           /* so set v equal to 4th element.    */
```

Since the first element of a C array is always element 0, both statements set v to the same thing, the fourth element of the array marquee.

In general, you must always declare the number of elements that an array will have. You do NOT have to declare this when:

- * The array is external; i.e., it is defined outside the current compile file.
- * You're defining only a pointer to an array.

```
char suki[];
```

- * You're initializing the array; i.e., filling it with values.

```
char peter[] = "Hi, Petey.\n";
```

- * You're using typedef to create a new name for an array.
- * The array has more than one dimension; you may only omit specifying the bounds of the first dimension, and the array must meet one or more of the preceding conditions. Example:

```
extern int traffic[][16][255];
```

This declares a three-dimensional array, named traffic, of integers.

Pointers

In C, pointers represent addresses in the computer's memory, that is, in your C program's address space. The pointer operators are `&`, the address operator, and `*`, the indirection operator.

`&identifier` gives the address of identifier

`*identifier` makes identifier a pointer

For any data type, you can create a pointer to data of that type by using the indirection operator, `*`. You must supply the type of the data to which the pointer is supposed to point. The following examples show how to use `*` and `&`.

```
char x;           /* declares a character variable named x. */
char *px;         /* declares a pointer, named px, to a      */
                  /* character not specified here. */
px = &x;          /* sets the pointer px to the address of x. */
```

As we said above, the name of an array in C is also a pointer. To obtain this pointer, declare the array and a pointer variable, then assign the array name to the pointer:

```
char *peachp;     /* declares a pointer, names peachp,      */
                  /* to a character not specified here. */
char peaches[10]; /* declares an array, named peaches, of  */
                  /* 10 characters. */
peachp = peaches; /* makes peachp point to the first      */
                  /* element of the array peaches. */
```

Here are some more examples of pointers and arrays:

```
int *ppeach[3];   /* declares an array, named ppeach, of  */
                  /* three pointers to integers. */
int *fpeach();    /* declares a function, named fpeach, that */
                  /* returns a pointer to an integer. */
int (*pfpeach)(); /* declares a pointer, named pfpeach, that */
                  /* points to a function, not specified here, */
                  /* that returns an integer. */
```

structs

struct is one of the derived data types, which include arrays, structs, enumerations, unions, and synonyms from data types that you create with typedef. structs are useful for defining aggregate data. By aggregate data we mean data that is composed of pieces of data of different types; by contrast, arrays are made up of pieces of data of the same type. structs are like records in Pascal, and like structures in PL/I.

structs are useful for linked lists, trees, and other dynamic data structures. structs also allow you to create a unit of storage to be used bit-by-bit (such units are called fields in C).

Declare structs using the following syntax:

```
struct structure-name
{
    member-list
}
variable(s) of this type;
```

The structure-name and list of variable of this type are optional; the braces are not necessary if the structure has only one member. Note that

```
struct { member-list } identifier;
```

declares an identifier to a be structure, but doesn't give the type a name, while

```
struct structure-name { member-list };
```

declares a structure type but doesn't create any variables of that type.

You can have self-referential structures; a member of a structure can be a pointer to another structure of the same type. The pointers are useful as links in a linked list.

An example of a field:

```
struct flagz {
    unsigned harpo    : 1;
    unsigned chico    : 1;
    unsigned groucho  : 6;
} opera;
```

This defines a field variable called opera (the structure type is flagz) with two one-bit fields and a six-bit field; the number after the colon specifies the width in bits.

To turn the bits on and off, use assignment statements:

```
opera.groucho = 014;
```

Whenever you mention a variable that is a struct, you must precede it with the struct keyword, as in

```
d = a + struct opera;
```

DG implementation differences:

1. Your C program can copy one structure into another, as in:

```
bridge = tunnel;
```

where bridge and tunnel are both the same kind of structure.

2. Your C program can pass structures to functions. The passage takes place by value. The function can return the structure.
3. Your C program can have a member of a structure that is a pointer to another, as-yet-undeclared structure. The program must declare the undeclared structure by the end of the current block, or before the beginning of a function. This is useful if you have two structures that contain pointers to each other. An example of a node in a binary tree:

```
struct treenode
{
    char[12] leaf;
    int    leafcount;
    *treenode leftnode;
    *treenode rightnode;
} treel;
```

To declare two more nodes:

```
struct treenode tree2, tree3;
```

unions

union is one of the derived data types, which include arrays, structs, enumerations, and unions. unions are useful for holding, at different times, objects of different types and sizes. The C compiler keeps track of size and alignment requirements inside the union.

Declare unions using the following syntax, which is the same as the syntax for structs:

```
union union's-name
{
    list of union members
}
list of variables of this type;
```

The union's-name and list of variables of this type are optional.

DG implementation differences: Your program may initialize a union. The C compiler assumes that the program is initializing the first member of the union.

Example: Here is a union called student, with three members: a, b, and c.

```
union student
{
    int a;
    double b;
    char[] c;
}
```

Now we declare mike to be a student; and then we initialize mike:

```
static union student mike = { 2 };
```

This will set mike.a equal to 2.

enums

enums provide a means for making scalar variables. enums are like user-defined types in Pascal.

Unlike the other derived types in C, the elements of an enum are set to constants. The enum declaration specifies identifiers for these constants.

The identifiers, in turn, must be set to constant values; they can either be set in the declaration or later in the program.

enum values have scope only in the functions in which the enum type is declared. By contrast, #define constants have scope throughout the remainder of the program file.

enum is useful for creating lists of constants; enum is an alternative to #define.

Example:

We declare our enumerated type, rainbow, as having a range of possible values from "radio" to "xray"; we also initialize radio to 10 and infrared to 25:

```
enum rainbow {radio=10, infrared=25, visible, ultrav, xray};
```

NOTE: If you don't give a value to an enum constant, C gives it the value of the previous constant plus 1. The default value for the first element of an enum is 0. In the example above, the program will take visible to be 26, ultrav to be 27, and xray to be 28.

We declare a variable band as being of type rainbow; in other words, band can ONLY have one of the five values in the brackets above.

```
enum rainbow band;
```

Finally, we initialize band:

```
band = visible;
```

Or we can declare and initialize everything in one statement:

```
enum rainbow {  
    radio=10,  
    infrared=25,  
    visible,  
    ultrav,  
    xray  
} band=visible;
```

Scope of Variables

=====

How you declare the variables in your program determines their scope. By scope we mean the ability of the variable or function to refer to and be referred to by other functions and variables in the rest of the program and in other programs.

- * An identifier declared outside all functions has scope throughout the rest of the program, including all functions. It can have scope outside the program, i.e., in other compiled files, if the storage class is extern; C makes identifiers declared outside all functions extern by default.

- * A variable has scope throughout the function in which it is declared. If the variable is of storage class `extern`, C extends the variable's scope to the end of the current compile file.
- * The formal parameters of a function have scope throughout that function.
- * The scope of an identifier declared at the head of a block has scope throughout the rest of that block; the scope of other variables of the same name is suspended until the end of that block.
- * Labels have scope throughout the entire function in which they appear.
- * All functions have scope throughout the entire program.
- * The `extern` storage class specifier causes identifiers in different places to refer to a single common object. In other words, `extern` gives variables and functions scope outside the program.

The typedef Keyword

=====

The `typedef` keyword lets you create synonyms for C data types and also for your own data types definitions. You can use these synonyms later to declare things -- as if the synonyms were keywords naming data types. For example:

```
typedef int MYINT;
```

lets you use the identifier `MYINT` to stand for `int` in all subsequent declarations in the program.

`typedef` is useful for abbreviating long declarations. It is also useful for setting up a program's code to be more easily portable to another computer.

Remember, `typedef` does NOT create a new type, it simply gives a synonym for an existing type.

For complete information on `typedef`, see Chapter 7.

Data Type Conversions

=====

Sometimes it is useful or necessary to convert data from one type to another. This can occur when a program assigns one variable to another, or when it passes arguments to functions.

This section describes several ways by which type conversions can occur in C:

- * the "Usual Arithmetic Conversions"
- * conversion by assignment
- * conversion by passing arguments
- * casts

The Usual Arithmetic Conversions

In general, C will automatically convert one data type to another using common-sense rules. For example, if `abci` is a float and `ordi` is an integer, in the expression

`(abci + ordi)`

C makes the entire expression float, to allow for `abci`'s fractional part. Similarly, C doesn't permit expressions that don't make sense, like using a float as a subscript.

Here are the "usual arithmetic conversions" in C:

- * C allows you to combine chars and ints freely in arithmetic expressions. The compiler automatically converts every char in an expression to an int. See below for more information on sign-extension of characters.
- * Relational expressions (like: `px > py`) and logical expressions (like: `a && b, c || d`) have the value 1 if true and 0 if false. The type of this value is int.

- * In arithmetic expressions where the operands are of mixed types, C follows the common-sense rules we mentioned above, in the following order.

First:

char and short int	convert to	int
float	converts to	double

Then:

if either operand is double, the other operand converts to double, and the result is double.

Otherwise:

if either operand is long int, the other operand converts to long int, and the result is long int.

Otherwise:

if either operand is unsigned int, the other operand converts to unsigned int, and the result is unsigned int.

Otherwise:

it follows that both operands are simply int; the result remains int.

Conversion by Assignment

In assignment expressions and statements, C converts the value on the right side of the assignment expression to the type of the left. The type of the left is the result.

Type conversions take place across assignments, as follows:

- * When a C program assigns a char to an int, C converts the char to an int first, by sign extension if necessary; see below for information about sign-extension of characters.
- * When a C program assigns an int to a char, C discards excess high-order bits in the int.
- * When a C program assigns a float to an int, C truncates the fractional part of the float.
- * When a C program assigns an int to a float, the result is float.
- * When a C program assigns a double to a float, C rounds the double before making the assignment.
- * When a C program assigns a long int to a short int, C truncates the excess high-order bits before making the assignment.

Passing Arguments to Functions

Type conversions also take place when arguments are passed to functions. C makes the following conversions before passing the arguments to the function:

char and short int	become	int
float	becomes	double

Casts

In addition, you can also force a type conversion with a cast. As an example, suppose we want to hand an integer, *i*, to a routine that expects a double. We can convert the integer by applying the data type double to it in parentheses, then issuing the call:

```
doubroutine((double) i);      /* cast i as a double.      */
```

A cast does not alter the type of *i*, only the type of the value. In this case, it alters the type of the value that is passed to *doubroutine*.

Sign Extension

=====

Sign extension can occur when C must convert a char into an int; for example, when the program attempts to store a character in an int variable.

By default, i.e., if you don't use the `/SIGNEDCHAR` switch on the C compiler, C guarantees that any character in the machine's character set will always be positive. This means characters, being 8-bit objects, will have a range of possible values from 0 to +255 for your program. Under these circumstances, when the program stores a char into an int variable (int variables being at least 16 bits), C will pad the left-most bits of the int with zeros, even if the character's leftmost bit happens to be 1.

On the other hand, if you use the `/SIGNEDCHAR` switch, the range of possible values for characters is -128 to 127. The characters will still be 8-bit objects. In ASCII, all the graphic characters will be positive as usual, but this option lets your program handle negative characters like EOF, which will be -1. Under these circumstances, when the program stores a char into an int variable, if the char is either negative or has its leftmost bit set to 1, C will pad the leftmost bits of the int variable with ones.

NOTE: Sign extension also applies to characters that the program specifies with an octal escape:

`'\377'`

has the value -1, providing you used the `/SIGNEDCHAR` switch on the compiler.

To avoid unwanted sign extension, use unsigned char variables instead of char variables to store characters. This will let your program handle all the characters in the character set, without their being sign-extended.

Storage Classes

=====

In C, an object can be in one of four standard storage classes: auto, static, register, and extern. It can also belong to a storage class whose name you created with typedef. And Data General C offers a special storage class, `$builtin`, which is for certain C runtime functions. `$builtin` is described in Chapter 7, and the C runtime routines are described in Chapter 10.

The storage class determines the permanence and scope of an object. For example, storage class `extern` lets object be used by other modules, outside the compiled program file. The other storage classes limit the object's scope to the enclosing function, or if the object is outside all functions, to the program file.

C allows only one storage class specifier per declaration.

The following sections describe the storage classes in C.

auto

`auto` stands for automatic. An automatic variable is one that exists only while its enclosing function is being executed. When the program exits that function, all the function's auto variables cease to exist.

`auto` is useful for local variables that don't need to be saved from one invocation of a function to the next.

The `auto` keyword isn't necessary in most declarations, since C makes all variables `auto` by default, except those variables that lie outside all the functions. Those variables default to `extern`.

`auto` variables are advisable for recursive functions; otherwise, each new iteration of the function will be able to overwrite local data. For example, this gives the variable `kris` storage class `auto`:

`auto kris;`

static

The complement of auto is static, which lets a function keep a variable around between executions of that function.

static makes the variable keep its value, even when the program isn't executing the block or function in which the variable occurs. static variables are useful as private permanent storage for a function.

NOTE: When you declare a static variable inside a function, the variable's scope is just that function, and the variable is said to be an internal static variable.

When you declare a static variable outside all the functions, its scope is the rest of the source file, and it is said to be an external static variable.

Don't confuse these distinctions with the storage class extern; extern lets a variable have scope outside the entire compiled file, regardless of whether it is declared inside a function or not.

External static variables won't conflict with the same names in other files of the same program. By contrast, the link utility WILL compare an extern variable to any other extern variable it finds with the same name in the same program.

Example:

These variables were used by two different functions that touched the same buffer; they were declared outside the functions:

```
static char temp[TEMPSIZE]; /* declare an array of chars. */  
static int  temp = 0;        /* declare an int.           */
```

register

If your C program declares a variable to be register, please note that the Data General C compiler treats all register variables as auto.

extern

The extern storage class is useful when your program consists of several modules that use the same variable. One module contains the declaration; the other modules that mention the variable declare it to be extern.

Using Data General C, the following declaration

```
extern notepad;
```

will cause the compiler to make the identifier notepad an integer. Furthermore, by default, i.e., if you don't use the /EXTL switch on the compiler, the compiler will generate storage for notepad. All extern variables with the name notepad that are in the same program will get the same storage location. On the other hand, if you use the /EXTL switch, you must declare notepad's data type somewhere, either in one of the other files to be compiled, or in assembly language.

You can repeat the declaration -- in other words, you can declare notepad as an extern variable twice in two different places -- but notepad's data type must be the same everywhere.

You CAN declare variables that are inside functions to be extern.

Example:

Let's say we compile our program from three separate source files: moduleA, moduleB, and moduleC. All three use the variable notepad to communicate with each other. We can declare notepad fully in moduleA (it's an integer), then mention it to be an extern in moduleB and moduleC:

ModuleA

```
int notepad;
```

ModuleB

```
extern notepad;
```

ModuleC

```
extern notepad;
```

typedef

For syntactic purposes, typedef is often thought of as a storage class. typedef simply creates a synonym for an existing type.

Typedef is described earlier in this chapter, and in detail in Chapter 7.

`$builtin`

Data General C includes a special storage class called `$builtin`. You use `$builtin` to declare certain functions in the Data General C runtime library; `$builtin` causes the compiler to build more efficient code. For information on which functions to use `$builtin` with, see Chapter 10.

Some Tips on Storage Classes

- * The storage class you'll probably use the most often is `auto`; `auto` is the default, and you almost never need to specify it.
- * To refer to or use an object that is declared fully elsewhere, use `extern`. Any identifier that is declared outside all functions is `extern` by default. All functions are `extern` by default.
- * The storage classes `auto`, `static`, and `register` all have local scope. Only `extern` has broader scope.
- * It's possible to create global variables and have C generalize them at link time without declaring them explicitly as `extern`. Make sure the declarations of all such shared variables are outside all functions in their respective modules, and that the same variables are declared the same way everywhere. When the modules are linked together, C will assume these variables all refer to the same variable.

The danger in this method is that it can work when you don't intend it to: variables that were NOT intended to correspond can get connected.

Initializations and Defaults

In C, you can initialize a variable in the same statement you use to declare it. Examples:

```
int planets = 9;    /* declare and init. the integer planets. */
```

This creates an integer called `planets` and sets its initial value to nine.

```
char slogan[23] = "I am a short message.\0";  
                /* declare and init. the char. array slogan. */
```

This creates an array of 23 characters and puts into it the characters `I a m a s h o r t m e s s a g e .` and a null at the end.

Notice that with strings, C automatically loads the array; you don't have to load it character-by-character yourself.

Data that is storage class `extern` gets initialized automatically as follows:

```
data type int    zeros
data type char   nulls
```

Data that is storage class `register` or `auto` does not get initialized automatically.

Default Data Types and Storage Classes

If you don't fully declare an object, C uses the following defaults:

- * Functions default to data type `int`, storage class `extern`.
- * Data that lies within `main` or within another function defaults to data type `int`, storage class `auto`.
- * Data that lies outside all functions defaults to data type `int`, storage class `extern`.

End of Chapter

CHAPTER 4

EXPRESSIONS

This chapter describes how to form expressions in C.

As we said in Chapter 2, expressions are the next largest size of building blocks in C: you use the C character set to form tokens; and you combine the tokens to form expressions. The specific kinds of tokens that form expressions are identifiers and operators.

What are expressions useful for? You use expressions to form statements; and you group the statements together into blocks of code.

This chapter briefly describes the C operators, which are described in more detail in Chapter 8. Then it describes two levels of expressions: lvalues and primary expressions, and other expressions.

C Operators

=====

Each C operator takes a specific number of arguments (identifiers), either one, two, or three; hence the names unary operator, binary operator, and ternary operator.

The C compiler groups some operators before others; in other words, there is an order of precedence among the operators. You can control the precedence with parentheses.

Before examining operators and expressions, it's useful to understand the precedence of the C operators.

Precedence

Precedence means the order in which the compiler groups operands to operators. The C compiler evaluates certain operators and their operands before others. If you haven't grouped your operands the way you want, by using parentheses, the compiler will group them according to its own rules. For example, the operators `<` `>` `<=` and `>=` have lower precedence than the arithmetic operators, `+` `-` `*` `/` and `%`. So in the expression

```
r < s - 1      /*                      */
```

since the `-` has higher precedence, C groups it this way:

```
r < (s - 1)    /* subtract 1 from s; compare result to r. */
```

For example, in the C statement

```
while (c=getchar() != EOF)
;
```

since the not-equal operator `!=` has higher precedence than the assignment operator `=`, the compiler will group operands around the `!=` first, resulting in this statement:

```
while (c= (getchar() != EOF))
;
```

This is not what the original statement was intended to mean; it was intended to mean:

```
while ((c=getchar()) != EOF)
;
```

Thus, in this case, you must put in the parentheses in order to override the precedence.

It's useful to look a bit more closely at the effects of grouping operands around operators.

Grouping

When we say grouping with respect to C operators, we are talking about how the C compiler associates operators with the nearest operands. This grouping of operators and operands implies an order of evaluation, in which the operators of the highest precedence get operands grouped to them first, and get evaluated first.

When we say an operator groups left-to-right, we mean that in an expression involving a chain of the same operator or operators of the same precedence, like

`a + b + c + d`

the C compiler will group the leftmost first, as follows:

`((a + b) + c + d`

Chapter 8 contains a table that lists the operators in order of precedence; it also shows the kind of operands they take, and the associativity (grouping) of the operands.

C Expressions

A C expression is a group of operators and identifiers that function as a unit. Operators can also combine expressions into larger expressions and, ultimately, into statements.

This example shows C tokens forming a larger and larger expression:

```
c
c=getchar()
((c=getchar()) != EOF)
((c=getchar()) != EOF && (c != '\n'))
```

Don't get confused by the terms unary and binary with respect to operators: a unary operator is one that takes one argument.

```
*expression                                /* indirection op. */
```

A binary operator is one that takes two arguments.

```
expression + expression                    /* addition op.    */
```

Only one operator takes three arguments, the conditional operator.

```
expression ? expression : expression      /* conditional op. */
```

Lvalues

The name lvalue, in which the letter l refers to left, comes from the assignment statement, in which the operand on the left must be an lvalue expression, i.e., must resolve to an object. In C, an object is a manipulatable region of storage; and an lvalue is an expression that refers to an object.

The following things can be lvalues:

identifier	e.g. integer, array, function call
primary-expression.[expression]	element in an array
lvalue.identifier	member of a structure
primary->identifier	member of a structure
*expression	a pointer
(lvalue)	an "adorned" lvalue

Things that resolve to arrays (as opposed to the first element of an array) and function names are not lvalues.

NOTE: An lvalue doesn't have to actually be on the left side of an assignment statement; but it has to BE ABLE TO BE on the left.

Primary Expressions

The term primary expression is used in the definitions of various kinds of C expressions. The following things qualify as primary expressions:

- identifier
- constant
- string
- (expression)
- primary-expression [expression]
- identifier (expression-list)
- lvalue.identifier
- primary-expression -> identifier

In which an expression-list can take the following forms:

- expression
- expression-list, expression

Kinds of Expressions

This section contains a survey of some of the familiar categories of expressions in C:

- * Expressions formed by brackets, parentheses, structure pointers, and structure-member operators
- * Increment and decrement expressions
- * Arithmetic expressions
- * Relational and logical expressions
- * Bitwise logical expressions
- * Assignment expressions
- * Expressions formed with the conditional operator

For details on each operator, and for a complete table listing the arguments for each operator and its precedence, see Chapter 8.

Brackets, Parentheses, Structure Pointers, ----- and Structure-Member Operators -----

We group these operators and symbols together simply because they have the highest precedence of all the C operators.

Brackets serve to delimit the bounds of an array, as in

primary-expression [expression]

Parentheses are used to group operands and operators together:

```
for (i=0; (((c=getchar()) != EOF) || (c != \n)); i++)
    airplane[c] = noclouds;
```

The structure member operator fetches a member of structure. We'll use this structure in the sample expressions that follow.

```
struct clothes {      /* declare struct type named clothes. */
    int suit;
    char tie;
    char[10] shirt;
    f at sh es;
} summer, *ps;        /* a var. and a pointer of this type. */
```

The following expressions will fetch the tie member of the struct summer:

```
summer.tie
(*ps).tie
```

The structure pointer operator, `->`, is a convenient shorthand for a pointer to a structure member. The following expressions mean the same as the expressions in the previous example:

```
summer->tie
ps->tie
```

The syntax for the structure pointer operator is:

```
primary-expression->identifier
```

Both the structure pointer operator and the structure member operator (sometimes called dot) evaluate from left to right. They are described in detail in Chapter 8.

Increment and Decrement Expressions

You make increment and decrement expressions using the increment and decrement operators. The increment operator (`++`) adds one to the operand; the decrement operator (`--`) subtracts one from the operand.

You may use them in either prefix (`++lvalue`) or postfix (`lvalue++`) form: prefix changes the operand before using its value, and postfix changes the operand after using its value. For example, if `y` equals 10, then

```
x = y++
```

sets `x` equal to ten, then increments `y` to 11.

In this case,

`x = ++y`

`y` gets incremented to 11 first. Then the assignment is made, setting `x` equal to 11 also.

The type of the result is the same as the type of the lvalue expression.

You can only use `++` and `--` with variables, not with constants.

Arithmetic Expressions

Arithmetic expressions are formed using the arithmetic operators on the appropriate operands.

C's arithmetic operators are: `+` `-` `*` `/`, and the modulus operator, `%`. The arithmetic operators, together with their operands, are:

<code>expression + expression</code>	addition
<code>expression - expression</code>	subtraction
<code>expression * expression</code>	multiplication
<code>expression / expression</code>	division
<code>expression-1 % expression-2</code>	give the remainder of expression-1 divided by expression-2

Integer division occurs when the two operands resolve to data type `int`; C truncates the result of this division to an integer before assigning it.

The unary minus operator gives the opposite (two's complement) of its operand:

`-expression`

When chained together, as in

`a + b + c + d`

the arithmetic operators all group left-to-right:

`((a + b) + c) + d`

When such constructions occur with `*` and `+`, which are associative and commutative, C may rearrange the order of evaluation. For example, with:

`((a + b) + c)`

C may evaluate it as `(a + (b + c))`. Side effects can occur.

A Note on Side Effects in C

In chained operations, the order of evaluation does not always occur in the order that appears in the source file. For example, this kind of chained expression has many uses:

(expression-a + expression-b + expression-c + expression-d)

Suppose a program uses such an expression, suppose expression-a and expression-b both modify another variable, and suppose expression-a must always modify the variable before expression-b does. The C compiler, however, does not know this; it simply evaluates a, b, c, and d in whatever order suits it.

To avoid these unwanted side effects, check the order of evaluation. This order depends on the operator; it is noted for each operator in the appropriate section in Chapter 8.

Relational Expressions, Equality Expressions, and Some Logical Expressions

C's relational operators are:

expression < expression	less-than
expression <= expression	less-than-or-equal-to
expression > expression	greater-than
expression >= expression	greater-than-or-equal-to

C's equality operators are:

expression == expression	is equal to
expression != expression	is not equal to

C's logical AND, logical OR, and logical NOT operators are:

expression && expression	logical AND
expression expression	logical OR
!expression	logical NOT

NOTE: bitwise logical operators are listed with the other bitwise operators below.

The expressions formed here all resolve to integers, and can have the value 0, if false, or 1, if true.

The relational operators all have the same precedence, which is higher than that of the equality operators, both of which have the same precedence. The equality operators in turn have higher precedence than the logical AND and logical OR operators.

You can chain logical OR and logical AND operations together, as in:

```
a || b || c || d
```

In such a construction, the `||` operator groups left-to-right; and as soon as one operand proves true, evaluation stops. With logical AND, the operator groups left-to-right and evaluation won't proceed if any operands prove false.

Logical NOT has higher precedence than all the logical and relational operators.

Bitwise expressions

C's bitwise operators are:

<code>expression >> expression</code>	shift right
<code>expression << expression</code>	shift left
<code>~expression</code>	ones complement
<code>expression expression</code>	bitwise inclusive OR
<code>expression ^ expression</code>	bitwise exclusive OR
<code>expression & expression</code>	bitwise AND

NOTE: Don't confuse `|` and `&` with the logical connectives `&&` and `!!`.

The operands to all bitwise operators must be integers. C performs the usual arithmetic conversions with the operands.

Given the expression `shiftee << shifter`, C interprets `shiftee` as a bit pattern and shifts it to the left by `shifter` bit positions. C fills the vacated bits with zeros.

The shift operators `<<` and `>>` group left-to-right. C performs the usual arithmetic conversions with this operation. Both operands to the shift operator must be integers.

The ones complement operator `~` interprets the operand as a bit pattern. It then converts each 1 into a 0 and vice versa. C performs the usual arithmetic conversions for the `~` operator. The operand must be an integer.

The bitwise inclusive OR operator `|` is useful for turning bits on. For example

```
dalmas = dalmas | MASK
```

sets to 1 in `dalmas` the bits that are set to 1 in `MASK`.

The bitwise AND operator is useful for bit manipulation, like masking off some bits. For example,

```
d = p & 0177
```

sets to zero in d all but the low-order seven bits of p.

The & operator is associative and applies only to integral operands.

You can chain bitwise AND, bitwise inclusive OR, and bitwise exclusive OR operations together, as in:

```
a | b | c | d
```

In such constructions, the C compiler may rearrange the order of evaluation.

You may combine all the bitwise operators with the = operator to form an assignment operator, as in

```
a |= b
```

which means: set a equal to the result of "a bitwise-OR b".

Assignment Expressions

C's assignment operator, =, assigns the value of the right operand to the left:

```
lvalue = expression
```

The result of the expression is then the value of its left operand; and the type of the expression is then the same type as the type of the left operand. C performs the usual arithmetic conversions.

You can chain assignment operations together, as in:

```
this = that = theother = 0;
```

In such constructions, the assignment operator groups right-to-left, producing:

```
this = (that = (theother = 0));
```

which sets all three variables to 0.

Compound Assignment Operators

Compound assignment operators provide a shorthand notation for assigning the result of some operation to an lvalue. For example, you can write this expression

```
i = i + 1
```

using a compound assignment operator like this

```
i += 2
```

The compound assignment operators are:

```
+= -= *= /= %= <<= >>=
```

You'll find more information under each individual operator (i.e., under + - * / % << and >>) in Chapter 8.

The Conditional Operator

The conditional operator performs an if-then-else using its three operands:

```
expression ? expression : expression
```

The order of evaluation is as follows: C evaluates the first expression. If it is nonzero, C evaluates the second expression. But if the first expression is zero, C evaluates the third expression. Thus, C evaluates either the second or the third expression, but not both.

The conditional operator provides an alternative to the if-else construction. Instead of writing

```
if expression-1
    expression-2
else
    expression-3
```

you can write

```
expression-1 ? expression-2 : expression-3
```

You can nest conditional operations, and you can chain them. When chained, C groups the conditional operations right-to-left.

Punctuation for Expressions: the Comma

Anywhere an expression can occur, an expression list can occur:

(expression, expression, expression)

And in a list of expressions, you use the the comma operator to produce left-to-right evaluation in the list. In such a list of expressions, all but the last value are discarded. For example, in the following group of expressions:

(i++, j += 3, k >= 98.6)

the final value of the parenthesized expression will be either 1 or 0, since the final expression is `k >= 98.6`, and this can have only two outcomes: true (1) or false (0).

Since only the result of the rightmost operand is saved, it follows that the type and value of the result are the type and value of the rightmost operand.

The comma is useful in for statements. There it produces left-to-right evaluation when more than one expression appears in one of the three segments of the for's control statement.

End of Chapter

CHAPTER 5

STATEMENTS

This Chapter discusses the kinds of statements you can create in the C language.

Statements are the next level up from expressions: you use tokens to make expressions, and out of expressions you make statements. As we said, you can group statements together into blocks of code. These blocks are just several statements grouped together with braces.

Blocks of code are useful for organizing your program; they're necessary in C, particularly when you create functions. The "Compound Statements" section below shows how to make and use compound statements. The next chapter talks about how to use compound statements, how to take advantage of C's block structure, and how to use blocks of code to create functions.

Simple Statements

=====

A C statement is an executable line of C code. It can be a single line line with a semicolon at the end, as in

```
pile[i] = true;
```

or several such lines enclosed in braces, as in

```
{
    pile[i] = true;
    j += newlimit;
    for (; j <= TOPHEAP; j++)
        r += pile[j];
}
```

As we said, there are really two levels of statements in C: simple and compound. The single-line statements we'll call simple statements, and the statement formed by the braces is called a compound statement.

For convenience, we'll divide simple statements into five functional categories:

expression statements
conditionals
loops
statements to get out of loops
flow-control statements

The next sections describe these categories.

Expression Statements

Any expression in C can also be a statement; to make it so, put a semicolon at the end of it. In fact, most statements in your C programs will probably be expression statements. Some examples:

Expression	Statement
-----	-----
a++	a++;
a == b	a == b;
a += b	a += b;
c = (count > LIMIT)	c = (count > LIMIT);

However, not all expression statements do something useful:

a++;

increments a, but

(a == b);

while being a legitimate statement with a result (1 if true, 0 if false), does nothing with its result.

Conditional Statements

In C there are four different ways to make conditional statements, those that perform a test and branch on the result:

- * if and else statements,
- * switch and case statements,
- * the ?: (conditional) operator, and
- * the && (logical AND) and || (logical OR) operators.

?: , && , and || are described in Chapter 4 and in detail in Chapter 8. The following sections describe if-else and switch-case statements.

if-else

The if statement is for making tests and decisions; it works much like if-else combinations in other languages.

In C, the if statement tests the numeric value of the parenthesized expression that follows it. If the expression is true (not equal to zero), the program then executes the following statement. If the parenthesized expression is false (equal to zero), the program proceeds without executing the substatement.

Use the if statement by itself like this:

```
if (expression)
    statement
```

The else statement is for use with the if statement. If the expression following an if statement proves false, and there is an else statement, the program skips over the if and its statement(s) and goes to the else.

The else keyword means: if (expression) is false (equal to zero), execute the statement that follows the else. Use the if and else statements together like this:

```
if (expression)
    statement-1
else
    statement-2;
```

The else-if construction is useful for offering more than one alternative to the if:

```
if (expression-1)
    statement-1
else if (expression-2)
    statement-2
else
    statement-3
```

In the above example, if expression-1 is true, the program executes statement-1; if expression-1 is false but expression-2 is true, the program executes statement-2; and if both expression-1 and expression-2 are false, the program executes statement-3.

Your program can get cumbersome when you nest if-elses. The switch statement can tidy up such nests; switch is described in the next section.

CAUTION: Nested if and else statements can produce apparent ambiguities; to avoid them, use braces, as in this example:

```
if (expression-1)
{
    if (expression-2)
        statement-1
}
else
    statement-2
```

The brackets are needed to show which if the else belongs to; the else belongs to the outer if statement. If the brackets weren't there, the C compiler will automatically associate the else with the last else-less if it encountered; the inner if statement. In other words, without the braces, statement-2 would be executed as a result of expression-2, not expression-1.

switch and case

Unlike the two-way branching offered by if-else, `?:`, `&&`, and `||`, the switch statement offers multi-way branching. You can use a switch statement and a series of case statements to replace a series of if...else if...else if...else statements.

The statement that follows switch is typically a compound statement. It can include one or more case statements, as shown in the example below. A simple statement would permit only one case and therefore doesn't make sense; and the C compiler gives a warning message if the statement following switch is not compound.

The following example shows a complete switch statement that uses several case statements and a default statement. Depending on the value of `c`, the program will select the case whose constant matches the value of `c`.

```
switch(c)
{
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
        counta++;           /* c == 0 through 9; increment      */
        break;              /* counta, then leave switch.    */
    case ' ':
    case '\t':
    case '\n':
        countb++;           /* c == space, tab, or NEW LINE; */
        break;              /* increment countb, leave switch.*/
    default:
        countc++;           /* c == none of the above; increment */
        break;              /* countc, then leave switch.    */
}
```

This example increments `counta` when `c` is 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9; and it increments `countb` when `c` is a space, tab, or NEW LINE. If `c` has none of those values, the example increments `countc`.

The expression that follows the switch statement must resolve to a constant.

- * If this constant matches one of the case constants, the program proceeds to the statement that follows the matched case prefix.
- * If no case constant matches the constant-expression, and if there is a default prefix, the program proceeds to the statement following the default prefix. In our example, that statement increments an alternate counter, counta.

Default makes it unnecessary to represent every possible case.

- * If no case matches and there is no default, then the program leaves the switch without executing any of the statements.

Case and default prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. The result is that you can several cases that lead to the same statement, as this example illustrates,

To avoid continuing through each case exit instead from the switch, use a break statement. The break statement allows the program to leave the switch without testing any more cases. In the example above, if there were no break statements, the program would have tested each case and performed all the statements, resulting in countd being incremented for every case.

Loops

You use loops to let the program execute a section of code repeatedly. C offers several kinds of loops: while, do-while, and for. Of course, it's possible to construct a loop with a label and a goto, but compared to the alternatives, the goto construction is hazardous and usually unnecessary.

The main differences among C's looping statements are in the point at which each loop tests for its exit condition.

while

In C, while loops are set up as follows:

```
while (expression)
    statement
```

As long as expression is true (not equal to zero), the program will execute the following statement. Since the expression is evaluated first, it's possible that the loop will not be executed at all; if the expression is false, the program will skip the statement altogether.

The following example copies characters from the standard input to the standard output until a NEW LINE or EOF arrives. It does not copy the NEW LINE:

```
while ((c = getchar() != EOF && (c != \n)
    putchar();
```

The alternative to while is do-while, which performs a loop and makes the termination test at the end of the loop.

When while is set up like this:

```
expression-1;
while (expression-2)
{
    statements
    expression-3
}
```

it can often be replaced with a for statement. For loops offer the advantage of grouping the loop control code -- expression-1, expression-2, and expression-3 -- together in one place.

```
for
--
```

The construction

```
for (expression-1; expression-2; expression-3) statement
```

behaves as if:

```
expression-1;
while (expression-2)
{
    statement
    expression-3
}
```

Like the while statement, for will keep executing the substatement as long as expression-2 is true (not equal to zero).

The syntax of for reflects a common format for loops:

- * expression-1 usually specifies initialization for the loop;
- * expression-2 specifies a test, such that the loop is exited when the test fails, succeeds, equals 0, etc.;
- * expression-3 often specifies an incrementation that is performed after each iteration.

expression-1 and expression-3 are often assignments or function calls and expression-2 is a comparison or test.

expression-1, expression-2, and expression-3 are all optional; the semicolons that separate them or not. The following is a legal for statement:

```
for (;;)

```

The for statement is useful for keeping loop control statements together:

```
for (i = 0; i < N; i++)
    peel mango[i];

```

This is C for: process (peel) the first N elements of the array, mango.

Your program can alter the limits of a for loop from within the loop. The controlling variable i retains its value when the loop terminates for any reason.

```
for (i=0; i < N; i++)
{
    crunch c[i];
    if c[i] == 10
        i=N;
}

```

Using a single for statement, you can have the program skip all white space (spaces, NEW LINES, and tabs):

```
for (i=0; s[i]!=' ' || s[i]!='\n' || s[i]!='\t'; i++)
;

```

This refrains from touching successive locations in the array s if they happen to contain a space (' '), NEW LINE ('\n') or tab ('\t').

```
do
--

```

The do statement performs the loop termination test at the end of each loop iteration instead of at the beginning. The syntax for do is:

```
do statement while ( expression ) ;

```

The test for the do loop takes place after the statement executes, so that the statement is always executed at least once. The loop will then continue as long as the expression proves true (not equal to zero).

Getting Out of Loops

In C, there are several statements that let a program leave a loop without passing the loop's exit tests or reaching the end of the loop's code. They are: break, continue, and goto.

break

break is one of two statements that force the program to leave a loop or a block of code. break gives the program an early exit from the smallest enclosing while, do, for, or switch statement. When leaving a loop, the program proceeds to the statement following the loop-terminating statement.

The syntax for break is always the same:

```
break;
```

The break statement is the most common way to leave a switch. Without the break statement, the switch would simply fall through to the next case, rather than leave the switch statement entirely.

By contrast, the continue statement sends the program to the beginning of the next iteration of the loop.

continue

A continue statement causes the program to immediately begin the next iteration of the smallest enclosing loop. continue is for use with while, for, and do.

- * In while and do loops, continue causes immediate execution of the first statement in the loop.
- * In for loops, continue causes control to pass to the reinitialization step.

continue won't stop at a switch, though: a continue inside a switch inside a loop goes up to the beginning of the loop.

By contrast, the break statement causes the program to leave the loop entirely. In practice, continue tends to be used less often than break.

The syntax for continue is always the same:

```
continue;
```

goto and Labels

goto is a basic flow-control statement. goto is for use with labels only. The purpose of a label is simply to tag a statement so that it can be found by a goto. The syntax for goto is always:

```
goto label;
```

The syntax for a label is always:

```
label : statement
```

where label is a legal C identifier.

A label is simply an identifier followed by a colon followed by a statement. You can attach a label to any statement, and a statement can have more than one label:

```
label1:  
label2:  
label3: statement
```

The case and default prefixes are both labels; however, you cannot jump to them directly with a goto statement.

The scope of a label is the function in which it is defined, regardless of whether the label was defined within a local scope; i.e., an enclosing compound statement. This means that a label can be reached only by gotos that are also inside the same function.

Although unfashionable nowadays, goto can still prove useful when a program needs to stop processing inside some deeply nested code.

Flow of Control

In C, there are several ways the program can proceed to another statement.

1. The simplest way is to execute the statement that follows the current statement. This is the default in C.
2. Decisions and Branching. In C, the conditional statement and the loops all offer ways to perform a test and then, according to the results, choose the next statement to execute.
3. Reiteration, that is, executing the same piece of code again, takes place with the loop statements: while, for, and do-while. All offer ways to select the same code again, or proceed to new code.

4. Jumping to another section of the program takes place when the program calls a function. Similarly, execution will return to the jumping off place as the result of a return statement in a called function, or as the result of the program reaching the final right brace of a called function.
5. break and continue offer alternate ways to leave loops, functions, and other blocks of code.
6. Some C operators also can determine the flow of control inside expressions. For more information on expressions, see Chapter 4.

```
(( expression-1 ) && ( expression-2 ) && ( expression-2 ))
```

If expression-1 proves false, the AND has failed and C does not evaluate expression-2 or expression-3.

Compound Statements

One statement in particular, the compound statement, allows you to group other statements together into a block of code. Compound statements are simply one or more statements inside a pair of braces ({ and }). In fact, anywhere you can put a statement in a C program, you can put a compound statement.

There is really only one compound statement:

```
{
    declaration(s)
    statement(s)
}
```

Compound statements and blocks of code are described in more detail in Chapter 6.

The Null Statement

Occasionally, a null statement is useful. In this for loop, the last statement in the for's header expression is null:

```
for (i=0; j <= 255; )
    if name[i] < 255
    {
        i = name[i];
        ...
    }
```

We might expect the last statement in the header to be `i++`. But the body of the loop adjusts `i` as it goes, and the header doesn't need to perform any further adjustments.

End of Chapter

CHAPTER 6

BLOCK STRUCTURE and FUNCTIONS

In the last chapter we described how to use the braces, { and }, to create a compound statement. Each statement inside the compound statement must have an appropriate terminator (either a semicolon or another brace). You can nest compound statements, as in

```
{
    declaration(s)
    statements
    ...
    {
        declaration(s)
        statements
        ...
    }
    statements
    ...
}
```

This chapter contains more information about compound statements. It also explains how to create and use functions. The last section explains how to call functions from a C program that are written in other languages.

Blocks of C Code

=====

The compound statements and functions of C allow you to write modular, block-structured code.

We've already seen how to use the compound statement under a while, for, if, or else statement to form a block of code, as in

```
for (i=0; ((c=getchar()) != EOF; i++)
{
    /* block begins here.  */
    ...
    statements
    ...
}
/* block ends here.  */
```

This block is strictly for use by the preceding for statement. In C you can also create blocks that are for use by other parts of the program as well; these are subroutines or procedures. In C, they're called functions.

The C compiler sees every C program as simply a collection of functions: the main function, and subsidiary functions. The code for one function can't appear inside another, but of course functions can call each other ad infinitum.

Your program can obtain other functions from the runtime library; the Data General C compiler comes with its own runtime library. Your program can also use other functions of its own or from other programs, either written in C or in other languages.

Here's an example of a C program with several functions:

```
main()
declarations
{
    declarations
    statements
    ...
}

function1()
declarations
{
    declarations
    statements
    ...
}

function2()
declarations
{
    declarations
    statements
    ...
}
```

How to Create a Block of C Code

=====

To delimit a block of C code, you use braces around a group of statements:

```
if (y < apogee)
{
    declarations    /* this block belongs to the if statement. */
    statements
    ...
}
```

When the C compiler sees these braces, it separates the block from the rest of the program, as follows:

- * Variables and constants declared inside the block have scope only inside the block, unless declared extern.
- * Variables and functions outside the block have scope over the variables and constants, providing those variables and constants are not redeclared inside the block.

How to Create a Function

=====

Functions are useful for allowing various parts of a program to share sections of code. To create a function in C, you must give a function name, a list of arguments to the function, and a body of code that the function will execute.

Functions in C use this syntax:

```
function-name (argument-list)
declarations of argument-list
{
    declarations of local-identifiers
    statements
}
```

where: argument-list is a list of formal arguments to this function;

declarations of argument-list are the declarations for all the formal arguments; and

declarations of local-identifiers are the declarations of identifiers this function will use, including variables, constants, and functions it will call.

You can't nest functions; in other words, you can't define the code for one function inside another. However, you can call one function from another, and often you must declare the called function before you can call it. The next section describes how to declare functions in C.

How to declare functions

By default, whenever the C compiler encounters the statements that make up a new function (as when it sees `function1` in the example above), it takes those statements to be the declaration of the function. By default, the compiler assumes the function returns data that is type `int`.

When one function calls another, the caller must declare the called function along with the rest of the variables and constants the caller will use. The caller doesn't list the called function's code, just the called function's name and type. In the following example, `function1` declares its own formal parameters, `a` and `b`, its own local variables, `c` and `d`, and a function it will call, `cplus`:

```
function1(a, b)    /* declare function1.          */
char a, b[];      /* declare formal parameters a and b.  */
{
    int c, d;      /* declare local variables c and d.      */
    float cplus(); /* declare function to be called: cplus.*/
    statements
    ...
}
```

In this case, the parentheses after `cplus` tell us it is a function. The C compiler will expect to find the actual code for the `cplus` function later in the program:

```
float cplus(e, f, g) /* declare cplus.          */
int e, f;           /* declare formal params.   */
double g;
{
    declarations    /* declare local indentifiers. */
    statements
    ...
}
```

In fact, a function may return some data, it may return a structure, or it may return no data at all. If a function will return some data, you must declare the function as such; in other words, you must declare the function in terms of the kind of data it returns. If a function will not return any data, you may declare that function to be `void`. `void` tells the C compiler that the function doesn't return anything, and not to expect any results.

Data Type

When you declare a function, you declare its data type. Actually, you declare the type of data the function returns.

The choices for functions are nearly the same as for data: they include

- * the fundamental types char, int, float, and double,
- * the adjectives short, long and unsigned,
- * the derived types struct and union,
- * any types you may have defined, and
- * any synonyms for types that you have created with the typedef keyword.

In C you can't return an entire array or an entire function from a function. But you can return a pointer to either an array or a function.

Here are some examples of functions.

Example 1. To declare a function that returns a double:

```
double function-name()
{
    statements
}
```

The parentheses show that this is a function, not just another variable, and that the statements inside the following braces are its code.

Example 2. A function returns a structure called daylist. First, we need a declaration of this structure's type including a name for the type:

```
struct michele          /* declares the struct type michele */
{
    char don[20];
    int herbs;
    int tea;
    int tofu;
    char cat;
}
```

We can make a declaration of the function and any variables that are of the same type. We'll call the function days:

```
struct michele days(), daylist;
```

This declares a function named days and a variable named daylist; days returns a struct of type michele and daylist is a struct of type michele.

And finally, we need to write the code for the function days:

```
struct michele days (food) /* beginning of code for days */
int food;                  /* declares the formal parameter */
{
    statements              /* code for days */
    ...
}
```

And here's how the program might call function1:

```
int rice = 3;
daylist = days(rice);
```

Storage Class

Unless you declare them otherwise, the C compiler declares all functions to be storage class extern. If you don't want the function to be extern, i.e., if you don't want it to be visible to other functions outside the current compile-file, declare the function to be static.

```
static int thooft(particle)
int particle;
{
    ...
}
```

This function is called thooft, accepts an argument called particle, and returns an integer.

The main() Function

=====

There's always at least one function in every C program: the main() function.

The purpose of main() is simply to provide a place for the C program to begin executing.

Two standard formal arguments in C, argc and argv, are for use in the main() function, and not in other functions. When you execute a C program, you can use the argc and argv parameters to get arguments from the command line. Suppose we type this command line:

```
) XEQ MYPROGRAM FILE1 FILE2
```

argc -- "argument count" -- will receive an integer representing the number of arguments that arrived; in this case there are three, including the name of the program: MYPROGRAM, FILE1, and FILE2. But since argc starts counting at 0, the value of argc will be 2.

argv is an array of pointers; each pointer points to a character string for the corresponding argument. C terminates the strings with NULLs. Remember that since argc and argv are formal parameters, you must declare them as such, as in this sample from the main function of MYPROGRAM:

```
main(argc, argv)      /* declares main routine for MYPROGRAM. */
int argc;             /* declares argc: argument count.      */
char *argv[];         /* declares argv: strings to hold args. */
{
    ...
}
```

MYPROGRAM can then sift through the array argv, read each argument, and do what it is supposed to do.

The following sample program prints out the arguments it receives:

```
main(argc, argv)
int argc;
char *argv[];
{
    int i;
    for (i=0; i < argc; i++)
        printf ("%s\n", argv[i]); /* print the string at argv[i] */
}
```

How to Call Functions

=====

This section explains how one function in a C program calls another. The next section explains how a function in a C program can call a function that is written in another language.

To call a function, a C program simply give the function's name and supplies the arguments that the function expects:

```
putchar(ch);          /* write a byte to the standard output*/

idcard(name, address); /* calls idcard, passes it two args:  */
                      /* name and address.                */
```

NOTE: Some functions, like getchar() and putchar(), are really in-line macro expansions; that is, they expand to more C code. No function call actually takes place.

For more on C macros, see Chapter 9; for more on putchar() and getchar(), see Chapter 11.

How to Pass Data To and From Functions

When your program calls a function and passes some arguments to the function, C passes copies of the arguments. The calling function cannot modify the original arguments; even if it tries, the changes will not affect the arguments passed. In other words, C permits call-by-value only.

If you need to have the functions modify the arguments directly, you can use the following methods:

1. The called function can return a value; so the calling function simply needs to copy the returned value from a temporary object back into the original object.
2. The calling function and the called function can operate on common variables. To create a common variable, define it outside both functions and declare it either static (if the functions are all in the current compile-file) or extern.
3. The calling function can pass pointers to the called function. This lets the called function follow the pointer(s) and modify the data directly; and since the actual pointer that is passed is not modified, this method doesn't violate call-by-value.

The return Statement

To return data to the caller, C functions use the return statement. return is useful for sending values back to callers, and that is its sole use. These examples show the syntax for return:

```
return;                /* returned value is not defined.      */

return expression;     /* expression is resolved, converted to */
                      /* type of the function, then returned. */
```

Functions of type void cannot use the second method, since they are not allowed to return anything.

Otherwise, to get back to the caller, execution has to "fall off the end" of the function by reaching the closing right brace. When this occurs, the returned value is not defined.

Either way, with or without a return statement, execution will return to the statement following the statement that called the function.

Any expression can follow return. Before the expression goes back to the caller, C will resolve it into a single value. Such an expression must be the same type as the function itself; if not, C will convert it to the function's type if possible, following the C rules for conversion by assignment.

The calling function is free to ignore the returned value. Also, a function need not return a value;

```
return;
```

causes control to go back to the caller, but no useful value is returned.

Functions of type void never return anything.

You may put as many return statements as you wish inside a function.

Recursive Calling

Functions can call themselves. These are recursive functions and they are especially useful for searching recursive data structures like binary trees.

Data that is declared auto or register in a recursive function will get a new allocation of storage in the next execution of the function. Data that is declared static or extern will have the same values in the next execution of the function.

The following routine searches a sorted binary tree for an integer value called sought. If it finds sought, it returns the pointer to the node where sought is; if it doesn't find sought, it returns zero.

```
struct tnode                                /* declare data */
{                                           /* type for nodes*/
    struct tnode *left;
    struct tnode *right;
    int value;
};

struct tnode *find (nodep, sought)
struct tnode *nodep;
int sought;
{
    if (nodep == (struct tnode*)0)
        return (nodep);                    /* return NULL */

    else if (nodep->value < sought)          /* too low */
        return (find (nodep->left, sought)); /* recurse left */

    else if (nodep->value > sought)          /* too high */
        return (find (nodep->right, sought)); /* recurse right */

    else
        return (nodep);                    /* found it */
                                           /* tell where */
                                           /* end of find() */
}
```

How to Call Functions Written in Other Languages

=====

This section describes how your C program can call functions that are outside the current compile-file and are written in other languages. It also explains how programs written in other languages can make calls to programs written in C.

For your convenience, the C compiler comes with a directory of program examples; the examples include a PL/I program calling a C program, and C programs calling PL/I, Pascal, and FORTRAN 77 programs. Please see the C release notice for complete information on these examples.

For such calls to take place, you must do three things:

1. Make sure both routines use the same calling sequence, particularly if they pass arguments back and forth.
2. Make sure the arguments that are passed, if any, have corresponding or at least harmonious data types.
3. Later, link all the routines together.

The following sections explain how to set up C functions so that they have the correct calling sequences, and how to set up the arguments that these functions pass so that they have the correct data types.

When you link programs of different languages together, you must include the libraries for both languages. For more information about linking C programs, see Chapter 11.

Calling Sequences

When one function written in C calls another function written in C, the C compiler automatically takes care of the calling sequence between the two functions. The previous section contains information on this kind of calling.

When one function written in C calls another function written in another language, or vice versa, you, the programmer, may have to take care of the calling sequence between the functions. Before we explain how you do this, it's useful to look more closely at the calling sequences that are available.

- * Virtual storage external calling sequence (VS/ECS). VS/ECS is a call-by-reference calling sequence, meaning that pointers to the arguments, not the actual arguments, are passed between functions. The functions can then manipulate the arguments. The AOS/VS 32-bit languages FORTRAN 77, PL/I, Pascal, and COBOL use VS/ECS.

To call programs written in these languages from a C program, your C program will need to use their calling sequence.

- * **C Calling Sequence.** C's calling sequence is similar to VS/ECS except that C is a call-by-value language, meaning that the called routines are not free to alter the arguments directly.

Programs written in other languages that wish to call C programs may wish to comply with C's calling sequence, or both programs may wish to resort to VS/ECS.

- * **Assembly Language Calling Sequence.** There is also a special sequence for calling assembly language modules from C functions, described below.

To make sure your C function uses the correct calling sequence, you must either

1. Tell the C compiler that your C function uses the VS/ECS convention by declaring the function with the appropriate other-language keyword: `$fortran`, `$pl1`, or `$pascal`; or
2. Tell the C compiler that your C function uses the assembly language calling sequence by declaring the function with the `$asm` keyword; or
3. Have the C function pass the necessary pointer(s) to the arguments, and make sure the arguments are of data types that will harmonize with the other language's data types.

NOTE: If you do NOT specify a function as either `$asm`, `$fortran`, `$pl1`, or `$pascal`, C assumes the function use the C calling sequence, which is the default.

The next sections describe the three calling sequences in more detail. Following them, the last section discusses declaration for other-language calling.

VS/ECS Calling Sequence

The AOS/VS 32-bit languages COBOL, FORTRAN 77, PASCAL, and PL/I use, by default, a calling sequence called virtual storage external calling sequence, or VS/ECS. These languages are sometimes called common languages since they use a common code generator.

If you declare a C function using one of the following keywords:

```
$fortran
$pascal
$pl1
```

the C compiler will cause that function to use VS/ECS calling sequence.

In that case,

- * You, the programmer, must make sure that the data types of any arguments that the function passes harmonize with the data types of the other language, and
- * Other functions in other languages should use VS/ECS in order to call this function.

The C compiler will take care of the rest of the details of the calling sequence.

For example, this C function, named local, will be able to call another function, named remote(), that is written in FORTRAN 77:

```
int local(a, b, c)
int a, b, c;          /* declare formal parameters.    */
{
    $fortran int remote(); /* declare remote: to be called. */
    ...
    remote(d);          /* call remote.                */
    ...
}
```

The following sections describe VS/ECS in even more detail.

Making the Call

As we said, VS/ECS is a call-by-reference calling sequence. When a function issues a call using VS/ECS, that function must push pointers to the arguments, not the arguments themselves, onto a stack.

- * Before pushing a pointer, the program
 1. Calculates any argument that is an expression or any by-value argument (one that must not be altered by the callee),
 2. Stores the result in a temporary buffer, and
 3. Determines the address of the temporary buffer.
 4. Then the program uses this address as a pointer and pushes it onto the argument stack.

In VS/ECS, the pointers to the arguments end up on the stack in the reverse order of their appearance in the argument list from the calling function.

* The alignment of these pointers follows these rules:

~ COBOL always sends out and must receive back byte pointers.

~ Unaligned bit values cause the program to push the word address of a four-word (64 bits) bit pointer that points to the actual value. The bit pointer itself has the following format:

* first double word contains the word address of the word where the bit occurs;

* second double word contains the offset into the word, in bits, of that bit.

PL/I structures and Pascal packed records consisting only of bit-aligned structures cause the program to pass a bit pointer.

~ Unaligned character values, except PL/I's CHARACTER VARYING, cause the program to push a byte pointer to the first character.

PL/I structures and Pascal records consisting of only character- or bit-aligned values cause the program to push a byte address.

~ All other arguments, including ALIGNED BIT and ALIGNED CHARACTER in PL/I, push a word pointer to the first word the argument requires.

Once the called routine receives the pointers from the calling routine, the called routine is free to use and modify the arguments themselves.

When the Call Returns

In VS/ECS, programs handle returned values as follows:

* Integers, pointers, and small bit fields (32 bits or less) return their values in AC0.

* Floating-point numbers return their values in FAC0.

* All other values expect the caller, before making the call, to

1. Allocate a buffer large enough to hold the return value, and
2. Place a word pointer to this buffer in AC2.

- * The code generated by the common language compilers expects to keep values in accumulators AC0, AC1, and AC2 across the call, providing the called functions don't use these accumulators for returned values.
- * The code generator considers values in accumulators FAC0, FAC1, FAC2, and FAC3 to be destroyed by the called function.

C Calling Sequence

If you do NOT specify a C function as either \$asm, \$fortran, \$pl1, or \$pascal, C assumes the function uses the standard C calling sequence, which is the default, whenever it calls or is called by another function. Naturally, C routines use this calling sequence to call each other.

Other languages that wish to call C routines may be limited by the calling sequences they can use. In that case, the C routines will have to comply with their calling sequence.

When the C compiler sets up the calling sequence for one C function to call another, it does so as described in the next two sections.

Making the Call

The calling sequence in C is similar to VS/ECS except that VS/ECS always pushes pointers to the arguments onto the argument stack; C usually pushes copies of the arguments onto the argument stack.

If the argument is, in fact, the address of something (as in &object), C assumes you want true call-by-reference and passes the appropriate pointer.

- * Before pushing the arguments, C performs the following:

Data Type	Action
structs, unions	go into temporary buffers; C then passes a single argument in their place, a word pointer to the buffer.
floats	converted to double
chars, unsigned chars, short ints	converted to ints
short unsigned values	converted to unsigneds

- * The arguments end up on the stack in reverse of the order they appeared in the argument list from the calling function.
- * The C compiler uses the XCALL or LCALL Macroassembler instruction to call all functions; VS/ECS uses these instruction to call functions, too. These instructions have an argument count field, and they expect this field to contain the size, in number of double words, of the argument list stack.

When the Call Returns

When handling return values, C programs follow these rules.

- * The C function's type causes the following results:

<u>Type of C Function</u>	<u>Result</u>
void	does not return anything to the calling function.
char, unsigned char, short int, short unsigned, int, unsigned, long int, long unsigned	result appears in ACO
pointer to any data type	result appears in ACO
float, double	result appears in FACO
struct, union	result appears in a buffer; calling function must pass a pointer in AC2 to that buffer; buffer must be large enough to hold result.

NOTE: char and unsigned char values passed using C calling sequence are not compatible with char values passed using VS/ECS.

- * AC3 is expected to contain the frame pointer after the callee returns.

The easiest way to cause the frame pointer to appear in AC3 is to have the callee issue a WSAVS or WSAVR AOS/VS Macroassembler instruction at the beginning of the function, and a WRTN instruction to return.

- * To return a value in ACO, store the return result in the place in the return block that holds ACO (FACO in CSYM.SR).

Assembly Language Calling Sequence

C has a special sequence to call routines that are written in AOS/VS Macroassembler language.

NOTE: You don't have to use this special sequence. The C function can use the normal C calling sequence, or you can declare the function with \$asm, which will cause it to use VS/ECS.

Similarly, assembly language routines can call C by using either the C calling sequence or VS/ECS.

CAUTION: You may not declare any C function to be \$asm.

- * If the called function is of type void, the caller expects the callee to return to the instruction following the call.
- * Otherwise, the called function must be of an integer type, and from such functions the return value is
 - 1 if the callee returned to the instruction following the call, and
 - 0 if the callee skipped the first 16-bit word after the call.
- * C does the call via a LJSR Macroassembler instruction, and the code generator assumes that
 - 1. ACO, AC1, AC2, and all the floating-point accumulators contain garbage.
 - 2. AC3 contains the frame pointer.
- * Using this sequence, a C function can pass up to three integral arguments (passed in ACO, AC1, and AC2) to a Macroassembler routine.

Declarations for Other-Language Calling

To identify an other-language routine to the C compiler, you must declare that routine among the variables and functions of the calling C function. To do so, use one of the other-language specifiers:

\$fortran	for routines written in FORTRAN 77
\$pl1	for routines written in PL/I
\$pascal	for routines written in Pascal
\$asm	for routines written in AOS/VS Macroassembler language

Naturally, you must also declare the C function that calls the other-language routine. For example, the function caller, below, calls a FORTRAN 77 routine named callee:

```
char caller(a, b)
int a, b;
$fortran callee();    /* declare the other-lang. func. */
{
    ...
    callee();          /* call the other-lang. func.    */
}
```

The callee routine is compiled in a separate file and later linked in with caller.

Finally, for C functions that will be called by other-language functions, you include an other-language keyword when you declare the C function. For example, `pensee` is a function that expects to be called by a routine written in Pascal:

```
$pascal pensee()
{
    declarations
    statements
}
```

For the C function that makes a call to a routine written in another language, you will have to declare the storage class and data type of the function and of its formal arguments.

Storage Class

The storage class of any C function that calls an other-language function must be `extern`.

Data Types

The following table compares C's data types with the data types of FORTRAN 77, Pascal, and PL/I.

N/A means the language does not support that data type.

For an unsigned data type in C, you might use a corresponding integer type in the other language; but be aware that the rules for arithmetic results might be different.

In Pascal, see also the switches on the Pascal compiler: `/INTEGER=SHORT` and `/INTEGER=LONG`.

Table 6-1. Comparison of Data Types in
C, FORTRAN 77, Pascal, and PL/I

C data type	FORTRAN 77	PASCAL	PL/I
char	CHARACTER*1	CHAR	CHAR(1)
unsigned char	CHARACTER*1	CHAR	CHAR(1)
short int	INTEGER*2	SHORT_INTEGER, INTEGER	FIXED BINARY(15)
short unsigned int	N/A	N/A	ALIGNED BIT(16)
int	INTEGER, INTEGER*4	LONG_INTEGER, INTEGER	FIXED BINARY(31)
unsigned int	N/A	N/A	ALIGNED BIT(32)
long int	INTEGER, INTEGER*4	LONG_INTEGER, INTEGER	FIXED BINARY(31)
long unsigned int	N/A	N/A	ALIGNED BIT(32)
float	REAL, REAL*4	REAL	FLOAT BINARY(21)
double	DOUBLE, REAL*8	DOUBLE_REAL	FLOAT BINARY(53)
struct	N/A	RECORD (see NOTE)	STRUCTURE (see NOTE)
union	N/A	RECORD with CASE (see NOTE)	UNION (see NOTE)
(pointer)	N/A	POINTER	POINTER
NOTE: Remember that C cannot deal with bit- or byte-aligned structures.			

As mentioned above, the C compiler comes with a directory of programming examples; please see the C Release Notice for complete information on this directory.

End of Chapter

CHAPTER 7

DICTIONARY OF C KEYWORDS

This chapter describes all the C keywords alphabetically. The next chapter describes all the C operators. Both chapters are designed for quick and easy reference; together, they form a complete dictionary of the C language.

The C keywords are:

auto	else	int	switch
break	entry	long	union
case	enum	register	unsigned
char	extern	return	void
continue	float	short	while
default	for	sizeof	
do	goto	static	
double	if	struct	

Keywords added by the Data General C Compiler are:

\$asm
\$builtin
\$fortran
\$pascal
\$pl1

\$builtin is described in this chapter; \$asm, \$fortran, \$pascal, and \$pl1 are described in Chapter 6, in the section "How to Call Functions Written in Other Languages."

\$builtin

keyword

Declares a \$builtin C library function.

Syntax

\$builtin declarator-list;

Description

The \$builtin keyword is a feature of the Data General C compiler. It lets you to declare certain functions in the C runtime library so that your program can use them. \$builtin uses the same syntax as the storage class specifiers. For more information on the \$builtin functions, see Chapter 10.

Example

To make a system call directly from your C program, you need to declare the sys() routine, as follows:

\$builtin sys();

auto keyword

This object has storage class auto.

Syntax

type-specifier auto declarator-list;

Description

auto is one of the four storage class specifiers: auto, static, register, and extern.

auto stands for automatic. An automatic variable is one that exists only while its enclosing function is being executed. When the program exits that function, all its auto variables cease to exist.

auto is useful for local variables that don't need to be saved from one invocation of a function to the next.

The function must explicitly initialize its auto variables; otherwise, they will have undefined values.

By contrast, the static storage class lets a function keep a variable around between executions of that function.

The auto keyword isn't necessary in most declarations; C makes all variables auto by default, except those variables that lie outside all the functions, which C makes extern by default.

Example

Both these statements declare an integer called win with storage class auto:

win;

auto win;

Related Topics

See also: static, extern, register.

break keyword

Exit immediately, i.e., from a switch statement or from a while
or for loop.

Syntax

```
break;
```

Description

break is one of two statements that force the program to leave a loop or a block of code. break gives the program an early exit from the smallest enclosing while, do, for, or switch statement. When leaving a loop, the program proceeds to the statement following the loop-terminating statement.

By contrast, the continue statement sends the program to the end of the current iteration of the loop.

Along with return, break is the most common way to leave a switch. Without the break statement, the switch would simply fall through to the next case, rather than leaving the switch statement entirely.

break can also make code less confusing, as shown in the example below.

Example

This example shows how the break statement can provide an exit from a for loop that does not specify an exit condition.

```
#include <stdio.h>

main()
{
    char line[80];
    for (;;)                /* null statements here. */
    {
        printf ("Type a line: ");
        if (fgets (line, 80, stdin) == NULL)
            break;          /* leave the for loop. */
        printf ("You typed %s\n", line);
    }
}
```

case

keyword

One choice for the preceding switch statement.

Syntax

case constant-expression: statement

Description

case is a prefix. case is for use with switch, and it indicates one code path the switch can take. case is useful for selecting one code path from a large choice of paths; i.e., it, along with switch, is an efficient substitute for a long sequence of if-elses.

When your program executes the switch statement, it evaluates the constant-expression and then compares it with each case constant.

- * If the constants match, the program proceeds to the statement that follows the matched case.
- * If no case constant matches the constant-expression, and if there is a default prefix, the program proceeds to the statement following the default prefix.
- * If no case matches and there is no default, then the program leaves the switch without executing any of the statements.

To create a default case, use the keyword default:

default: statement

case and default prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. cases and default can occur in any order. To exit from a switch, use a break statement.

case values must be constants, and of type int.

No two case constants in the same switch may have the same value.

Example

The following example shows a complete switch statement that uses several case statements and a default statement. Depending on the value of `c`, the program will select the case whose constant matches the value of `c`.

```
switch(c)
{
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
        counta++;           /* c == 0 through 9; increment */
        break;             /* counta, then leave switch. */
    case ' ':
    case '\t':
    case '\n':
        countb++;          /* c == space, tab, or NEW LINE; */
        break;             /* increment countb, leave switch. */
    default:
        countc++;          /* c == none of the above; increment */
        break;             /* countc, then leave switch. */
}
```

This example increments `counta` when `c` is 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9; and it increments `countb` when `c` is a space, tab, or NEW LINE. If `c` has none of those values, the example increments `countc`.

Related Topics

See also: `switch`, `break`, and `default`.

char

keyword

This object is data type character.

Syntax

[unsigned] char declarator-list;

Description

char is one of the four data type specifiers: char, int, float, and double. you can use the data type adjective unsigned with char.

char gives a variable a single byte of storage, 8 bits. Your program can use a char variable to store any one of the standard ASCII characters. The value of a char variable is the integer code for the character it contains; in Data General C this means the possible values range from 0 to +255. However, if you use the /SIGNEDCHAR switch on the C compiler, the default range for normal character variables is -128 to +127.

char is useful for declaring character arrays, pointers to characters, and functions that return characters. Since it's possible to get the integer code for a character, you can do greater-thans and less-thans with char variables, and this is useful for comparing and alphabetizing arrays of characters.

You can't have negative characters without using the /SIGNEDCHAR switch on the C compiler, as mentioned above. Otherwise, to capture a signed character, like EOF, use an int.

You can store an integer in a character variable, but the integer's value will be truncated if it is out of range.

Examples

```
char thing;          /* declares a one-character variable named */
                     /* thing.                                */

char *pthing;        /* declares a pointer, named pthing, to a */
                     /* character.                                */

pthing = thing;       /* sets the character-pointer pthing to */
                     /* point to the character thing.        */
```

```

char athing[];          /* declares an array, named athing, of      */
                        /* characters.                                     */

char fthing(); /* declares a function, named fthing, that */
               /* returns a character.                                     */

char *fpthing(); /* declares a function, named fpthing, that */
                 /* returns a pointer to a character.                                     */

char thing = '\n';      /* declares thing as a character and      */
                        /* initializes it to NEW LINE character. */

```

Related Topics

See also: int, unsigned, sign-extension, char constants,
the /SIGNEDCHAR and /UNSIGNEDCHAR switches on the C compiler.

continue keyword

Jump to the end of the current loop.

Syntax

```
continue ;
```

Description

A continue statement causes the program to immediately begin the next iteration of the smallest enclosing loop. continue is for use with while, for, and do.

- * In while and do, continue causes immediate execution of the text part.
- * In for loops, continue causes control to pass to the reinitialization step.

continue won't stop at a switch, though: a continue inside a switch inside a loop directs the program to the beginning of the loop.

By contrast, the break statement causes the program to leave the loop entirely. In general, continue tends to be used less often than break.

Example

This routine skips the negative elements in the array spooky:

```
for (i=0; i < N; i++)  
{  
    if (spooky[i] < 0)  
        continue;          /* skip negative elements of spooky. */  
                            /* else clause is not needed.      */  
    ...                    /* process positive elements.    */  
}
```

Related Topics

See also: while, for, do-while, and break.

default keyword

Default case for the preceding switch.

Syntax

default : statement

Description

The default prefix is the label for the default case in a switch statement. If the outcome of the switch does not match any of the cases, the program proceeds to the statement that follows default, if there is one. The default statement is optional; if no default prefix appears and none of the cases match, the program simply leaves the switch and proceeds.

case and default statements can occur in any order.

The default statement is useful for preventing the program from falling through the switch statement without doing anything. The case and default prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes.

To exit from a switch, use break.

Example

Please see the example in the sections on case and switch.

Related Topics

See also: switch and case.

do

keyword

Do the following, then see if expression is still true; repeat while true.

Syntax

```
do statement while ( expression ) ;
```

Description

The do-while combination is one three methods in C for creating loops: while, for, and do-while. With do, the test takes place after each execution of the substatement. This means no test takes place before the first execution of the statement. While the test is true -- in other words, while expression is nonzero -- execution repeats until the value of expression becomes zero.

do is useful when you want a loop that goes back over previous code rather than forward over upcoming code.

Example

This example shows a do-while loop.

```
int i = 0, j = 0;
do
{
    i += 1;
    j = i;
}
while (i <= 10) ;
```

/* loop begins here. */

/* loop ends here. */

/* test for exit is here. */

Related Topics

See also: while, for.

double keyword

This object is a double-precision floating-point number.

Syntax

[{short | long}] double declarator-list;

Description

double is one of the four data type-specifiers: char, int, float, and double. You can use the data type adjectives short and long with double.

When C must convert a double to a float, it rounds the double, then truncates it to float length.

double is useful for large floating-point numbers. The following list shows the size limits that integers, floats, and doubles enjoy in C:

int	32 bits
short int	16 bits
long int	32 bits
float	32 bits: 23 mantissa + 8 exponent + 1 sign
short float	same as float
long float	64 bits: 55 mantissa + 8 exponent + 1 sign
double	same as long float
short double	same as float
long double	same as long float

Examples

This creates a double-precision number r and initializes it to 0:

```
double r = 0;
```

This declares a function atoi that returns a double-precision number:

```
double atoi();
```

else keyword

Otherwise, do the following.

Syntax

```
if (expression)
    statement-1
else
    statement-2
```

Also:

```
else if (expression)
    statement
```

Description

else is for use with the if statement. if is for making tests and decisions; it works much like if-else combinations in other languages. If the expression in the if statement comes out false, and there is an else statement, the program skips over the if and its statement(s) and goes to the else.

The else-if construction is also useful for offering more alternatives to the if:

```
if (expression)
    statement
else if (expression)
    statement
else
    statement
```

CAUTION: Nested if and else statements can produce apparent ambiguities; to avoid them, use braces, as in this example:

```
if (expression-1)
{
    if (expression-2)
        statement-1
}
else
    statement-2
```

The braces are needed to show which if the else belongs to; the else belongs to the outer if statement. If the brackets weren't there, the C compiler would automatically associate the else with the last else-less if it encountered; the inner if statement. In other words, without the braces, statement-2 would be executed as a result of expression-2, not expression-1.

Example

This example uses if, else if, and else to test the value of cond.

```
if (cond < 0)                /* cond is low, so      */
    high = mid - 1;          /* eliminate high half. */
else if (cond > 0)           /* cond is high, so      */
    low = mid + 1;          /* eliminate low half.   */
else                         /* found it;             */
    return (mid);           /* tell where.           */
```

Related Topics

See also: if, the conditional operator, ?:, and the switch and case statements.

entry keyword

The C language currently reserves the entry identifier for future implementation. The C compiler won't let you use entry as an identifier.

enum keyword

The following constants are the permitted values for the enum data type, enum's-name.

Syntax

```
enum enum's-name
{
    enum-list
}
declarator-of-this-kind-of-enum;
```

where enum list is a comma-separated list of: identifier, or
identifier = constant-expression.

Description

enum is one of the derived types, which include arrays, structs, unions, and enums. enums provide a means for making scalar variables.

Unlike the other derived types in C, the elements of an enum are set to constants, as follows: The enum declaration specifies identifiers for these constants. The identifiers, in turn, must be set to constant values; they can either be set in the declaration or later in the program.

enum values have scope only in the functions in which the enum type is declared. By contrast, #define constants have scope throughout the remainder of the program file.

enum is useful for creating lists of constants whose scope is limited to the enclosing function, and for permitting scalar variables.

Examples

We declare our enumerated type, rainbow, as having a range of possible values from "radio" to "xray"; we also initialize radio to 10 and infrared to 25:

```
enum rainbow {radio=10, infrared=25, visible, ultrav, xray};
```

NOTE: This example doesn't give any values to visible, ultrav, or xray. If they don't get explicit values elsewhere, the C compiler will make visible 26, ultrav 27, and xray 28.

We declare a variable, band, as being of type rainbow; in other words, band can ONLY have one of the five values in the braces above:

```
rainbow band;
```

Finally, we initialize band:

```
band = visible;
```

Or, alternatively, we can declare and initialize everything in one statement:

```
enum rainbow {  
    radio=10,  
    infrar=25,  
    visible,  
    ultrav,  
    xray  
} band=visible;
```

Related Topics

See also: constants, the #define preprocessor statement.

extern keyword

This object's storage class is extern; the object is visible outside the current compile-file.

Syntax

type-specifier extern declarator-list;

Description

extern is one of the four storage class specifiers: auto, static, register, and extern.

extern is useful for co-ordinating global variables in large programs, e.g., programs whose code spreads over several files.

Using Data General C, the following declaration

extern identifier;

will cause the compiler to make identifier an integer. Furthermore, by default, i.e., if you don't use the /EXTL switch on the compiler, the compiler will generate storage for the identifier. All extern variables with the same name that are in the same program will get the same storage location. On the other hand, if you use the /EXTL switch on the compiler, you must declare the variable somewhere, either in one of the other files to be compiled, or in assembly language.

You can repeat the declaration -- in other words, you can declare the same extern variable twice in two different places -- but the data type must be the same everywhere.

You CAN declare variables that are inside functions to be extern.

Examples

Let's say we compile our program from three separate source files: moduleA, moduleB, and moduleC. All three use the variable notepad to communicate with each other. We can declare notepad fully in moduleA (it's an integer), then mention it to be an extern in moduleB and moduleC:

ModuleA

ModuleB

ModuleC

extern int notepad;

extern notepad;

extern notepad;

It's also permissible to declare it fully in more than one function, but the declarations must be the same:

```
extern int doctor;      extern int doctor;      extern doctor;
```

Related Topics

See also: auto, static, register, and typedef.

float keyword

This object is a single-precision floating-point number.

Syntax

[{short | long}] float declarator-list;

Description

float is one of the four data type-specifiers: char, int, float, and double. float means a floating-point number; that is, a number that has a fractional part. You can use the data type adjectives short and long with float.

The following list shows the size limits that integers, floats, and doubles enjoy in C:

int	32 bits
short int	16 bits
long int	32 bits
float	32 bits: 23 mantissa + 8 exponent + 1 sign
short float	same as float
long float	64 bits: 55 mantissa + 8 exponent + 1 sign
double	same as long float
short double	same as float
long double	same as long float

float is useful for simply making your program use less memory.

NOTE: If you do not use the /SHORTARITHMETIC switch on the compiler, C carries out all floating point arithmetic in double-precision. C lengthens all floats that appear in expressions by zero-padding the fraction. This means that you get the full benefit of C's accuracy during computations. But then the result will get stored back into the same size variable you started with.

Alternatively, when C converts a double to a float, it first rounds the double, then truncates it to float length.

When converting floats to integers, DG C truncates the fractional part.

Example

The following statement declares three floating-point real numbers:

```
float dog, bone, city;
```

Related Topics

See also: double, long, short.

for keyword

Do expression-1; then, while expression-2 is true, do the following statement and expression-3.

Syntax

```
for ( [expression-1]; [expression-2]; [expression-3] ) statement
```

Description

for is one of the three statements that create loops: while, for, and do-while.

As long as expression-2 is true (not equal to zero) expression-3 and the substatement will execute. In other words,

```
for (expression-1; expression-2; expression-3) statement
```

behaves as if:

```
expression-1;  
while (expression-2)  
{  
    statement  
    expression-3  
}
```

The functionality of for is based on the fact that often, expression-1 and expression-3 are assignments or function calls and expression-2 is a comparison or test.

For is useful for keeping loop control statements together:

```
for (i = 0; i < N; i++)  
    peel mango[i];
```

This is C for: process the first N elements of the array, mango.

Your program can alter the limits of a for loop from within the loop. However, the controlling variable -- i in the following example -- retains its value when the loop terminates for any reason.

```
for (i=0; i < N; i++) {  
    zoom c[i];  
    if c[i] == 80000  
        i=N;  
}
```


expression-1, expression-2, and expression-3 are all optional; the semicolons that separate them are not:

```
for (;;)
{
    ...
}
```

This would produce an endless loop, which the program would have to exit by other means; e.g., using the break statement.

Examples

This loop skips white space -- spaces, NEW LINEs, and tabs:

```
for (i=0; s[i]!=' ' || s[i]!='\n' || s[i]!='\t'; i++)
;
```

In this example, two of the expressions contain multiple operations. In expression-1 we initialize i and j (which requires a function call), in expression-2 we test both i and j, and finally in expression-3 we increment i and decrement j:

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
{
    ...
}
```

Note that the commas are functioning as operators, guaranteeing that the expressions get evaluated left to right.

Related Topics

See also: loops in general, the while and do statements, and the comma operator.

goto

keyword

Jump to the statement at label.

Syntax

```
goto identifier ;
```

Description

goto is the most fundamental of flow-control statements; It is useful for instructing the program to go somewhere other than the next statement. goto can go only to a label.

A label is simply a legal C identifier followed by a colon followed by a statement. You can attach a label to any statement, and a statement can have more than one label. The scope of a label is the function in which it is defined. In other words, labels can be reached only by gotos inside the same function.

Although unfashionable nowadays, goto can still prove useful when a program needs to stop processing inside some deeply nested code (see the example below).

Example

This example shows some nested while loops and a goto statement to get out of them. "error:" is a label.

```
while ( ... )
  while ( ... )
    while ( ... )
      {
        ...
        if ( calamity )
          goto error;
      }
    ...
  error:
  ( first, read the manual )
```

Related Topics

See also: labels, flow control.

if

keyword

If (expression) is true, do the following.

Syntax

```
if (expression)
    statement-1
else
    statement-2
```

Description

The if statement represents one of several ways that C offers to make tests and decisions: if-else, the ?: operator, the switch and case statements, and the && and || operators. You use if to make a decision. To get the alternate case, use an else statement.

An if statement tests the numeric value of an expression. If the expression is true (not equal to zero), the program executes statement-1; if it is false (equal to zero) and there is an else part, the program executes statement-2.

Your program can get cumbersome when you nest if-elses. The switch statement can tidy up such nests considerably.

Watch out for:

```
if (expression-1)
{
    if (expression-2)
        statement-1
}
else
    statement-2
```

In this example, you need the braces to make sure the else gets associated with the right if, otherwise, like most other languages, C associates an else with the last else-less if.

Example

This example shows an if followed by a compound statement followed by an else followed by a simple statement.

```
if (done == 1)
{
    gohome (hat, coat);
    done = 0;
}
else
    phone (date, explain);
```

Related Topics

See also: the conditional operator, `?:`, the switch and case statements, and the `&&` and `||` operators.

int

keyword

This object is an integer.

Syntax

```
int declarator-list;
short [int] declarator-list;
long [int] declarator-list;
[{short | long}] unsigned [int] declarator-list;
unsigned [{short | long}] [int] declarator-list;
```

Description

int is one of the four data type specifiers: char, int, float, and double. You can use all three data type adjectives with int: short, long, and unsigned.

int is the default data type in C, for both data and functions. Once you've said long, short, or unsigned, you can omit "int".

Both chars and pointers are integers underneath.

In C, integers are useful for: counting things, pointing to things, storing characters, especially negative ones.

The following list shows the size limits that integers, floats, and doubles enjoy in C:

int	32 bits
short int	16 bits
long int	32 bits
float	32 bits: 23 mantissa + 8 exponent + 1 sign
short float	same as float
long float	same as double
double	64 bits: 55 mantissa + 8 exponent + 1 sign
short double	same as float
long double	same as same as double
char	8 bits

Examples

```
int thing;           /* declares an integer named thing.      */

int *pthing;         /* declares a pointer named pthing to an integer not specified yet. */

pthing = &thing;      /* sets the integer-pointer pthing to the address of the integer thing. */

int *athing[3];       /* declares an array, named athing, of three pointers to integers. */

int (*pathing)[];     /* declares a pointer, named pathing, to an array, not specified here, of integers. */

pathing = athing;     /* sets the array-pointer, pathing, to point to the array athing. */

int *fthing();        /* declares a function, named fthing, that returns a pointer to an integer. */

int (*pfthing)();     /* declares a pointer, named pfthing, that points to a function, not specified here, that returns an integer. */
```

long keyword

This object is a large number.

Syntax

```
long [int] declarator-list;
[short | long] unsigned [int] declarator-list;
unsigned [short | long] [int] declarator-list;

[short | long] {double | float} declarator-list;
```

Description

long is one of three data type adjectives: long, short, unsigned.
You can use long with the data type specifiers int, double, and float.

long is useful for storing large numbers, and for getting greater precision than that offered by float.

The following list shows the size limits that integers, floats, and doubles and their long and short versions enjoy in C:

int	32 bits
float	32 bits: 23 mantissa + 8 exponent + 1 sign
double	64 bits: 55 mantissa + 8 exponent + 1 sign
short int	16 bits
short float	same as float
short double	same as float
long int	32 bits
long double	same as double
long float	same as double

Example

long gemini;	an integer, gemini, having 32 bits of storage
long double gemini;	both these declarations give the double-precision variable, gemini, 64 bits of storage.
double gemini;	

register keyword

This object has storage class register.

Syntax

type-specifier register declarator-list;

Description

register is one of the four storage class specifiers: auto, static, register, and extern

Under AOS/VS, the Data General C compiler treats variables of type register the same as auto.

Your program can get the address of a register variable, using the address operator, but the C compiler gives a level 1 error message.

Example

In this example we are declaring a function called high.

```
register high (cloud, fog);    /* formal parameters of high -- */
register int cloud, fog;      /* cloud, fog -- are storage class */
{                             /* register, data type int.    */
    register ion;             /* a local variable, class register */
    ...
}
```

Related Topics

See also: extern, and the external variables and functions in general.

return

keyword

Return to the calling routine.

Syntax

return; returned value is not defined.

return expression; returned value is converted to
the type of the function.

Description

You use return to send a value from a called function back to the calling function.

The return statement is the only way in C for a called function to return a value to the calling function. Otherwise, to get back to the caller, execution has to "fall off the end" of the function by reaching the closing right brace, and in that case the returned value is not defined. Either way, execution will return to the statement following the statement that called the function.

Any expression can follow return. Before the expression goes back to the calling function, C will resolve it into a single value. Such an expression must be the same type as the function itself; if not, C will convert it to the function's type if possible.

Recall that functions default to int; otherwise, you must specify the data type the function will return.

The calling function is free to ignore the returned value. A function need not return a value; return by itself causes control to go back to the caller, but no useful value is returned.

You may put as many return statements as you wish inside a function.

Example

This example sets x equal to the value that the function, flannel, returns:

```
main()
{
  int a, b, c, x, flannel();
  ...
  x = flannel(a, b, c);    /* set x equal to flannel's result. */
  ...
}

flannel (first, second, third)
int first, second, third;
{
  int y;
  ...
  return(y);              /* send value of y back to caller. */
}
```

Related Topics

See also: functions in general and break.

short keyword

This object is a short number.

Syntax

```
short [int] declarator-list;
[short] unsigned [int] declarator-list;
unsigned [short] [int] declarator-list;

[short] {double | float} declarator-list;
```

Description

short is one of the three adjectives for the data type specifiers: short, long, and unsigned.

short is useful for instructing the compiler to allocate a minimum of space.

The following list shows the size limits that integers, floats, and doubles and their short and long versions enjoy in C:

int	32 bits: 31 mantissa + 1 sign
float	32 bits: 23 mantissa + 8 exponent + 1 sign
double	64 bits: 55 mantissa + 8 exponent + 1 sign
short int	16 bits: 15 mantissa + 1 sign
short float	same as float
short double	same as float
long int	same as int
long double	same as double
long float	same as double

Example

```
short forest; /* declare a short integer named forest. */
```

Related Topics

See also: long.

sizeof

keyword

Determine the number of bytes an object occupies.

Syntax

```
sizeof (expression)
sizeof (type name)
sizeof unary-expression
```

Description

sizeof is a compile-time unary operator that computes the size of any object, including data types themselves. sizeof yields an integer constant representing the number of bytes in the object. For example:

Object	sizeof Yields:
variable	total number of bytes
array	bytes in one element times number of elements
struct	bytes in one structure times number of structures

Your program will find sizeof useful when it needs to communicate with storage-allocating, buffer-managing, and I/O functions.

Do NOT use sizeof with: labels, functions, arrays whose highest bound is not specified, and structures that haven't been declared yet.

Example

This example uses sizeof to find out how many bytes the structure orange occupies; then it calls alloc with this number and sets p to point to an orange-type structure:

```
struct orange {
    struct orange *next;
    char name[100];
} *p;

p = (struct orange *)alloc(sizeof(struct orange));
```

static keyword

This object has storage class static.

Syntax

type-specifier static declarator-list;

Description

static is one of the four storage class specifiers: auto, static, register, and extern.

static makes the variable keep its value, even when the program isn't executing the block or function in which the variable occurs. static variables are useful as private permanent storage for a function.

NOTE: When you declare a static variable inside a function, the variable's scope is just that function, and the variable is said to be an internal static variable.

When you declare a static variable outside all the functions, its scope is the rest of the source file, and it is said to be an external static variable.

Don't confuse these distinctions with the storage class extern; extern lets a variable have scope outside the entire compile-file, regardless of whether it is declared inside a function or not.

static variables are useful for keeping local variables around after a function is exited. By contrast, variables declared as storage class auto will cease to exist between executions of a function.

External static variables won't conflict with the same variable names in other files of the same program. By contrast, the C compiler WILL compare an extern variable to any other extern variable it finds with the same name in the same program.

Your program will initialize its static variables only once, before the program starts executing. The C compiler guarantees that static variables will be initialized as zero (or null), unless your program initializes them.

Examples

These variables were used by two different functions that touched the same buffer; they were declared outside the functions:

```
static char buf[BUFSIZE];  
static int  bufp = 0;
```

struct keyword

This is a user-defined data type: it has the following name, is composed of the following data elements, and the following variables are of this type.

Syntax

```
struct struct's-name
{
    members of the structure
}
declarators-of-this-kind-of-struct;
```

Description

struct is one of the derived data types, which include arrays, structs, enumerations, and unions. structs are useful for defining aggregate data. By aggregate data we mean data that is composed of pieces of data of different types; by contrast, arrays are made up of pieces of data of the same type. structs are like records in PASCAL, and like structures in PL/I.

structs are useful for linked lists, trees, and matrixes. structs also allow you to create a unit of storage to be used bit-by-bit (such units are called fields in C).

The struct's-name and declarators-of-this-kind-of-struct are optional. You can have self-referential structures; a member of a structure can be a pointer to another structure of the same type. The pointers are useful as links in a linked list.

Note the following DG implementation differences for structs:

- * Your program can copy one structure into another, as in:

```
noun = verb;
```

where noun and verb are both the same kind of structure.

- * Your program can pass structures to functions. The passing takes place by value. The function can return the structure.
- * Your program can have a member of a structure that is a pointer to another, as-yet-undeclared structure. The program must declare the undeclared structure by the end of the current block, or before the beginning of a function. This is useful if you have two structures that contain pointers to each other.

Examples

To define a data type that can serve as a node in a binary tree, and to declare a variable of that type:

```
struct treenode      /* define struct type named treenode. */
{
    char[12] leaf;
    int leafcount;
    *treenode leftnode;
    *treenode rightnode;
} tree1;              /* declare variable, tree1.          */
```

To declare some more variables of type treenode:

```
struct treenode tree2, tree3, tree4;
```

This example defines a field:

```
struct flagz {
    unsigned harpo   : 1;
    unsigned chico   : 1;
    unsigned groucho : 6;
} opera;
```

This defines a structure type, called flagz, and declares a field variable of that type, called opera. All the filed variables of type flagz will have two 1-bit fields and a 6-bit field; the number after the colon specifies the width in bits.

The colon in the declaration above causes the C compiler to allocate bits, specifically, the number of bits given by the expression on the right side of the colon. The colon causes the variable to default to data type unsigned int.

You cannot get the address of a variable declared with a colon, nor can you create an array of bits.

switch keyword

Evaluate expression; if the result matches a case, perform the statement(s) that follow the case.

Syntax

switch (expression) compound-statement

where the prefix

default:

may precede at most one statement inside compound-statement; and prefixes of the form

case constant-expression: [statement]

may precede other statements inside compound-statement.

The compound-statement itself is usually composed of a series of case statements, as in:

```
{  
  case constant-expression:  
    statement  
  case constant-expression:  
    statement  
  case constant-expression:  
    statement  
  ...  
}
```

Description

The switch statement is one of C's several ways to make tests and branch on conditions: if and if-else, switch-case, the conditional operator `?:`, and the `&&` and `||` operator. Unlike the essentially two-way branching offered by the others, switch offers multi-way branching. Thus, switch is useful for deciding among a wide range of values. In particular, it is useful for replacing a series of if...else if...else if...else statements.

The switch statement cause control to transfer to one of a set of statements, depending on the value of expression. C performs the usual arithmetic conversion on the expression; the result of expression must be type int.

The statement that follows switch is typically a compound statement. It can include one or more case statements, as shown in the example below, and one default statement. A simple statement would permit only one case, and thus would defeat the purpose of switch.

Not all the possible outcomes of expression need be accounted for with a case statement; a default statement will handle the remaining outcomes.

Example

The following example shows a complete switch statement that uses several case statements and a default statement. Depending on the value of c, the program will select the case whose constant matches the value of c.

```
switch(c)
{
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
        counta++;           /* c == 0 through 9; increment */
        break;             /* counta, then leave switch. */
    case ' ':
    case '\t':
    case '\n':
        countb++;           /* c == space, tab, or NEW LINE; */
        break;             /* increment countb, leave switch. */
    default:
        countc++;           /* c == none of the above; increment */
        break;             /* countc, then leave switch. */
}
```

This example increments counta when c is 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9; and it increments countb when c is a space, tab, or NEW LINE. If c has none of those values, the example increments countc.

typedef keyword

This name is now a synonym for the given data type-specifier.

Syntax

typedef type-specifier name;

Description

The typedef keyword lets you create synonyms for C data types and also for your own data type definitions. You can use these synonyms later to declare things -- as if the synonyms were keywords naming data types.

Remember, typedef does NOT create a new type, it simply gives a synonym for an existing type.

typedef is useful for abbreviating long declarations. is is also useful for setting up a program's code to be more easily portable to another computer. For example, suppose you have to move a program to another computer and you're not sure the int variables will work properly. Rather than change all the int declarations in your program, you can create a type called INTPORT:

typedef int INTPORT;

Now you can declare all variables that must be integers as simply INTPORTs, then later, when you move the program, you can change the single INTPORT definition later to whatever you need -- int, or something else.

typedef does not define any objects or storage; it just defines an identifier.

Examples

The declaration

typedef int OUT;

makes OUT a synonym for int. In other words, OUT will cause variables to be integers just as int will.

Using the following typedef definitions:

```
typedef int feet, *metersp;  
typedef struct { ... } complex
```

You can declare:

```
feet distance;  
metersp metricp;  
complex z, *zp;
```

The type of distance works out to be int; metricp works out to be "pointer to int"; z and *zp work out to be a structure and a pointer to one of those structures.

union keyword

This is an area of storage for varying types, as follows.

Syntax

```
union union's-name
{
    members of the union
}
declarators-of-this-kind-of-union;
```

Description

union is one of the derived data types, which include arrays, structs, enumerations, and unions. A union is like a structure in which all the members begin at offset 0 and which is large enough to contain any of its members. At any one time, only one member can be stored in the union.

unions are useful for holding, at different times, objects of different types and sizes. The C compiler keeps track of size and alignment requirements inside the union.

The union's-name and declarators-of-this-kind-of-union are optional.

Using the Data General C compiler, you can have your program initialize a union. The compiler assumes that the program is initializing the first member of the union.

Example

Here is a union called student, with three members: a, b, and c.

```
static union student
{
    int a;
    double b;
    char[] c;
}
```

Now we declare mike to be a student; and then we initialize mike:

```
student mike = 2;
```

This will set mike.a equal to 2.

unsigned

keyword

This object has no sign.

Syntax

```
[unsigned] char declarator-list;
[short | long] unsigned [int] declarator-list;
unsigned [short | long] [int] declarator-list;
```

Description

unsigned is one of the three adjectives (long, short, unsigned) for the four data types. You can use unsigned to modify char and int.

unsigned numbers are useful for performing calculations that will always be positive. unsigned characters are useful to insure that value of a char is always in the range 0 through +255, even if you have used the /SIGNEDCHAR switch on the compiler, which gives normal characters the range -128 through +127.

When an unsigned integer and a plain integer combine in a C program, C converts the plain integer to unsigned and the result is unsigned. The value of the result will be the least unsigned integer congruent to the signed integer (modulo 2^{wordsize}). In a twos complement representation like Data General's, this conversion is conceptual and there is no actual change in the bit pattern.

unsigned also releases the sign bit to be a mantissa bit; the following list shows the size limits that signed and unsigned integers enjoy in C:

short unsigned int	16 bits: 16 mantissa
int	32 bits: 31 mantissa + 1 sign
unsigned int	32 bits: 32 mantissa
long unsigned int	same as unsigned int

Example

This statement declares an array of 256 unsigned characters.

```
unsigned char ebkadic[256]; /* an array of unsigned chars. */
```

Related Topics

See also: sign-extension, char.

void

keyword

This function returns nothing at all.

Syntax

```
void function-name();
```

Description

void lets you declare a function as returning nothing.

void is useful for preventing meaningless return values from appearing to the caller function.

You can use void only with functions, and with no other kind of object. You can use void in a cast, however, to convert the result of a function:

```
(void) nausee(j, p, s);
```

Example

This statement declares the function rien() to be one that returns nothing:

```
void rien();
```

while

keyword

While expression is true, do the following.

Syntax

```
while (expression) statement
```

```
while (expression)
{
    ...
}
```

Description

while is one of the three C statements that create loops: while, for, and do-while.

As long as the expression remains true (not equal to zero), the program will execute the substatement. Since the expression is evaluated first, it's possible that the loop won't be executed at all; if the expression is false, the program will skip the statement altogether.

The alternative to while is do-while, which performs a loop and makes its termination test at the end of the loop.

When while is set up like this:

```
expression-1;
while (expression-2)
{
    ...
    statements
    ...
    expression-3;
}
```

it can often be replaced with a for statement:

```
for (expression-1; expression-2; expression-3)
{
    ...
    statements
    ...
}
```

Example

The following example copies characters from the standard input to the standard output until a NEW LINE or EOF arrives. It does not copy the NEW LINE:

```
while (((c=getchar()) != EOF && (c != '\n'))
    putchar();
```

End of Chapter

CHAPTER 8

DICTIONARY OF C OPERATORS

This chapter describes all the C operators. The previous chapter describes all the C keywords alphabetically. Both chapters are designed for quick and easy reference; together, they form a complete dictionary of the C language.

The C operators are:

+	&	^		?:	--	/	=
==	\	>	>=	++	*	!=	<<
<	<=	&&	!		%	*	~
>>	.	->	-	-	sizeof	(type-name)	

sizeof is a unary compile-time operator that yields the size, in bytes, of its operand. sizeof is a reserved word in C, and it is described in detail in the C Keywords chapter.

(type-name) stands for the cast function in C, which works like an operator to change the type of an expression. Casts are explained in Chapter 3.

Table 8-1. Precedence of Operators in C

Operator	What Is It?	Associativity
PRIMARY-EXPRESSION operators (all have same precedence)		
		left to right
()	grouping	
[]	array reference	
primary-expression -> identifier	in-structure pointer	
primary-lvalue.identifier	structure member	
UNARY operators (all have the same precedence)		
		right to left
! expression	logical NOT	
~ expression	one's complement	
++lvalue, lvalue++	increment	
--lvalue, lvalue--	decrement	
-expression	unary minus	
(type-name) expression	cast	
* expression	indirection ("points to")	
& lvalue	"address of"	
sizeof (expression), sizeof (type-name)	how many bytes in	
BINARY and TERNARY operators (precedence in the order listed)		
MULTIPLICATIVE operators (all have the same precedence)		
expression * expression	multiplication	left to right
expression / expression	division	left to right
expression % expression	modulus	left to right
ADDITIVE operators (both have the same precedence)		
expression + expression	addition	left to right
expression - expression	subtraction	left to right
SHIFT operators (both have the same precedence)		
expression << expression	shift left	left to right
expression >> expression	shift right	left to right

(continues)

Table 8-1. Precedence of Operators in C

Operator	What Is It?	Associativity
BINARY and TERNARY operators (continued)		
RELATIONAL operators (all have same precedence)		
expression < expression	less-than	left to right
expression > expression	greater-than	left to right
expression <= expression	less-than-or-equal	left to right
expression >= expression	greater-than-or-equal	left to right
EQUALITY operators (all have the same precedence)		
expression == expression	equality ("is equal to")	left to right
expression != expression	inequality ("is not equal to")	
BITWISE and LOGICAL operators (have precedence in the order listed)		
expression & expression	bitwise AND	left to right
expression ^ expression	bitwise exclusive OR	left to right
expression expression	bitwise inclusive OR	left to right
expression && expression	logical AND	left to right
expression expression	logical OR	left to right
CONDITIONAL operator		
expression ? expression : expression	conditional expression ("if-then-else")	right to left
ASSIGNMENT operator		
lvalue = expression	assign	right to left

(continued)

Table 8-1. Precedence of Operators in C

Operator	What Is It?	Associativity
BINARY and TERNARY operators (continued)		
COMPOUND ASSIGNMENT operators (all have the same precedence)		
lvalue += expression	increment and assign	right to left
lvalue -= expression	decrement and assign	right to left
lvalue *= expression	multiply and assign	right to left
lvalue /= expression	divide and assign	right to left
lvalue %= expression	divide and assign	right to left
lvalue >>= expression	shift right and assign	right to left
lvalue <<= expression	shift left and assign	right to left
lvalue &= expression	perform bitwise AND and assign	right to left
lvalue ^= expression	perform bitwise exclusive OR and assign	right to left
lvalue = expression	perform bitwise inclusive OR and assign	right to left
COMMA OPERATOR		
expression, expression	expression separator	left to right

(concluded)

+ Addition operator

operator

Gives the sum of the operands.

Syntax

expression + expression

Description

The addition operator + yields the sum of the operands. C performs the usual arithmetic conversions with this operation. + has the same precedence as the subtraction operator - .

You can add pointers to integers and integers to pointers.

When adding an integer to a pointer, C converts the integer to an address offset by multiplying it by the length of the object to which the pointer points. The result is a new pointer of the same type as the original pointer. The new pointer points to another object in the same array, appropriately offset from the original object.

For example, if p points to an element in an array,

p + 1

points to the next element in the array.

You can combine the + operator with the = operator to form an assignment operator, as in

a += b

which means: a = a + b.

You can chain addition operations together, as in:

a + b + c + d

In such constructions, C may rearrange the order of evaluation.

You cannot add pointers to a type other than an integral type.

Related Topics

See also: the subtraction operator, pointers.

& Address operator operator

Gives the address of the operand.

Syntax

&lvalue

Description

The address operator & yields the address of the operand; in other words, the address operator yields a pointer. If the type of the lvalue is "abc", the type of the result of an address operation is "pointer to abc".

You can only take the address of an lvalue; i.e., an object that can legally appear on the left side of an assignment statement.

Once you have an address, you can use the indirection operator, * , to see what's at the address.

Examples

These examples show how the address operator works.

```
char p;          /* declares a character variable named p. */
char *q;         /* declares a pointer, named q, to a char. */
&p              means "the address of p".
q = &p          /* sets q equal to the address of p. In */
                /* other words, it makes q point to p. */
*q              now means "what q points to" (it points to p).
```

= Assignment operator operator

Assigns the value of the right operand to the left.

Syntax

lvalue = expression

Description

The assignment operator = assigns the value of expression on the right to the lvalue operand on the left. The result of the expression is then the value of its left operand; and the type of the expression is then the same type as the type of the left operand. The operand on the left MUST be an lvalue; that is, it must resolve to a single object:

(a + b) = c; NOT LEGAL

c = (a + b); LEGAL

When it does an assignment, C converts the right operand to the type of the left operand, using the usual arithmetic conversions of C.

You can chain assignment operations together, as in:

this = that = theother = 0;

In such constructions, the assignment operator groups right-to-left, producing:

this = (that = (theother = 0));

which sets all three variables to 0.

You can assign a pointer to an integer, an integer to a pointer, and a pointer to a pointer of another type. If the pointers have different alignments (e.g., byte-aligned or word-aligned) the compiler will automatically convert the alignment. For example, assigning a character pointer to an integer pointer will proceed smoothly; shift the byte pointer right one bit.

Compound Assignment Operators

Compound assignment operators provide a shorthand notation for assigning the result of some operation to an lvalue. For example, you can write this expression

```
i = i + 1;
```

using a compound assignment operator like this:

```
i += 1;
```

The compound assignment operators are:

```
+= -= *= /= %= <<= >>= &= ^= |=
```

You'll find more information under each individual operator (i.e., + - * / % << >> & ^ |).

Conversion by Assignment

In C, data type conversions can take place across assignments, as described in the section "Usual Arithmetic Conversions", in Chapter 3.

& Bitwise AND operator

operator

Performs the bitwise AND function on the operands.

Syntax

expression & expression

Description

The bitwise AND operator & is useful for bit manipulation, like masking off some bits. For example,

d = p & 0177;

sets to zero in d all but the low-order 7 bits of p.

C performs the usual arithmetic conversions. The & operator is associative and applies only to integral operands.

You can chain bitwise AND operations together, as in:

a & b & c & d

In such constructions, C may rearrange the order of evaluation.

You may combine the & operator with the = operator to form an assignment operator, as in

a &= b

which means: a = a & b.

Related Topics

See also: bitwise inclusive OR, bitwise exclusive OR, left-shift, right-shift, one's complement operators, and &&.

^ Bitwise exclusive OR operator operator
Performs the bitwise exclusive OR function on the operands.

Syntax

expression ^ expression

Description

The bitwise exclusive OR operator ^ is useful for turning bits on. The operands to ^ must be integers. C performs the usual arithmetic conversions with this operator.

You can chain bitwise exclusive OR operations together, as in:

a ^ b ^ c ^ d

In such constructions, the C compiler may rearrange the order of evaluation.

You may combine the ^ operator with the = operator to form an assignment operator, as in

a ^= b;

which means: a = a ^ b.

Related Topics

See also: bitwise AND, bitwise inclusive OR, left-shift, right-shift, and one's complement operators.

| Bitwise inclusive OR operator operator

Performs the bitwise inclusive OR function on the operands.

Syntax

expression | expression

Description

The bitwise inclusive OR operator | is useful for turning bits on. For example

device = device | MASK;

sets to 1 in device the bits that are set to 1 in MASK.

C performs the usual arithmetic conversions with this operation. Both operands must be integral types.

You can chain bitwise OR operations together, as in:

a | b | c | d

In such constructions, the C compiler may rearrange the order of evaluation.

You may combine the | operator with the = operator to form an assignment operator, as in

a |= b;

which means: a = a | b.

NOTE: Don't confuse | and & with the logical connectives && and || .

Related Topics

See also: bitwise AND, bitwise exclusive OR, left-shift, right-shift, and one's complement operators.

?: Conditional operator

operator

Performs an if-then-else using the three expressions.

Syntax

expression ? expression : expression

Description

The conditional operator ?: provides an alternative to the if-else construction. Instead of writing

```
if expression-1
    expression-2
else
    expression-3
```

you can write

expression-1 ? expression-2 : expression-3

The order of evaluation is as follows: C evaluates the first expression. If it is non-zero, C evaluates the second expression. But if the first expression is zero, C evaluates the third expression. Thus, C evaluates either the second or the third expression, but not both.

You can nest conditional operations, and you can chain them. When chained, C groups the conditional operations right-to-left.

When performing arithmetic conversions among the operands, C attempts to bring the second and third expressions to a common type; C follows the usual arithmetic conversions here. If one of the expressions is an integer, for example, 0, and the other expression is a pointer, the result will have the pointer's type.

Related Topics

See also: if-else and switch-case.

-- Decrement operator

operator

Subtracts 1 from the operand.

Syntax

-- lvalue

lvalue --

Description

The decrement operator -- subtracts one from its operand. You can use the decrement operator in either prefix (--operand) or postfix (operand--) form. Prefix form decrements the operand before using its value; postfix form decrements the operand after using its value. For example, if y is equal to 10

x = y--;

sets x equal to 10, then decrements y to 9. In this case,

x = --y;

y is decremented to 9, then the assignment is made, setting x equal to 9 also.

You can use ++ and -- only with variables, not with constants. The type of the result is the same as the type of the lvalue expression.

If the variable is a pointer, C decrements the variable by the size of the object pointed to, not necessarily by 1. For example, if p points to an element in an array, p-- sets p to point to the previous element.

Example

Where counter is 7, in

i = counter--;

And

i = --counter;

In the first statement, i gets set to 7; in the second, i gets set to 8. In both, counter gets set to eight.

/ Division operator

operator

Divide the first operand by the second.

Syntax

expression / expression

Description

The division operator / divides the first operand by the second. / has the same precedence as * and %. C performs the usual arithmetic conversions with this operation.

In the case where one integer is divided by another, C will truncate the fraction part, if any, of the result, before using or assigning the result. To avoid this truncation situation, either

- * make one of the operands a float or a double, or

- * cast one of the expressions to double, as in

((double) a)/b);

You may combine the / operator with the = operator to form an assignment operator, as in

a /= b;

which means: a = a / b.

You can chain division operations together, as in:

a / b / c / d

But in such constructions, you must specify grouping; use parentheses to do so. Otherwise, by default, the division operator, like all multiplicative operators, groups left-to-right.

Related Topics

See also: the modulus operator, %, which gives the remainder only.

== Equality operator

operator

Determines whether the operands are equal or not.

Syntax

expression == expression

Description

The equality operator == yields 1 if the operands are not equal, 0 if they are equal. In other words, it gives the opposite results from the inequality operator != .

== has the same precedence as the !=, which is lower than the relational operators.

When you compare a pointer to an integer with AOS/VS C, the pointer is compared directly to the integer, using an unsigned comparison.

A pointer with the value 0 is guaranteed not to point to any object, and will appear to be equal to 0. In conventional usage, such a pointer is considered to be null.

You can chain equality operations together.

a == b == c

means: compare a to b, produce a 0 or 1, depending on the result, and compare that 0 or 1 to c.

Related Topics

See also: the inequality operator.

> Greater-than operator operator

Determines whether the left operand is greater than the right one or not.

Syntax

expression > expression

Description

The greater-than operator > yields 1 if the left expression is greater than the right, 0 if the the left expression is less than or equal to it. In other words, like the other relational operators, it yields 1 if true and 0 if false.

Greater-than produces an integer result. C performs the usual arithmetic conversions with this operation. > has the same precedence as the other relational operators.

You can compare two pointers. The result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

Example

This expression tests whether or not a is greater than b.

a > b

>= or => Greater-than-or-equal-to operator operator

Determines whether the left operand is greater than or equal to the right operand or not.

Syntax

expression >= expression

Description

The greater-than-or-equal-to operator >= yields 1 if the left expression is greater than or equal to the right, 0 if the left expression is less than it. In other words, like the other relational operators, >= yields 1 if true and 0 if false.

Greater-than-or-equal produces an integer result. C performs the usual arithmetic conversions with this operation. >= has the same precedence as all the other relational operators.

You can compare two pointers. The result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

Example

This expression tests whether or not the variable five is less than the variable six or equal to the variable six.

five >= six

++ Increment operator

operator

Adds 1 to the operand.

Syntax

++ lvalue

lvalue ++

Description

The increment operator ++ adds one to the operand. You may use it in either prefix (++operand) or postfix (operand++) form. Prefix increments the operand before using its value, and postfix increments the operand after using its value. For example, if y equals 10, then the assignment expression

x = y++

sets x equal to ten, then increments y to 11. In the following case,

x = ++y

y gets incremented to 11 first. Then the assignment is made, setting x equal to 11 also.

The type of the result is the same as the type of the lvalue expression.

You can only use ++ and -- with variables, not with constants.

If the variable is a pointer, C increments the variable by the size of the object pointed to, not necessarily by 1. For example, if p points to an element in an array, p++ sets p to point to the next element.

Example

In the following examples, counter starts off at 7. In the first example, i gets set to 7; in the second, i gets set to 8. In both, counter gets set to eight.

i = counter++

i = ++counter

* Indirection operator operator

Gives what the operand points to.

Syntax

*(expression)

Description

The indirection operator * discovers what lies at an address. The operand must be a pointer, not merely an integer.

You can declare an object to be a pointer with the indirection operator:

```
char *j;      /* declare a pointer, named j, to a character. */
```

The indirection operator is useful in searching through arrays, exploring the branches of trees, and permitting call-by-value functions to change said values.

You can nest the indirection operator:

```
**egg          means "what the-object-that-egg-points-to
                points to".
```

You can put large integers into pointers, and with the address operator, use them to explore the memory of your computer; be forewarned that AOS/VS takes a dim view of this and will likely spring an inward address trap.

Example

These examples show how the indirection operator works.

```
char p;        /* declares a character variable named p. */
```

```
char *q;       /* declares a pointer, named q, to a char. */
```

```
&p            means "the address of p".
```

```
q = &p        /* sets q equal to the address of p. In      */
               /* other words, it makes q point to p.   */
```

```
*q            now means "what q points to" (it points to p).
```

```

int thing;          /* declares an integer named thing.      */
int *pthing;        /* declares a pointer named pthing to an integer not specified yet. */
pthing = &thing;    /* sets the integer-pointer pthing to the address of the integer thing. */
int *athing[3];     /* declares an array, named atthing, of three pointers to integers. */
int (*pathing)[];   /* declares a pointer, named pathing, to an array, not specified here, of integers. */
pathing = atthing;  /* sets the array-pointer, pathing, to point to the array atthing. */

int *fthing();      /* declares a function, named fthing, that returns a pointer to an integer. */
int (*pfthing)();   /* declares a pointer, named pfthing, that points to a function, not specified here, that returns an integer. */

```

Related Topics

See also: &, the address operator, and [].

!= Inequality operator

operator

Determines whether the operands are not equal.

Syntax

expression != expression

Description

The inequality operator != yields 1 if the expressions are not equal, 0 if they are equal. In other words, it gives the opposite results from the equality operator ==.

!= has the same precedence as ==.

When you compare a pointer to an integer with AOS/VS C, the pointer is compared directly to the integer, using an unsigned comparison.

You can chain equality operations together.

a != b != c

means: compare a to b, produce a 0 or 1, depending on the result, and compare that 0 or 1 to c.

Related Topics

See also: ==, the equality operator.

< Less-than operator

operator

Determines whether the first operand is less than the second or not.

Syntax

expression < expression

Description

The less-than operator < yields 1 if the left expression is less than the right, 0 if the left expression is greater or equal to it. In other words, like the other relational operators, < yields 1 if true and 0 if false.

Less-than produces integer results. C performs the usual arithmetic conversions with this operation. < has the same precedence as the other relational operators.

As with greater-than, you can compare two pointers. The result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

When you compare a pointer to an integer with AOS/VS C, the pointer is compared directly to the integer, using an unsigned comparison.

You can chain relational operations together.

a < b < c

means: compare a to b, produce a 0 or 1, depending on the result, and compare that 0 or 1 to c.

Example

This expression tests whether or not a is less than b.

a < b

<= or =< Less-than-or-equal-to operator operator

Determines whether the first operand is less than or equal to the second operator or not.

Syntax

expression <= expression

Description

The less-than-or-equal-to operator <= yields 1 if the left expression is less than or equal to the right, 0 if the left expression is greater than it. In other words, like the other relational operators, it yields 1 if true and 0 if false.

Less-than-or-equal-to produces integer results. C performs the usual arithmetic conversions with this operation. <= has the same precedence as the other relational operators.

You can compare two pointers. The result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

When you compare a pointer to an integer with AOS/VS C, the pointer is compared directly to the integer, using an unsigned comparison.

You can chain relational operations together.

a <= b <= c

means: compare a to b, produce a 0 or 1, depending on the result, and compare that 0 or 1 to c.

Example

This expression tests whether or not a is less than b or equal to b.

a <= b

&& Logical AND operator

operator

Performs the logical AND function on the operands.

Syntax

expression && expression

Description

The logical AND operator && returns 1 if both its operands are nonzero, 0 otherwise. The result is always an integer.

Unlike &, && guarantees left-to-right evaluation. Moreover, the second operand is not evaluated if the first operand is 0.

You can chain logical AND operations together, as in:

a && b && c && d

In such a construction, the && operator groups left-to-right; and evaluation won't proceed if any operands prove false.

With &&, the operands need not have the same type, but each must have one of the fundamental types, an enumerated type, or be a pointer. You cannot use a struct or union with &&.

! Logical negation operator (logical NOT) operator

Performs the logical NOT operation on the operands.

Syntax

! expression

Description

The logical negation operator ! returns 1 if the value of its operand is 0, 0 if the value of its operand is nonzero. The type of the result is int. The operands may be of any arithmetic type, including pointers.

Example

Saying

if (!wood)

is a shorter way of saying

if (wood == 0)

Related Topics

See also: the != and == operators.

```
*****
|| Logical OR operator                                operator
Performs the logical OR function on the operands.
*****
```

Syntax

expression || expression

Description

The logical OR operator `||` returns 1 if both its operands are nonzero, 0 otherwise. The result is always an integer.

Unlike `|`, `||` guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand is 1.

You can chain logical OR operations together, as in:

a || b || c || d

In such a construction, the `||` operator groups left-to-right; and as soon as one operand proves true, evaluation stops.

With `||`, the operands need not have the same type, but each must have one of the fundamental types, an enumerated type, or be a pointer. You cannot use a struct or union with `||`.

% Modulus operator

operator

Gives the remainder from the division of the left operand by the right.

Syntax

expression % expression

Description

The modulus operator % yields the remainder when the first operand is divided by the second. The operands must both be integers. C performs the usual arithmetic conversions with this operation. % has the same precedence as * and /.

By contrast, to discard the fraction part of the result and keep the integer part, use integer division; that is, use the division operator, /, with integers for operands.

You can combine the % operator with the = operator to form an assignment operator, as in

a %= b;

which means: a = a % b.

You can chain modulus operations together, as in:

a % b % c % d

In such constructions, the modulus operator groups left-to-right.

Example

To determine if gaul is divisible by three but not four:

```
if (gaul % 3 == 0 && gaul % 4 != 0)
    /* its france */
else
    /* its not */
```

Related Topics

See also: the / (division) operator.

* Multiplication operator operator

Gives the product of the operands.

Syntax

expression * expression

Description

The multiplication operator * yields the product of two operands. The operands must be integral or floating-point. C performs the usual arithmetic conversions with this operation. * has the same precedence as / and %.

You can combine the * operator with the = operator to form an assignment operator, as in

a *= b

which means: a = a * b.

You can chain multiplication operations together, as in:

a * b * c * d

The * operator is associative, which is to say the C compiler may rearrange the order of evaluation.

~ Ones complement operator operator

Gives the ones complement of the operand.

Syntax

~expression

Description

The ones complement operator ~ interprets the operand as a bit pattern; it then converts each 1 into a 0 and vice versa. C performs the usual arithmetic conversions for the ~ operator. The operand must be an integer.

This operator is useful in expressions like

x & ~077

which masks the last six bits of x to 0.

Example

b = ~(~0 << c)

This expression creates a mask in b in which the rightmost c bits are all ones.

<< Shift left operator operator

Shift the left operand left by the number of bit positions given by the right operand.

Syntax

expression << expression

Description

Given the expression shiftee << shifter, C interprets shiftee as a bit pattern and shifts it to the left by shifter bit-positions. C fills the vacated bits with zeros.

The shift operators << and >> group left-to-right. C performs the usual arithmetic conversions with this operation. Both operands to the shift operator must be ints.

If the right operand is negative, the shift operator becomes >> ; and if the absolute value of the right operand is greater than 32, the result is zero. These interpretations are features of the Data General C compiler.

The << operator may be combined with the = operator to form an assignment operator, as in

a <<= b

which means: a = a << b.

The shift operators are useful for bit-twiddling in general.

Related Topics

See also: right-shift, bitwise AND, bitwise inclusive OR, bitwise exclusive OR, and one's complement operators.

>> Shift right operator operator

Shift the left operand right by the number of bit positions given by the right operand.

Syntax

expression >> expression

Description

Given the expression shiftee >> shifter, C interprets shiftee as a bit pattern and shifts it to the right by shifter bit-positions.

The shift operators << and >> group left-to-right. C performs the usual arithmetic conversions with this operation. Both operands to the shift operator must be ints.

AOS/VS C always fills in the high-order bits with zeros.

If the right operand is negative, the operator becomes << ; and if the absolute value of the right operand is greater than 32, the result is zero. These interpretations are features of the Data General C compiler.

You may combine the >> operator with the + operator to form an assignment operator, as in

a >>= b

which means: a = a >> b.

The shift operators are useful for bit-twiddling in general.

Related Topics

See also: left-shift operator, bitwise AND, bitwise inclusive OR, bitwise exclusive OR, and one's complement operators.

sizeof

operator

sizeof is a unary compile-time operator that yields the size, in bytes, of its operand. sizeof is a reserved word in C, and it is described in detail in Chapter 7.

. Structure member operator operator

Takes a member of a structure.

Syntax

primary-lvalue.identifier

Description

You use the structure member operator to fetch a member of a struct. Primary lvalue is the name of an actual structure, and identifier is the formal name of a member of that structure. Both the structure member operator and the structure pointer operator evaluate from left to right.

Examples

In the examples that follow, we'll use these declarations:

```
struct date
{
    int day;
    int month;
    int year;
} birthdate, *pd;

struct personnel
{
    char name[31];
    struct date birthdate;
} mystaff;
```

Since pd is a pointer to a structure, we can use it to fetch the year member of a structure of that type:

(*pd).year

To get the month element from the structure, mystaff:

mystaff.birthdate.month

The grouping of the above expression is:

(mystaff.birthdate).month

```

*****
->  Structure pointer operator                operator
Points to a member of a structure.
*****

```

Syntax

primary-expression -> identifier

Description

The structure pointer operator -> is a convenient shorthand for a pointer to a structure member. Primary-expression is the name of an actual structure and identifier is the name of a member of that structure. Both the structure pointer operator and the structure member operator, . , evaluate from left to right.

Examples

In the examples that follow, we'll use these declarations:

```

struct date
{
    int day;
    int month;
    int year;
} birthdate, *pd;

struct personnel
{
    char name[31];
    struct date birthdate;
} mystaff, *pms;

```

Using the above declarations, C takes these expressions:

```

*pms.birthdate.month
mystaff.birthdate.month

```

and groups them as follows:

```

*((pms.birthdate).month)
(mystaff.birthdate).month

```

Since pd points to a structure, we can refer to the year member of that structure as follows:

```
(*pd).year
```

The -> notation for this same member would be

```
pd->year
```

pd->year is the year member of the structure that pd points to.

The structure operators -> and . are at the top of the precedence hierarchy along with () and []. Using the following declaration:

```
struct {  
    int x;  
    int *y;  
} *p;
```

then

```
++p->x
```

increments x, not p, because the precedence of -> over ++ gives this grouping:

```
++(p->x)
```

To alter this default grouping, using parentheses:

(++p)->y	increments p before accessing y
*p->y++	increments y after accessing whatever it points to (just like *s++)
(*p->y)++	increments whatever y points to
*p++->y	increments p after accessing whatever y points to.

- Subtraction operator operator

Subtracts the right operand from the left.

Syntax

expression - expression

Description

The subtraction operator - yields the difference between the right operand and the left. C performs the usual arithmetic conversions with this operation. - has the same precedence as +.

You can chain subtraction operations together, as in:

a - b - c - d

In such constructions, the order of evaluation is left-to-right.

You can subtract an integer from a pointer. In this case, the same conversions as for addition apply.

If you subtract two pointers to objects of the same type, C converts the result to an integer that represents the number of objects separating the objects of the pointers. C does the conversion by dividing the result of the subtraction by the length of the object. This conversion will in general give unexpected results unless the pointers point to objects in the same array. The unexpected results are because pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

Related Topics

See also: the addition operator.

- Unary minus operator (unary negation operator) operator

Gives the negative of the operand.

Syntax

-(expression)

Description

The unary minus operator - gives the negative of its operand. In other words, it produces the two's complement of the operand. C performs the usual arithmetic conversions with this operation.

C computes the negative of an unsigned quantity by subtracting its value from 2^n , where n is the number of bits in an int.

You can chain unary minus operations together, as in:

-(-normano)

This gives normano, unchanged.

```

*****
, Comma operator                                operator

Groups expressions.
*****

```

Syntax

expression, expression

Description

In a list of expressions, the comma operator produces left-to-right evaluation in the list. Furthermore, all but the last value are discarded: For example, in the following group of expressions:

```
(i++, j += 3, k >= 1)
```

the final value of the parenthesized expression will be either 1 or 0, since `k >= 1` can have only two outcomes: true (1) or false (0).

In general, anywhere an expression can occur, an expression list can occur. However, since the arguments in function calls, constants in the declarations of enums, and declarator lists all use commas, the comma operator reverts to the meanings required by those contexts. For example,

```
f(a, b)
```

calls the function `f` and passes it two arguments;

```
f((a,b))
```

calls the function `f` and passes it one argument, `b` (after evaluating `a`, of course).

Since only the result of the rightmost operand is saved, it follows that the type and value of the result are the type and value of the rightmost operand.

The comma is useful in `for` statements. There it produces left-to-right evaluation when more than one expression appears in one of the three segments of the `for`'s control statement.

End of Chapter

CHAPTER 9

PREPROCESSOR STATEMENTS

Preprocessor Syntax

=====

Preprocessor statements allow you efficient ways to specify

- * macro substitution
- * file inclusion
- * conditional compilation
- * listing
- * cross referencing
- * assertion testing

In some implementations of the C language, preprocessor statements are actually processed before compilation begins; hence the name. In Data General's implementation, the statements are processed at compilation time, though not necessarily first. In either case, the preprocessor statements function as compiler control lines, allowing you to direct the compiler along specific courses of action. In what follows, when we speak of the preprocessor's actions, we are referring to the actions the compiler carries out when it encounters preprocessor statements.

You can restrict the applicability of a preprocessor statement to a segment of a program, or specify its applicability throughout the program. In the latter case, you have the option, for certain statements, of specifying them directly in the compilation command line with local /define and /include switches.

Preprocessor statements begin with a number sign (#) in the first column, and end with a NEW LINE (rather than a semicolon). To write a preprocessor statement longer than one line, place a backslash (\) before each interior NEW LINE. The complete set of preprocessor statement forms is

```
#<NL>
#define    <identifier>    <token-string><NL>
#define    <identifier> "("<identifier-list>)"<token-string><NL>
#undef     <identifier><NL>
#include   "filename"<NL>
          /* C quotation marks around filename this line only */
#include   <filename><NL>
          /* C angle brackets around filename this line only */
#if       CONSTANT-EXP<NL>
#ifdef    <identifier><NL>
#ifndef   <identifier><NL>
#else<NL>
#endif<NL>
#undef    <identifier><NL>
```

```

#[line]    <constant> <identifier><NL>
           /* line is optional */
#list<NL>
#nolist<NL>
#noxref<NL>
#xref<NL>
#assert    CONSTANT-EXP<NL>

```

The number sign (#) must occur in column one of the preprocessor statement.

Macros

=====

You may define substitutions in your source file with C macro definitions. The macro names have the same form as C identifiers. You may redefine the macro for different sections of code. The substitutions are valid from the point of definition either to the end of the source file or to the point of redefinition. Substitution does not take place within quoted strings.

Simple Macros

Simple macros have the form

```
#define <name> <string><NL>
```

where name has the form of an identifier, and string is any syntactically valid string of C code. If your file contains the line

```
#define MAX 1000<NL>
```

then the string 1000 replaces MAX everywhere (except in quoted strings) from the point of definition to the end of the file or the point of redefinition. It is customary to cite macro names in uppercase.

The macro terminates with a NEW LINE, not a semicolon. If you wish to include a NEW LINE within the macro definition, precede it with a \.

If you wish the macro definition to be valid from the beginning of the program, you may specify it in the compilation command line as

```
|      XEQ[/S] CC ... <identifier>/define=<value> ...<NL>
```

where identifier corresponds to the name in #define, and value to the string. You may place the macro definition anywhere in the command line following the global switches, and

you may specify as many definitions as you like. You may define an identifier as itself with

```
XEQ[/S] CC ... <identifier>/define ...<NL>
```

This is helpful if you wish to compile conditionally, for example for debugging.

Macros defined from the command line in the form

```
macro_name/DEFINE
```

as well as the predefined macros AOSVS and aosvs take the value 1 instead of the identifier itself.

Macros with Arguments

You may define macros with arguments, so that the replacement text depends on the way the program calls the macro. Such macros have the form

```
#define <name>(arg1,...,argn) <string><NL>
```

where string contains one or more occurrences of each argument argi. The parenthesis preceding the arguments must follow the macro name immediately. Otherwise, the compiler assumes a macro with no arguments. If an argument occurs more than once in the replacing string, and contains side effects such as assignments or function calls, it will be evaluated more than once, which you may not want. You cannot specify macros with arguments from the compilation command line. The string may contain any number of NEW LINES, as long as a backslash (\) precedes each one.

The preprocessor substitutes arguments to macro expansions for the argument identifier inside strings. For example,

```
#define PRINT( fmt, expr )      printf("expr = %fmt\n", expr)

main()
{
    int i = 1;
    PRINT( d, i );
}
```

would print out

```
i = 1
```

(The preprocessor does not substitute macros within strings.)

The preprocessor processes macros inside other macros at the time of macro expansion, rather than at the time of macro definition. For example, the fragment

```
#define bar    1
#define foo    bar
#define bar    2

    int    a = foo;
```

assigns 2 to a rather than 1. You may nest macros in this way indefinitely. When expanding a macro, the preprocessor uses the previous definition of the macro used within itself to avoid hanging. This allows you to specify a macro like

```
extern int    fwrite();

#define fwrite(s,b,c)    fwrite(s, (char *)b, c)
```

which casts the second argument to fwrite to be a pointer to a character.

For an example of a macro with arguments,

```
#define NOR(P, Q) (!P && !Q)
```

lets you simulate a logical NOR operator. Following this definition, the line

```
NOR(( a < b), (b < c)) ? s++ : s--;
```

resolves to

```
(!(a<b) && !(b<c)) ? s++ : s--;
```

and s increments only if neither a<b nor b<c, and decrements otherwise.

Invalidating Macros

You may limit the scope of a #define by specifying

```
#undef <identifier><NL>
```

| C macros stack. If you redefine a macro name, a subsequent #undef
| statement restores the previous definition. If there is no previous
| definition, the identifier is no longer valid.

Include Files

=====

You may include the contents of other files in the file you wish to compile. Your source file may contain either the line

```
#include "<NL>filename"
```

or the line

```
#include <filename><NL>
```

where filename is the file you wish to include in the compilation.

If you use the form with double quotation marks, the compiler uses the current searchlist to look for filename. If you use the form with angle brackets, and you also use the compilation switch /INCDIRECTORY=<dir-name>, the compiler first searches in the specified directory for the file. If it does not find the file there, it attempts to open the file from the current directory using the current searchlist.

If you use the /L[=filename] switch, you can use the /INCLUDES switch to specify the listing of include files on the listing file switch, or you can block the listing of include files on the listing file with the /NOINCLUDES switch. (The default value for this pair of switches is /INCLUDES.)

The contents of filename are valid only from the point in the program at which you specify the inclusion. If you wish the contents to be valid throughout the program, specify the inclusion at the beginning of the source file. Include files can contain other include files to a nesting limit of seven.

Alternatively, if you wish the contents to be valid throughout the program, you may specify the inclusion with a local switch in the compilation command line.

```
XEQ[/S] CC ... filename/include ...<NL>
```

You may specify up to seven include files in the compilation command line. The include filenames may occur anywhere after the global switches.

Conditional Compilation

=====

You may test a condition and specify compilation depending upon the result of the test with three different hypothetical preprocessor statements.

Test an Expression

The compiler evaluates the constant-exp following the #if. If the expression is true, it returns a nonzero value, and the lines following

```
#if <constant-exp><NL>
```

compile. If the expression is false, it returns a value of 0, and the compiler skips the following code.

Test for a Definition

When the compiler encounters

```
#ifdef <identifier><NL>
```

it checks to see whether you stipulated the identifier previously with a #define statement or a /define compilation switch (and have not undefined it). If the identifier is valid, compilation continues following the #ifdef statement. If it is not, the compiler skips the code following the statement.

When the compiler encounters

```
#ifndef <identifier><NL>
```

it continues compilation only if the identifier is not valid: either you did not stipulate the identifier with a #define statement or a /define compilation switch, or else you have invalidated the identifier with an #undef statement. If the identifier is currently valid, the compiler skips the following code.

Specify an Alternative

You may wish to specify an alternative section of code in case the #if, #ifdef, or #ifndef conditions fail. Enter an

```
#else<NL>
```

before the alternative code. The lines following it compile if and only if the lines between the #if, #ifdef, or #ifndef statements and the #else statement do not compile.

End a Conditional Compilation

Whichever of #if, #ifdef, or #ifndef you use, and regardless of whether you used an #else, you must use

#endif<NL>

to specify the end of the conditional compilation.

You may nest these conditional preprocessor statements within each other.

Listing Control

Three devices allow you to control the amount and kinds of listing the compiler gives you.

Line Control

For error diagnostics, a statement of the form

#line <constant> [<identifier>]<NL>

or

<constant> [<identifier>]<NL>

makes the compiler take the constant as the line number of the next source line, and the identifier as the name of the current input file. The current input file does not change if you omit the identifier.

Suspension of Listing

You can suspend the listing of following lines with

#nolist<NL>

and begin listing again with

#list<NL>

For example, the lines

```
#nolist<NL>
#include      <stdio.h><NL>
#include      <math.h><NL>
#include      <dglib.h><NL>
#list<NL>
```

at the beginning of a program module allow you to suppress the unwieldy listing of standard libraries in your listing file.

Cross Referencing

If you use the /XREF switch in the compilation line, you generate a cross reference. If you use a /L[=listfile] switch, the compiler appends an alphabetical cross-reference of all program entities to the listing file, with line numbers, including #define macros.

Cross referencing in C

- * omits references to unused identifiers (in code not compiled through conditional compilation)
- * specifies distinct entries for separate variables with the same name in different functions
- * cross references macros, even within other macros
- * pinpoints which structure a member belongs to
- * gives the type of each identifier, and the function it is declared in

For example, if you compile the program cross.c

```
        /* Demonstrate the cross reference facility */
        /* ident has many different uses below */

int     ident;

struct  foo    {
        char    ident;          /* use as a structure member */
        int     other :8;
} st1, st2;

func_1(){
        double  ident;          /* different ident in scope */

        ident = 2;
        return( 2 );
}

func_2(){
        ident = 1;
        st1.ident = func_1();
}

#define ident  3                /* use in macro */

int     a = ident;
```

with the /XREF switch, you generate the cross reference

Source file: xref.C

Compiled on 8-Jul-82 at 2:44:51 by AOS/VS C Rev mm.nn.oo.ppp

Cross Reference Listing

a	extern int 27*
foo	struct 5* 8 8 stl 8* 21=
func_1	extern int () 11* 21
func_2	extern int () 19*
ident	#define ident 3 25* 27
ident	extern int 3* 20=
ident	{member of struct foo} char 6* stl.ident 8* 21=
ident	auto double Function func_1 12* 14=
other	{member of struct foo} unsigned : 8 7*
stl	extern struct foo 8* 21=
st2	extern struct foo 8*

Context Flags:

*	Declare or #Define
none	Use
=	Set
?	Set or Use
^	Dynamic Reference
U	#Undef of Macro
I	#Ifdef/#Ifndef Use

Number of Symbols:	11
Number of Cross References:	34

You can suspend cross referencing in your program with

```
#nohref<NL>
```

and resume it with

```
#xref<NL>
```

If you do not specify the /XREF switch in the compilation line, these preprocessor statements have no effect.

Testing Assertions

=====

You can specify a test for the truth or falsity of an expression with the statement

```
#assert CONSTANT-EXP<NL>
```

| If the expression evaluates to true (non-zero) at compilation time,
| the compilation continues without generating a level 3 error. If the
| expression evaluates to false (0), a severity 3 error is generated.

The statement

```
#assert sizeof( datum1 ) > sizeof( datum2 )
```

returns an error unless datum1 is larger than datum2.

End of Chapter

CHAPTER 10

THE RUNTIME ENVIRONMENT

Data General's implementation of the C programming language provides you with a wide variety of runtime functions and macros. As with all functions, you must declare the types of the runtime functions you use. The default type is `int`.

C runtime functions are truly functions in that they evaluate to a determinate value for their type. At execution time, when a function is called, copies of the values are passed to it. The values in the calling function do not change unless you explicitly provide for them to with assignment and return statements, or pass a pointer to the function that it can use to modify the value. You have the option of not returning anything, in which case function result is indeterminate. To preclude returning anything, you may specify functions to be of the void type; these are essentially like procedures or subroutines of other languages.

Data General's implementation of the C language shares a common runtime environment with other high-level 32-bit languages. Exercising some care, you can, for instance, call a function written in PL/1 from a C program, and vice versa.

The compiler does not check to see whether you have furnished too many or too few arguments to a function, or whether the arguments are of the correct type.

By convention, library function names are lowercase. You may declare your own function in place of a standard function with the same identifier, as long as it is not of the `$builtin` storage class. The compiler will use your definition instead of the library definition. Data General assumes no responsibility for incorrect results if you substitute your own module for a library module.

The following sections summarize the classes of available standard functions. For fuller descriptions, see the entry for each in the Library that concludes this chapter.

Arithmetic Functions

=====

Arithmetic functions get absolute values (`abs`, `fabs`), powers and roots (`exp`, `hypot`, `ldexp`, `pow`, `scale10`, `sqrt`), maxima and minima (`ceil`, `floor`, `fmax`, `fmin`, `max`, `min`), modulus values (`fmod`, `modf`), and logarithms (`log`, `log10`, `frexp`). They include trigonometric (`acos`, `asin`, `atan`, `atan2`, `cos`, `sin`, `tan`) and hyperbolic

trigonometric (cosh, sinh, tanh) functions. They provide for pseudorandomization (rand, randl, srand).

Conversion Functions

=====

The conversion functions allow you to map from character strings to various numeric types (atof, atoi, atol, atou), from various numeric types to character strings (ftoa, itoa, otoa, utoa), to all lowercase (tolower, _tolower), and to all uppercase (toupper, _toupper).

File-Handling Functions

=====

The file-handling functions allow for opening files with various options, including block, binary, and shared page I/O (fopen, freopen), and closing, renaming, and deleting files (fclose, rename, delete). You can get the channel number of an open file (channel, fchannel), get (ftell) and set (fseek, lseek) a pointer in a file, and rewind (rew) a file. You can assign a buffer (setbuf) to a file, create a temporary file (tmpfile), and return a pointer to a unique temporary filename (tmpnam).

Input/Output Functions

=====

The macro getc reads the next character from a file, and putc writes the next character to a file. Since these are macros, you may not pass them as parameters to a function or take their addresses. For those purposes, use the fgetc and fputc functions.

The fgets function reads a string from a specified file, and fputs writes a string to a specified file.

The function gets reads a string from the standard input file (usually the terminal), and puts writes a string to the standard output file (also usually the terminal).

The getchar macro reads a character from the standard input file, and putchar writes a character to the standard output file.

The getw function reads an integer value (four bytes in the 32-bit environment) from a specified file, and putw writes an integer value to a specified file.

For doing block I/O to binary files, you can use fread to read in a specified number of bytes from a specified number of records in a specified file, and fwrite to write out a specified number of bytes to a specified number of records on a specified file.

You can specify three versions of formatted input and output. The fscanf function reads in data in a specified format from a specified file, sscanf reads in data in a specified format from an internal string, and scanf reads in data in a specified format from the standard input file (usually the terminal). The fprintf function writes out data in a specified format to a specified file, sprintf writes data in a specified format to an internal string, and printf writes out data in a specified format to the standard output file (usually the terminal).

The ungetc function returns a single character to the input file.

The ferror function returns a nonzero if an error occurred during input/output. The fflush function flushes the output buffer of the specified stream.

The exit function flushes all buffers and terminates the process normally.

Memory Management Functions

=====

The alloc and malloc functions allocate one block of memory without zeroing the memory, alloca allocates more memory from the user's stack, and calloc allocates a number of blocks and zeros them out.

You can change the size of a previously allocated area with realloc. You can zero out a section of memory with the zero function. You can move words from one location in memory to another with the blockmove function.

You can allocate an area from the heap large enough to hold a string, move the string into the area, and return a pointer to the area with the strsave function. You can specify a maximum length for the string with the strnsave function.

You can release memory allocated through alloc with the free function.

Miscellaneous Functions

=====

You can provide for a nonlocal flow of control with the setjmp function and call for its execution with the longjmp function.

The main function must appear somewhere in your program, since that is where program execution begins. In the AOS/VS environment, you can furnish arguments to the main function

from the CLI at execution time. (In the AOS/RT32 environment, arguments are ignored.)

The nargs function returns the number of arguments furnished to the executing function. The saverror function stores an error code in a static area for later use. The wdleng function returns the integer value 32, the word length in bits of the type int.

You can set screen characteristics with the setscrn function, and set the cursor position for the next screen read with the setxy function.

You can delay a task for a given number of seconds with the sleep function.

Program Environment Functions

=====

You can execute a CLI command from a C program with the system function. You can make an AOS/VS system call from a C program with either a specific sys_<system-call-mnemonic> function or the general sys function.

The curdir function stores the name of the current directory, getdir returns a pointer to the pathname of the current working directory, createdir creates a directory or control point directory, and dir changes the working directory. The pathname function returns a pointer to the complete pathname of a file or directory. The mktemp function returns a complete unused temporary filename.

The getlogin function returns a pointer to a static buffer containing the login name of the calling process. The getfp function returns the frame pointer of the father's environment. The setfp function sets the wide frame pointer (WFP).

The getsb function returns the value stored at the wide stack base (WSB), gets1 returns the value of the wide stack limit (WSL), and getsp returns the value of the wide stack pointer (WSP). The setsb function sets the WSB, sets1 sets the WSL, and setsp sets the WSP.

The isatty function returns a 1 if the I/O file is using a terminal for I/O, intso returns a 1 if the standard output file is a terminal, and intss returns a 1 if the standard input file is a terminal.

The lasterror function returns the code for the most recent error, and perror prints the most recent error message. The abort function immediately aborts the user program with the system error message corresponding to the provided runtime error code.

The `dg_ctime` function returns the current time, and `dg_localtime` returns a pointer to a nine-word array for the current time.

Queue-Handling Functions

The `qsearch` function allows you to search a queue for an element meeting a specified condition. You can place an element in the queue with the `insert_after`, `insert_before`, `insert_head`, or `insert_tail` functions. You can delete an element from the queue with the `remove_element`, `remove_head`, or `remove_tail` functions.

Character-Handling Macros

The character-handling macros in the table below check single characters, and return a nonzero if the character fits the corresponding class.

Macro	Checks for the character class
<code>isalnum</code>	alphanumeric
<code>isalphanum</code>	alphanumeric
<code>isalpha</code>	alphabetic
<code>isascii</code>	ASCII character or EOF
<code>isctrl</code>	control character
<code>isdigit</code>	ASCII representation of (0 through 9)
<code>ishex</code>	hexadecimal
<code>isxdigit</code>	hexadecimal
<code>isline</code>	line terminator
<code>islower</code>	lowercase
<code>ispunct</code>	punctuator
<code>isspace</code>	space character
<code>isupper</code>	uppercase

Since these are macros, you cannot pass them as parameters.

String-Handling Functions

String-handling functions compare or manipulate character arrays, or strings. The `repeat` function replicates a string a given number of times within a character array. The `any` function checks to see whether a specific character occurs anywhere within a string. The `verify` function determines whether one string contains a character not in another. The `strpbrk` returns a pointer to the first character in both strings. The `equal` function

determines whether two strings are equal. The tail function returns a pointer to the null byte within a string. The swab function swaps successive bytes in a string pairwise. The zeropad function replaces leading spaces in a string with zeros.

The strchr function searches for the first, and strrchr for the last occurrence of a character in a string. The strlen function returns the length of a character string.

The remaining eight functions are paired. For the longer version, containing an additional n, you must stipulate maximum string length value:

Function -----	Operates to -----
sign(n)cmp	compare successive (signed) character values in two strings
str(n)cat	concatenate one ASCII string onto the tail of another
str(n)cmp	compare successive (unsigned) character values in two strings
str(n)cpy	copy one string onto another

Standard \$builtin Functions

=====

In the AOS/VS and AOS/RT32 environments, you can generate code for some of the library functions directly on line. This allows you to generate a more efficient call or faster code, at times eliminating the function call and return overhead.

When you wish to use the \$builtin variants of these functions, you must use the \$builtin storage class declaration. The library lists the alternate calling sequences for these functions. If you declare two functions of the same name, one of them extern and the other \$builtin, the compiler does not signal an inconsistency. It uses the \$builtin version in either case.

Standard Include Files

=====

Data General's implementation of the C programming language provides you with several standard include files, which define standard functions and macros.

<u>Include file</u>	<u>Defines</u>
assert.h	runtime assertions
cerr.h	error codes used by runtimes
ctype.h	character macros
dglb.h	AOS/VS routines
sys:dir.h	include file for UNIX emulator
math.h	arithmetic functions
packets	an include directory containing various system call and INFOS II packets
packets:block_io.h	packets used in ?ALLOCATE, ?GOPEN, ?GTRUNC, ?PMPTF, ?PRDB, ?PWRB, ?RDB, ?SPAGE, and ?SOPPF system calls
packets:common.h	used by other include files
packets:create.h	?CREATE system call packets
packets:exec.h	?EXEC system call packets
packets:fdp.h	INFOS II file definition packet
packets:fdp_bit.h	INFOS II file definition packet using bit fields
packets:filestatus.h	packets used in ?STAT system call
packets:histogram.h	packets used in ?WHIST system call
packets:idef.h	packets used in ?IDEF system call
packets:infos.h	includes all INFOS II packets
packets:infos_defines.h	various INFOS II #defines
packets:ipc.h	packets used in ?IREC, ?IS.R, and ?ISEND system calls
packets:kdp.h	INFOS II key definition packet
packets:kdp_bit.h	INFOS II key definition packet using bit fields

packets:lsp.h	INFOS II link subindex packet
packets:lsp_bit.h	INFOS II link subindex packet using bit fields
packets:misc.h	packets used in ?LABEL, ?MBFC, ?MBFU, ?MBTC, ?MTU, and ?SINFO system calls
packets:normal_io.h	packets used in ?CLOSE, ?GPOS, ?OPEN, ?READ, ?SPOS, ?TRUNCATE, and ?WRITE system calls
packets:pp.h	INFOS II processing packet
packets:pp_bit.h	INFOS II procesing packet using bit fields
packets:process.h	packets used in ?GTMS, ?PROC, ?PSTAT, and ?RUNTM system calls
packets:sdp.h	INFOS II subindex definition packet
packets:sdp_bit.h	INFOS II subindex definition packet using bit fields
packets:subindex.h	included by fdp.h and sdp.h to define subindex flags
packets:subindex_bit.h	included by fdp.h and sdp.h to define bit fields for subindexes
packets:synch.h	packets used in binary synchronous communication system calls
packets:task.h	packets used in ?TASK amd ?UIDSTAT system calls
packets:time.h	used by other include files
packets:vdh.h	INFOS II volume definition packet
parlang_rt.h	lang_rt error codes
paru.h	AOS/VS system parameters
queue.h	declarations for queue routines
setjmp.h	declarations for setjmp and longjmp
sgtty.h	include file for UNIX emulator

signal.h	include file for UNIX emulator
sys:stat.h	include file for UNIX emulator
stdio.h	file handling functions, standard input, output, and error files
sysid.h	AOS/VS system call numbers
sys_calls.h	definitions for sys_call_<system-call-mnemonic>()
time.h	time runtimes
sys:timeb.h	include file for UNIX emulator
times.h	include file for UNIX emulator

You may rule out certain options with macro definitions, specified with a #define in the source code or a /d on the compilation line.

Define the macro -----	In order to -----
_NOASSERT	disable checking for system call packet lengths
_NOBIT	preclude the structures with the bit field definitions (for use with infos.h)
_NOMASK	preclude the structures that assume the use of masking to get at various bit fields (for use with infos.h)
_NEXPAND	prevent expansion of these macros in line, replacing them with function calls to fgetc or fputc
	getc putchar putc putchar

This debugging tool allows you to put breakpoints at all user calls to get or write single characters with the assembly language debugger. Define _NEXPAND before the include file stdio.h.

Library of Standard Functions and Macros

=====

In this library of functions and macros, each entry contains

- * the function's name
- * a brief description of what it does
- * calling sequences, with any necessary includes and declarations
- * one or more examples

and may contain further information, such as

- * a more detailed explanation of the function
- * the library that defines the function
- * error and normal returns
- * related functions

Table 10-1, "Standard Functions and Macros," contains a brief description of each function or macro. For a more detailed description, consult the individual entry in the library.

Table 10-1. Standard Functions and Macros

NAME	DESCRIPTION
abort	Immediately aborts the user program.
abs	Takes the absolute value of the integer passed to it.
acos	Returns the arc cosine of the number passed to it.
alloc	Allocates and zeros out one block of memory.
alloca	Allocates more memory from the stack.
any	Returns 1 if a string contains a character and 0 otherwise.
asin	Returns the arc sine of the number passed to it.
assert	Provides runtime assertions.
atan	Returns the arc tangent of the number passed to it.
atan2	Returns the arc tangent of the two numbers passed to it.
atof	Converts an ASCII string into a floating-point number.
atoi	Converts an ascii string to type int.
atol	Converts an ascii string to type long.
atou	Converts an ASCII character string to an unsigned integer.
blockmove	Moves bytes from one location in memory to another.
calloc	Allocates blocks of memory and zeros them out.
ceil	Returns the ceiling of the number passed to it.
channel	Returns the AOS/VS channel number for a UNIX file number.

(continues)

Table 10-1. Standard Functions and Macros

NAME	DESCRIPTION
cos	Returns the cosine of the number passed to it.
cosh	Returns the hyperbolic cosine of the type passed to it.
createdir	Creates a directory or control point directory.
curdir	Stores the name of the current directory.
dg_ctime	Returns the current time.
dg_localtime	Returns a pointer to a nine-word array representing current time.
dir	Changes the working directory.
equal	Determines whether two strings are equal.
exit	Flushes buffers and terminates the process normally.
exp	Returns the constant e raised to the power of the number passed to it.
fabs	Returns the absolute value of the number passed to it.
fchannel	Returns the AOS/VS channel for a file opened with fopen.
fclose	Closes a specified file.
fdopen	Associates a buffered I/O stream with a UNIX file descriptor.
feof	Returns a nonzero when it encounters an end of file.
ferror	Returns a nonzero if an error occurred during I/O.
fflush	Flushes the output buffer of the specified stream.
fgetc	Gets a character from a general file.
fgets	Reads a string from a specified file.

(continued)

Table 10-1. Standard Functions and Macros

NAME	DESCRIPTION
fileno	Returns the UNIX emulated file number of the specified open file.
floor	Returns the floor of the number passed to it.
fmax	Returns the floating maximum of two numbers.
fmin	Returns the floating minimum of two numbers.
fmod	Returns the remainder of the first argument divided by the second argument.
fopen	Opens a file.
fprintf	Writes to a file.
fputc	Puts a character out to the specified stream.
fputs	Writes a string to a specified file.
fread	Reads n records of m bytes each from a specified file.
free	Releases memory allocated through alloc, calloc, malloc.
freopen	Closes a channel and reopens a file or device.
frexp	Returns a double mantissa $x \leq 1$ and stores an integer n indirectly through a pointer where the original value is $x*2^n$.
fscanf	Reads from a file.
fseek	Positions the file to a specified byte.
ftell	Returns the current position in a file.
ftoa	Converts a floating-point number to a string.
fwrite	Writes n records of m bytes each to a specified file.
getc	Reads a single character from a file.

(continued)

Table 10-1. Standard Functions and Macros

NAME	DESCRIPTION
getchar	Reads a single character from the standard input.
getdir	Returns a byte pointer to the pathname of the current working directory.
getenv	Get a pointer to environment variables.
getfp	Returns the frame pointer of the father's environment.
getlogin	Returns a byte pointer to the username.
gets	Reads from the standard input file until it encounters a NEW LINE.
getsb	Retrieves the value stored at the WSB (wide stack base).
getsl	Retrieves the value of the current WSL (wide stack limit).
getsp	Retrieves the value of the WSP (wide stack pointer).
getw	Gets an integer value from the input buffer.
hypot	Finds the hypotenuse of a right triangle.
infos	Make an INFOS II request for the user.
insert_after	Inserts an element into a queue after another queue element.
insert_before	Inserts an element into a queue before another queue element.
insert_head	Inserts an element at the head of a queue.
insert_tail	Inserts an element at the tail of a queue.
intso	Returns 1 if the standard output file is a terminal.
intss	Returns a 1 if the standard input file is a terminal.

(continued)

Table 10-1. Standard Functions and Macros

NAME	DESCRIPTION
isalnum	Returns nonzero if character is alphanumeric.
isalpha	Returns nonzero if character is alphabetic.
isalphanum	Returns nonzero if character is alphanumeric.
isascii	Returns nonzero if argument is an ASCII character or EOF.
isatty	Returns a 1 if the I/O file is using a terminal for I/O.
iscntrl	Returns nonzero if argument is a control character.
isdigit	Returns nonzero if character is the ASCII representation of (0 ... 9).
ishex	Results in nonzero if character is hexadecimal.
isline	Returns nonzero if character is a line terminator.
islower	Returns nonzero if character is lowercase.
ispunct	Returns nonzero if character is a punctuation mark.
isspace	Returns nonzero if the argument is a space character.
isupper	Returns nonzero if character is uppercase.
isxdigit	Results in nonzero if character is hexadecimal.
itoa	Converts an integer to a character string.
lasterror	Returns the code for the most recent error.
ldexp	Returns a value times the nth power of 2.
log	Returns the natural log of the number passed to it.
log10	Returns the log based 10 of the number passed to it.

(continued)

Table 10-1. Standard Functions and Macros

NAME	DESCRIPTION
longjmp	Performs a nonlocal jump in the flow of control.
main	Enters a C program.
malloc	Allocates blocks of memory, but does not zero them out.
max	Returns the integer maximum of two numbers.
min	Returns the integer minimum of two numbers.
mktemp	Returns a complete unused temporary filename.
modf	Returns the fractional part of a value as a double quantity, storing the integer part indirectly.
nargs	Returns the number of arguments for the executing routine.
otoa	Converts an integer into an octal character string.
pathname	Returns a pointer to the complete pathname of a file or directory.
perror	Prints a string and the last error message on the error file.
pow	Returns the value of the first argument raised to the power of the second argument.
printf	Writes to standard output.
putc	Writes a character to a given file.
putchar	Writes a character to the standard output.
puts	Writes to the standard output file until it encounters a NULL byte.
putw	Puts an integer value into the output buffer.

(continued)

Table 10-1. Standard Functions and Macros

NAME	DESCRIPTION
qsearch	Searches for an element in a queue meeting a given condition.
rand	Creates and returns a random number in the range 0 to 32767.
randl	Creates and returns a random number in the range 0 to 1.
realloc	Changes the size of a previously allocated area.
remove_element	Deletes an element from a queue.
remove_head	Deletes an element from the head of a queue.
remove_tail	Deletes an element from the tail of a queue.
repeat	Replicates a string n times within a character array.
rew	Rewinds a file to the beginning.
scale10	Multiplies a number by a power of 10.
scanf	Reads from standard input.
setbuf	Assigns a buffer to a specified file.
setfp	Sets the WFP (wide frame pointer).
setjmp	Provides for nonlocal flow of control.
setsb	Sets the WSB (wide stack base).
setscrn	Sets screen characteristics.
setsl	Sets the WSL (wide stack limit).
setsp	Sets the WSP (wide stack pointer).
setxy	Sets the cursor position for the next screen read.

(continued)

Table 10-1. Standard Functions and Macros

NAME	DESCRIPTION
signcmp	Compares successive character values of two strings.
signncmp	Compares successive character values of two strings.
sin	Returns the sine of the number passed to it.
sinh	Returns the hyperbolic sine of the number passed to it.
sleep	Delays a task for a given number of seconds.
sprintf	Writes to an internal string.
sqrt	Returns the square root of the number passed to it.
srand	Places a user-specified seed in both rand and randl.
sscanf	Reads from an internal string.
strcat	Concatenates one ASCII string onto the tail of another.
strchr	Searches for first occurrence of a character in a string.
strcmp	Compares successive character values of two strings.
strcpy	Copies one string onto another.
strlen	Determines the length of a character string.
strncat	Concatenates one ASCII string onto the tail of another.
strncmp	Compares successive character values of two strings.
strncpy	Copies one string onto another.

(continued)

Table 10-1. Standard Functions and Macros

NAME	DESCRIPTION
strnsave	Allocates an area long enough to hold a string from the heap, moves the string into the area, and returns the pointer to the area.
strpbrk	Returns a pointer to the first character in both strings.
strrchr	Searches for the last occurrence of a character in a string.
strsave	Allocates an area long enough to hold the string from the heap, moves the string into the area, and returns a pointer to the area.
swab	Swaps successive bytes pairwise.
sys	Provides an interface to the operating system.
system	Executes a CLI command.
sys_<system-call-mnemonic>	Makes an AOS/VS system call from a C program.
tail	Returns a pointer to the null byte within a string.
tan	Returns the tangent of the number passed to it.
tanh	Returns the hyperbolic tangent of the number passed to it.
tmpfile	Creates and opens a temporary file.
tmpnam	Returns a pointer to a temporary filename.
tolower	Maps uppercase alphabetic to lowercase alphabetic.
_tolower	Maps uppercase alphabetic to lowercase alphabetic.
toupper	Maps lowercase alphabetic to uppercase alphabetic.
_toupper	Maps lowercase alphabetic to uppercase alphabetic.

(continued)

Table 10-1. Standard Functions and Macros

NAME	DESCRIPTION
traceback	Prints a stack traceback and continues.
ttyname	Returns a byte pointer to the name of a file.
ungetc	Returns a character to the input buffer.
unlink	Deletes a file.
utoa	Converts an unsigned integer to a character string.
verify	Determines whether one string contains a character not in the other.
wdleng	Returns the integer value 32, the word length in bits of the type int.
zero	Zeroes a section of memory.
zeropad	Substitutes zeros for leading spaces.

(concluded)

abort function

Immediately aborts the user program.

Calling Sequence

```
abort();
```

Returns

There is no return for this function.

The abort routine prints a stack traceback on the error file, and then generates a breakfile and aborts.

Related Functions

See also the exit function.

Example

```
/* program test for the abort() function */  
  
#include <stdio.h>  
  
main(argc, argv)  
{  
    int    argc;  
    char   *argv[];  
  
    if ( argc > 1 )  
    {  
        printf("Number of arguments to program is %d.\n", argc-1);  
        printf("Preparing for normal exit...\n");  
    }  
    else  
    {  
        printf("Preparing to abort...\n");  
        abort();  
    }  
}
```

If you call the program test with three arguments A, B, C, the output is

```
Number of arguments to program is 3.  
Preparing for normal exit...
```

and if you call it with no arguments, you receive the message

```
Preparing to abort...
```

```
ERROR          74438.  
from $traceback in task 1 (unique task id is 1).
```

```
Call Traceback:
```

```
from fp=16000014356, pc=.sig.core+14  
from fp=16000014340, pc=$interrupt+15  
from fp=16000014322, pc=abort+11  
from fp=16000014272, pc=main+45  
from fp=16000014150, pc=.MAIN.+73, C initializer  
from fp=          0, pc=I.INIT+46
```

```
Signal SIGIOT (abort routine) aborted program with breakfile.
```

abs \$builtin function

Takes the absolute value of the integer passed to it.

Calling Sequence

```
int      m, n, abs();  
...
```

or

```
$builtin int abs();  
int      m, n;  
...  
m = abs(n);
```

where

m and n are integer variables

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

The include file math.h defines this function, and uses the \$builtin version.

Returns

This function does not check for an integer.

Related Functions

See the fabs function.

Example

```
/* program test for the abs() function */

#include <stdio.h>
#include <math.h>

int      atoi(), m, n, abs();

main(argc, argv)

    int      argc;
    char      *argv[];
{
    n = atoi( argv[1] );
    printf("n = %d has absolute value m = %d.\n", n, m = abs( n ));
}
```

If you call the program test with an argument of -13, the output is

n = -13 has absolute value m = 13.

```
*****
acos                                     $builtin function

Returns the arc cosine of the number passed to it.
*****
```

Calling Sequence

```
double m, n, acos();
...
or

$builtin double acos();
double m, n;
...
m = acos(n);
```

where

m and n are of type double and $-1. \leq n \leq 1$.

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

The include file math.h defines this function, and uses the \$builtin version.

| Returns

| If this routine gets an illegal input, it returns 0.0
| and sets errno to EDOM (defined in errno.h).

Related Functions

See also the cos, cosh functions.

Example

```
-----  
        /* program test for the acos() function */  
  
#include <stdio.h>  
#include <math.h>  
  
double  atof(), m, n, acos();  
  
main(argc, argv)  
  
        int      argc;  
        char     *argv[];  
  
    {  
  
        n = atof( argv[1] );  
        printf("n = %f has the arc cosine m = %f.\n", n, m = acos( n ));  
  
    }
```

If you call the program test with an argument of .666666,
the output is

```
n = 0.666666 has the arc cosine m = 0.841069.
```

alloc function

Allocates and zeros out one block of memory.

Calling Sequence

```
char    *ptr;  
char    *alloc();  
int     size;  
...  
ptr = alloc(size);
```

where

size is the number of bytes in the block

Since hardware representations for pointers to characters differ from pointers to other types, use the type-change operation to convert the result to a pointer of the appropriate type, such as

```
ip = (int *) alloc(n, sizeof(int));
```

The include file `stdio.h` defines this function.

Returns

The `alloc` function returns 0 if no core was allocated.

Related Functions

See also the `alloca`, `calloc`, `malloc`, `realloc` functions.

Example

```
/* program test for the alloc() function */

#include <stdio.h>

char    *ptr, *alloc();
int      size;

main(argc, argv)

        int      argc;
        char     *argv[];
{
    size = atoi( argv[1] );
    ptr = alloc( size );
    printf("size = %d bytes in the block %o.\n", size, ptr );
}
```

A call to the program test with an argument of 1024
generates the output

size = 1024 bytes in the block 34003711770.

(The address returned will probably be different.)

alloca \$builtin function

Allocates more memory from the stack.

Calling Sequence

```
char    *ptr, *alloca();
int      size;
...

or

$builtin char    *alloca();
char    *ptr;
int      size;
...
ptr = alloca( size );
```

where

size is the number of bytes to allocate from the stack

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

Unlike other allocation functions, this one takes memory directly from the caller's stack. The memory is implicitly returned to the stack when the routine that called alloca returns to its caller.

Since hardware representations for pointers to characters differ from those for pointers to other types, use the type change operation to convert the pointer to the correct type.

The include file `dglib.h` defines this function, and uses the \$builtin version.

Related Functions

See also `alloc`, `calloc`, `malloc`, `realloc`.

Example

```
/* program test for the alloca() function */

#include <stdio.h>

char    *ptr, *alloca();
int     size;

main(argc, argv)

    int     argc;
    char    *argv[];
{
    size = atoi( argv[1] );
    ptr = alloca( size );
    printf("size = %d bytes in the block %o.\n", size, ptr );
}
```

A call to the program test with an argument of 512 generates the output

```
size = 512 bytes in the block 34000020424.
```

(The address will differ depending on the amount of stack used and extern/static data.)

any function

Returns 1 if a string contains a character and 0 otherwise.

Calling Sequence

```
char    c, *str;
int      res, any();
...
res = any(c, str);
```

Returns

The function returns a 1 if the character occurs in the string.
It returns a 0 if the character does not occur in the string.

Related Functions

See also the equal, verify, strpbrk, signcmp, strcmp functions.

Example

```
/* program test for the any() function */

#include <stdio.h>

char    c, *string;
int      result, any();

main(argc, argv)

    int    argc;
    char   *argv[];
{
    result = any( argv[1][0], argv[2] );
    printf("The string:\n\t%s\n%s the character %s.\n",
        argv[2], result==1?"contains":"does not contain", argv[1] );
}
```

If you call the program test with arguments y and cathode_ray_tube
the output is

The string:
cathode_ray_tube
contains the character y.

and if you call it with arguments n and cathode_ray_tube the
output is

The string:
cathode_ray_tube
does not contain the character n.

```
*****
asin                               $builtin function
```

```
Returns the arc sine of the number passed to it.
*****
```

Calling Sequence

```
double m, n, asin();
...
or

$builtin double asin();
double m, n;
...
m = asin(n);
```

where

m and n are of type double and $-1. \leq n \leq 1$.

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

The include file math.h defines this function, and uses the \$builtin version.

Returns

If this routine gets an illegal input, it returns 0.0 and sets errno to EDOM (defined in errno.h).

Example

```
/* program test for the asin() function */

#include <stdio.h>
#include <math.h>

double m, n, atof(), asin();

main(argc, argv)
    int    argc;
    char   *argv[];
{
    n = atof( argv[1] );
    printf("The arc sine of n = %f is m = %f.\n", n, m = asin( n ));
}
```

If you call the program test with the argument $-.277$, the output is

The arc sine of $n = -0.277000$ is $m = -0.280670$.

```
*****
assert                                     function
Provides runtime assertions.
*****
```

Calling Sequence

```
#include    <assert.h>
```

```
...
```

```
assert( expression );
```

If the expression is false at runtime, the program terminates, and a message is written on the standard error file. The message includes the name of the source file being compiled, line number, and page number (if greater than one).

If you define the macro NDEBUG before the include file, the assertion is omitted.

The assert macro causes a stack traceback if the expression is false, and calls the exit function with 2 to terminate the program.

```

*****
atan                                     $builtin function

Returns the arc tangent of the number passed to it.
*****

```

Calling Sequence

```

        double m, n, atan();
        ...
or
        $builtin double atan();
        double m, n;
        ...
        m = atan(n);

```

where

m and n are of type double

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

The include file math.h defines this function, and uses the \$builtin version.

```

| Returns
| -----
|
| If floating exceptions are raised, this function returns the
| largest possible floating point number and sets errno to ERANGE
| (defined in errno.h).

```

Related Functions

See also the atan2, tan, tanh functions.

Example

```
/* program test for the atan() function */
#include <stdio.h>
#include <math.h>

double m, n, atof(), atan();

main(argc, argv)

    int     argc;
    char    *argv[];
{
    n = atof( argv[1] );
    printf("The arc tangent of n = %f is m = %f.\n",
          n, m = atan( n ));
}
```

If you call the program test with the argument .00001, the output is

The arc tangent of n = 0.000010 is m = 0.000009.

atan2 \$builtin function

Returns the arc tangent of the two numbers passed to it.

Calling Sequence

```
double m, n, o, atan2();
```

```
...
```

or

```
$builtin double atan2();
```

```
double m, n, o;
```

```
...
```

```
m = atan2(n,o);
```

where

m, n, and o are of type double

The range of the result m is -3.1415926535, +3.1415926535.

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

The include file math.h defines this function, and uses the \$builtin version.

Related Functions

See also the atan, tan, tanh functions.

Example

```
/* program test for the atan2() function */  
  
#include <stdio.h>  
#include <math.h>  
  
double m, n, o, atof(), atan2();  
  
main(argc, argv)  
  
    int    argc;  
    char   *argv[];
```

```

{
n = atof( argv[1] );
o = atof( argv[2] );
printf("The arc tangent of n = %f and o = %f is m = %f.\n",
      n, o, m = atan2( n, o ));
}

```

If you call the program test with the arguments 3.4 and 5.6,
the output is

The arc tangent of n = 3.400000 and o = 5.600000 is m = 0.545655.

atof function

Converts an ASCII string into a floating-point number.

Calling Sequence

```
double result;  
char string[];  
double atof();  
...  
result = atof(string);
```

where

string begins with a valid arithmetic expression

Conversion stops when the function encounters an invalid character.

The include file `stdio.h` defines this function.

Related Functions

See also the `ftoa`, `atoi`, `atol`, `atou` functions.

Examples

```
/* program test for the atof() function */  
  
#include <stdio.h>  
  
double result, atof();  
  
main()  
{  
  
printf("%f\n", result = atof( " -12.03e12 " ));  
printf("%f\n", result = atof( ".07E-1" ));  
printf("%f\n", result = atof( "19.81" ));  
printf("%f\n", result = atof( "19b81" ));  
  
}
```

The output from the program test is

```
-12030000000000.000000  
0.007000  
19.810000  
19.000000
```

As the output shows, the atof function skips leading white space, treats a hyphen as a minus sign, and treats the characters 'e' and 'E' as exponents. But it stops processing the last string before the 'b', which is an invalid character.

atoi function

Converts an ascii string to type int.

Calling Sequence

```
int    result, atoi();
char   string[];
...
result = atoi(string);
```

where

result is of type int

string is a byte pointer to a character array that contains optional spaces, an optional plus or minus sign, a string of digits

The first nondigit terminates the scan. The function follows C conventions for radix specification.

The include file stdio.h defines this function.

Returns

No errors are flagged.

Related Functions

See also the itoa, atof, atol, atou functions.

Example

```
/* program test for the atoi() function */

#include <stdio.h>
#include <math.h>

int    result, atoi();
int    i = 1;
```

```

main(argc, argv)

    int    argc;
    char    *argv[];
{
    while ( i < argc )
    {
        printf("The result from %s is %d.\n", argv[i],
               result = atoi( argv[i] ));
        i++;
    }
}

```

If you call the program test with

```
x test [input_nos]
```

where the file input_nos contains

```

    1776 &
-1776 &
    1776 &
17-76 &
17LXX6

```

you generate the output

```

The result from 1776 is 1776.
The result from -1776 is -1776.
The result from 1776 is 1776.
The result from 17-76 is 17.
The result from 17LXX6 is 17.

```

atol function

Converts an ascii string to type long.

Calling Sequence

```
long    result, atol();
char    string[];

result = atol(string);
```

where

result is of type long

string is a byte pointer to a character array that contains optional spaces, an optional plus or minus sign, a string of digits

The first nondigit terminates the scan. The function follows C conventions for radix specification.

The include file `stdio.h` defines this function.

Returns

No errors are flagged.

Related Functions

See also the `atof`, `atoi`, `atou` functions.

Example

```
/* program test for the atol() function */

#include <stdio.h>
#include <math.h>

long    result, atol();
long    i = 1;
```



```

main(argc, argv)

    int    argc;
    char    *argv[];
{
    while ( i < argc )
    {
        printf("The result from %s is %ld.\n", argv[i],
               result = atol( argv[i] ));
        i++;
    }
}

```

If you call the program test with

```
x test [input_nos]
```

where the file input_nos contains

```

2001 &
-2001 &
2001 &
20-01 &
MM1

```

you generate the output

```

The result from 2001 is 2001.
The result from -2001 is -2001.
The result from 2001 is 2001.
The result from 20-01 is 20.
The result from MM1 is 0.

```

atou function

Converts an ASCII character string to an unsigned integer.

Calling Sequence

```
unsigned num, atou();  
char      *string;  
...  
num = atou(string);
```

where

num is an unsigned integer to receive the result

string is a byte pointer to a character array
containing optional space characters followed by
digits

The first nondigit terminates the string.

The include file `stdio.h` defines this function.

Returns

No error message is given if the number is too large.

Related Functions

See also the `utoa`, `atof`, `atoi`, `atol` functions.

Example

```
/* program test for the atou() function */  
  
#include <stdio.h>  
#include <math.h>  
  
unsigned      result, atou();  
unsigned      i = 1;
```

```

main(argc, argv)

    int    argc;
    char    *argv[];
{
    while ( i < argc )
    {
        printf("The result from %s is %u.\n", argv[i],
               result = atou( argv[i] ));
        i++;
    }
}

```

If you call the program test with

x test.c [input_nos]

where the file input_nos contains

```

10 &
      -10 &
+10.5 &
-10.5 &
32,255

```

you generate the result

```

The result from 10 is 10.
The result from -10 is 4294967286.
The result from +10.5 is 10.
The result from -10.5 is 4294967286.
The result from 32 is 32.
The result from 255 is 255.

```

```
*****
blockmove                                     function
```

Moves bytes from one location in memory to another.

```
*****
```

Calling Sequence

```
char    *source, *dest, *blockmove();
int      num;
...
blockmove(dest, source, num);
```

where

source is a pointer to the bytes to move

dest is a pointer to the location to move them to

num is the number of bytes to move

The include file `dglib.h` defines this function.

Returns

The return value is a byte pointer to the destination.

Related Functions

See also the `strncpy`, `strcpy` functions.

Example

```
/* program test for the blockmove() function */

#include <stdio.h>
#include <dglib.h>

int      i, count;
char     *dest;
char     *source = "ABCDEFGHIJKLMNOPQRSTUVWXYZ\
abcdefghijklmnopqrstuvwxy0123456789\
!@#$%^&*()_+=~`\\|[]{}\\'\"";:/.,<>";
```

```

main( argc, argv )
    int     argc;
    char    *argv[];
{
    dest = alloc( strlen( source ) );
    count = atoi( argv[1] );

    for( i = 1; i <= count; i++ ){
        blockmove( dest, source, i );

        printf("The first %d characters are:\n%.@s\n\n",
            i, i, dest );
    }
}

```

A call to the program test with

x test 7

generates the output

The first 1 characters are:
A

The first 2 characters are:
AB

The first 3 characters are:
ABC

The first 4 characters are:
ABCD

The first 5 characters are:
ABCDE

The first 6 characters are:
ABCDEF

The first 7 characters are:
ABCDEFG

calloc function

Allocates blocks of memory and zeros them out.

Calling Sequence

```
char    *ptr, *calloc();  
int      n, size;  
...  
ptr = calloc( n, size );
```

where

n is the number of bytes needed for the object

size is the number of objects to allocate

The total memory allocated is then $n \times \text{size}$ bytes. The function zeros the memory it allocates.

The include file `stdio.h` defines this function.

Returns

The function returns a pointer to the allocated block, or 0 if it cannot allocate that much memory from the heap.

Related Functions

See also the `alloc`, `alloca`, `malloc`, `realloc` functions.

Example

```
/* program test for the calloc() function */  
  
#include <stdio.h>  
  
char    *ptr, *calloc();  
int      n, size;
```

```

main(argc, argv)

    int     argc;
    char    *argv[];
{
    size = atoi( argv[1] );
    n = atoi( argv[2] );
    ptr = calloc( n, size );
    printf("Size of %d bytes x %d objects =\n",
           size, n );
    printf("an allocation %d in the block %o.\n",
           size * n, ptr );
}

```

A call to the program test with

x test 64 8

generates the output

Size of 64 bytes x 8 objects =
 an allocation 512 in the block 34003702770.

(The location will probably vary with execution.)

```
*****
ceil                                                    function
*****
```

Returns the ceiling of the number passed to it.

```
*****
```

Calling Sequence

```
double m, n,    ceil();
...
m = ceil(n);
```

where

m and n are of type double

The ceiling of a floating-point number f.p is f+1 if .p > 0,
and f otherwise.

The include file math.h defines this function.

This function does not check for type double.

Related Functions

See also the floor, fmax, fmin functions.

Example

```
/* program test for the ceil() function */

#include <stdio.h>
#include <math.h>

double m, n, atof(), ceil();
int i = 1;

main(argc, argv)
    int argc;
    char *argv[];
{
    while ( i < argc )
    {
        n = atof( argv[i] );
        printf("The ceiling of %f is %f.\n", n, m = ceil( n ));
        i++;
    }
}
```


If you call the program test with

x test [input_nos]

where the file input_nos contains

34.5 &
3.14159 &
9.0 &
.0001

you get the output

The ceiling of 34.500000 is 35.000000.
The ceiling of 3.141590 is 4.000000.
The ceiling of 9.000000 is 9.000000.
The ceiling of 0.000100 is 1.000000.

```
*****
channel                                     function

Returns the AOS/VS channel number for a UNIX file number.
*****
```

Calling Sequence

```
#include <pio.h>

FILE    *fd;
int      chno;
...
chno = channel( fd );
```

cos \$builtin function

Returns the cosine of the number passed to it.

Calling Sequence

```
double m, n, cos();  
...  
or  
  
$builtin double cos();  
double m, n;  
...  
m = cos(n);
```

where

m and n are of type double

n is expressed in radians

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

The include file math.h defines this function, and uses the \$builtin version.

This function does not check for type double.

Related Functions

See also the acos, cosh functions.

Example

```
/* program test for the cos() function */  
  
#include <math.h>  
  
double m, n, atof(), cos();  
int i = 1;
```

```

main(argc, argv)

    int    argc;
    char    *argv[];
{
    while ( i < argc )
    {
        n = atof( argv[i] );
        printf("The cosine of %f is %f.\n", n, m = cos( n ));
        i++;
    }
}

```

If you call the program test with

```
x test [input_nos]
```

where the file input_nos contains

```

55.5 &
5.55 &
0.555 &
0.0555

```

you get the output

```

The cosine of 55.500000 is 0.498726.
The cosine of 5.550000 is 0.743046.
The cosine of 0.555000 is 0.849900.
The cosine of 0.055500 is 0.998460.

```

```

*****
cosh                                     $builtin function

Returns the hyperbolic cosine of the type passed to it.
*****

```

Calling Sequence

```

        double m, n, cosh();
        ...
or
        $builtin double cosh();
        double m, n;
        ...
        m = cosh(n);

```

where

m and n are of type double
m gets the hyperbolic cosine of n

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

The include file math.h defines this function, and uses the \$builtin version.

This function does not check for type double.

Related Functions

See also the acos, cos functions.

Returns

If floating exceptions are raised, this function returns the largest possible floating point number (with the correct sign) and sets errno to ERANGE (defined in errno.h).

Example

```
/* program test for the cosh() function */

#include <math.h>

double m, n, atof(), cosh();
int i = 1;

main(argc, argv)
    int    argc;
    char   *argv[];
{
    while ( i < argc )
    {
        n = atof( argv[i] );
        printf("The hyperbolic cosine of %f is %f.\n",
               n, m = cosh( n ));
        i++;
    }
}
```

If you call the program test with

x test [input_nos]

where the file input_nos contains

```
5.55 &
0.555 &
0.0555 &
0.00555
```

you get the output

```
The hyperbolic cosine of 5.550000 is 128.620721.
The hyperbolic cosine of 0.555000 is 1.158006.
The hyperbolic cosine of 0.055500 is 1.001540.
The hyperbolic cosine of 0.005550 is 1.000015.
```

createdir function

Creates a directory or control point directory.

Calling Sequence

```
char    *name;  
long    size;  
int     err;  
...  
err = createdir(name,size);
```

where

name is a byte pointer to the directory name

The function creates a directory if size = 0, otherwise a CPD.
The created file has the ACLs of the creating process.

The include file dglib.h defines this function.

Returns

This function returns the error code in an error occurred,
otherwise 0.

Related Functions

See also the curdir, dir functions.

Example

```
/* Program test for the createdir() function */  
  
#include <dglib.h>  
  
long    err, size, atol();  
  
main( argc, argv )  
    int    argc;  
    char   *argv[];
```

```

{
    size = atol( argv[2] );
    if ( ( err = mkdir( argv[1], size ) ) != 0 )
        printf("Got error %o in creating directory.\n",
               err );
    else
        printf("Created directory %s\n", argv[1] );
}

```

A call to program test with

x test subc 1000

generates the output

Created directory subc

and a second identical call generates

Got error 26 in creating directory.

curdir function

Stores the name of the current directory.

Calling Sequence

```
int    curdir(), err;  
char   name[256];  
...  
err = curdir( name );
```

where

name receives the current directory name

Returns

If an error occurs, err is the AOS/VS error code, and 0 otherwise.

Related Functions

See also the createdir, dir functions.

Example

```
/* program test for the curdir() function */  
  
#include <dglib.h>  
  
char   dirname[256];  
int    err, curdir();  
  
main(argc, argv)  
{  
    int    argc;  
    char   *argv[];  
  
    err = curdir( dirname );  
    printf("directory is %s and error is %d.\n",  
          dirname, err );  
}
```

A call to the program test gets the output

directory is :UDD3:NITRAM:C:CDOC and error is 0.

dg_ctime function

Returns the current time.

Calling Sequence

```
#include <time.h>
```

```
char    *now, *dg_ctime();  
short   tvec[2];  
...  
now = dg_ctime( tvec );
```

where

now is a pointer to a static string

tvec points to two short integers that contain the
time in Data General format:

the first word is the number of days
since 12/31/67

the second word is the number of biseconds
since 12:00 a.m.

If tvec = 0, dg_ctime uses the current time.

The include file time.h defines this function.

Returns

The function returns

```
dow[ ]mth[ ]dm[ ]hh:mm:ss[ ]year
```

where

[] is a space
dow is three characters for the day of the week
mth is three characters for the month of the year
dm is two digits for the day of the month
hh is two digits for the current hour on a 24-hour clock
mm is two digits for the current minute of the hour
ss is two digits for the current second of the minute
year is four digits for the current year

Related Functions

See also the `dg_localtime` function.

Example

```
/* program test for the dg_ctime() function */

#include <stdio.h>
#include <time.h>

char    *now, *dg_ctime();

main(argc, argv)

    int    argc;
    char    *argv[];
{
    printf("The time and date is\n\n%s\n", now = dg_ctime( NULL ));
}
```

A call to the program test on August 19, 1982 at 10:07:06 generates the output

The time and date is

Thu Aug 19 10:07:06 1982

```
*****
dg_localtime                                     function
```

Returns a pointer to a nine-word array representing current time.

```
*****
```

Calling Sequence

```
#include <time.h>
```

```
struct tm      *dg_localtime(tvec);
short int      tvec[2];
struct tm      *ptr;
...
ptr = dg_localtime( *tvec[0] );
```

where

the time argument points to two short integers in Data General format:

the first word is the number of days since 12/31/67

the second word is the number of biseconds since midnight

If tvec is 0, dg_localtime returns the current time.

the include file time.h defines struct tm as

```
struct tm {
    int      tm_sec;          /* seconds in the minute */
    int      tm_min;          /* minutes in the hour   */
    int      tm_hour;         /* hours in the day      */
    int      tm_mday;         /* day of the month       */
    int      tm_mon;          /* month (0 = January)   */
    int      tm_year;         /* year - 1900           */
    int      tm_wday;         /* day of the week (0 = Sunday) */
    int      tm_yday;         /* ordinal day in the year */
    int      tm_isdst;        /* = 1 if daylight savings time */
};
```

The include file time.h defines this function.

Related Functions

See also the dg_ctime function.

Example

```
/* Program test for the dg_localtime() function */

#include <time.h>

struct tm      *ptr;

main()
{
    ptr = dg_localtime( 0 );

    printf("Number of seconds = %d\n", ptr->tm_sec );
    printf("Number of minutes = %d\n", ptr->tm_min );
    printf("Number of hours = %d\n", ptr->tm_hour );
    printf("Day of month = %d\n", ptr->tm_mday );
    printf("Month number = %d\n", ptr->tm_mon );
    printf("Year = %d\n", ptr->tm_year+1900 );
    printf("Day of week = %d\n", ptr->tm_wday );
    printf("Day of year = %d\n", ptr->tm_yday );
    if( ( ptr->tm_isdst ) == 1 )
        printf("Daylight savings\n");
    else
        printf("Standard time\n");
}
```

A call to the program test at

11:35:17 06-OCT-82

generates the output

```
Number of seconds = 17
Number of minutes = 35
Number of hours = 11
Day of month = 6
Month number = 9
Year = 1982
Day of week = 3
Day of year = 278
Standard time
```

dir function

Changes the working directory.

Calling Sequence

```
char    *dirname;  
int      err, dir();  
....  
err = dir(dirname);
```

where

dirname is a pointer to a directory pathname

The include file `dglib.h` defines this function.

Related Functions

See also the `createdir`, `curdir` functions.

equal
macro

Determines whether two strings are equal.

Calling Sequence

```
char    *string1, *string2;
int      equal(), str_equal;
...
str_equal = equal( string1, string2 );
```

where

string1 and string2 are pointers to strings ending in the null character ('\0')

Returns

If the two strings are equal, the return value is nonzero.

Related Functions

See also the any, str(n)chr, str(n)cmp, strpbrk, verify functions.

Example

```
/* program test for the equal() function */

#include <stdio.h>

char    *strcpy();
int      result, equal();

main(argc, argv)
{
    int      argc;
    char    *argv[];

    result = equal( argv[1], argv[2] );
    printf("%d - strings are%sequal.\n", result,
        ( result == 0 ) ? " not " : " " );
}
```


A call to the program test with the successive argument pairs

abc xyz
abc ABC
abc ab
ab abc
abc abc

generates the output

0 - strings are not equal.
0 - strings are not equal.
0 - strings are not equal.
0 - strings are not equal.
1 - strings are equal.

exit function

Flushes buffers and terminates the process normally.

Calling Sequence

```
[int    err;]  
...  
exit([ err ]);
```

Other C environments require an argument to be passed.

The include file `stdio.h` defines this function.

Returns

If the function has a nonzero argument, it represents an error code to be passed back to the father.

Related Functions

See also the `fclose`, `fflush` functions.

Example

```
/* program test for the exit() function */  
  
#include <stdio.h>  
  
int    err, atoi(), exit();  
  
main(argc, argv)  
{  
    int    argc;  
    char   *argv[];  
  
    err = atoi( argv[1] );  
    exit( err );  
}
```

If you call the program test with an argument of 256, you receive the message

WARNING

Program 'exit'ed with a status of o400

WARNING: FROM PROGRAM

```

*****
exp                                     $builtin function

Returns the constant e raised to the power of the number passed
to it.
*****

```

Calling Sequence

```

        double m, n, exp();
        ...
or
        $builtin double exp();
        double m, n;

        m = exp(n);

```

where

m and n are of type double

The constant e has an approximate value 2.718... and is the base of the natural system of logarithms.

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

The include file math.h defines this function, and uses the \$builtin version.

```

| Returns
| -----
|
| If floating exceptions are raised, this function returns the
| largest possible floating point number and sets errno to ERANGE
| (defined in errno.h).

```

Related Functions

See also the ldexp, scale10, log, log10, frexp functions.

Example

```
/* program test for the exp() function */

#include <math.h>

int      i = 1;
double   m, n, atof(), exp();

main(argc, argv)

    int      a          char    *argv[];
{
    while ( i < argc )
    {
        n = atof( argv[i] );
        printf("Value of e to the power of %f = %f\n",
               n, m = exp( n ));
        i++;
    }
}
```

If you call the program test with

x test [input_nos]

where the file input_nos contains

```
1 &
2.718281 &
3.14159
```

you generate the output

```
Value of e to the power of 1.000000 = 2.718281
Value of e to the power of 2.718281 = 15.154249
Value of e to the power of 3.141590 = 23.140631
```

```
*****
fabs                                     $builtin function
```

Returns the absolute value of the number passed to it.

```
*****
```

Calling Sequence

```
double m, n, fabs();
```

```
...
```

or

```
$builtin double fabs();
```

```
double m, n;
```

```
...
```

```
m = fabs(n);
```

where

m and n are of type double

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

The include file math.h defines this function, and uses the \$builtin version.

Related Functions

See also the abs function.

Example

```
/* program test for the fabs() function */
```

```
#include <math.h>
```

```
int    i = 1;
double m, n, atof(), fabs();
```

```
main(argc, argv)
```

```
int    argc;
char   *argv[];
```

```

{
while ( i < argc )
{
    n = atof( argv[i] );
    printf("Absolute value of %f% = %f\n",
        n, m = fabs( n ));
    i++;
}
}

```

If you call the program test with

x test [input_nos]

where the file input_nos contains

```

12.3e-2 &
-34.5e-2 &
-.9999 &
2.718281 &
3.14159

```

you generate the output

```

Absolute value of 0.123000= 0.123000
Absolute value of -0.345000= 0.345000
Absolute value of -0.999900= 0.999900
Absolute value of 2.718281= 2.718281
Absolute value of 3.141590= 3.141590

```

fchannel function

Returns the AOS/VS channel for a file opened with fopen.

Calling Sequence

#include <pio.h>

```
FILE    *fd;  
int     chno;  
...  
chno = channel( fd );
```

fclose function

Closes a specified file.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;  
...  
fclose(fd);
```

where

fd is a pointer to an I/O packet

The function flushes the file's buffer before closing the file.

The include file `stdio.h` defines this function.

Returns

The function returns -1 on an error and 0 otherwise.

Related Functions

See also the `exit`, `fflush` functions.

Example

```
/* program test for the fclose() function */  
  
#include <stdio.h>  
  
FILE    *fopen(), *fd;  
  
char     *type;  
int      i = 1;  
  
main(argc, argv)  
  
    int      argc;  
    char     *argv[];
```

```

{
    type = "r";
    while ( i < argc )
    {
        fd = fopen( argv[i], type );
        printf("%c\n", fgetc( fd ));
        fclose( fd );
        i++;
    }
}

```

If you call program test with

```
x test char_a char_b char_c char_d
```

where each file char_x contains just

```
xxxxxxx
```

you generate the output

```

a
b
c
d

```

The program opens each file in turn, gets the first character, and closes the file.

fdopen function

Associates a buffered I/O stream with a UNIX file descriptor.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *stream, *fdopen();  
int      fildes;  
char     *options;  
...  
fdopen( fildes, options );
```

where

fildes is a UNIX file descriptor (returned with creat, dup open, or the like, and

options is a byte pointer to a string specifying the standard opening types

"a"	Open for appending only (UNIX compatible).
"r"	Open for reading only (UNIX compatible).
"w"	Delete, recreate, and open for writing only (UNIX compatible).

or one of the additional Data General opening types

"a+"	Open for appending (read and write).
"b"	Open for block I/O.
"j"	Open for binary read.
"k"	Open for binary write.
"r+"	Open for read/write access. Return an error if the file does not exist.
"s"	Open for shared page I/O.
"u"	Open for updating.
"x"	Open file exclusively for updating.
"w+"	Delete, recreate, and open a file for read/write access.

Provided your file is a disk file, you need not take any special action to alternate between reading and writing. The I/O routines correctly switch from one mode to the other. (If your program has to run under other C compilers, you may have to use an fseek(stream, 0L, 1) when switching I/O types.)

Returns

The function returns NULL if an opening error occurs.

```

*****
feof                                                    macro

Returns a nonzero when it encounters an end of file.
*****

```

Calling Sequence

```

#include <stdio.h>

FILE    *iop;
int      eofset;
...
eofset = feof ( iop );

```

Since feof is a #define macro, you will get bizarre errors if you declare it as a function.

Example

```

/* program test for the feof() function */

#include <stdio.h>

FILE    *fopen(), *fd;

char     *type, c;
int      i = 1, eofbit;

main(argc, argv)
{
    int      argc;
    char     *argv[];

    type = "r";
    while ( i < argc )
    {
        fd = fopen( argv[i], type );
        while ( 0 == feof( fd ) )
            printf("%c", c = fgetc( fd ));
        printf("eofbit = %d.\n", eofbit = feof( fd ));
        fclose( fd );
        i++;
    }
}

```

If you call the program test with

x test line1 line2 line3

where the three files contain respectively

This is the first line.

This is a second line of text.

This is the third and last line.

you generate the output

This is the first line.

eofbit = 32.

This is a second line of text.

eofbit = 32.

This is the third and last line.

eofbit = 32.

```
*****
ferror                                                    macro
```

Returns a nonzero if an error occurred during I/O.

```
*****
```

Calling Sequence

```
-----
```

```
FILE    *iop;
int      errset;
...
errset = ferror( iop );
```

Since ferror is a macro, you will get bizarre errors if you attempt to redeclare it or pass it to a function.

Example

```
-----
```

```
/* program test for the ferror() function */

/* This example shows a use for the ferror macro.      */
/* Basically it expects stdin to come from the terminal */
/* and enables the timeout characteristic - it          */
/* reports the error if the user doesn't type in time.  */
/* Program argument: argv[1] = number of minutes to    */
/* allow user.                                          */

#include <stdio.h>
#include <paru.h>

char    ttyname[] = "@CONSOLE";

main( argc, argv )
{
    int      argc;
    char     *argv[];

    short    terminal[$CLMAX]; /* characteristics */
    int      ch;
    if ( sys_gechr( ttyname, $CLMAX, terminal ) )
    {
        perror( "gechr" );
        exit( 1 );
    }
    terminal[1] |= ( 0100000 >> $CTO ); /* enable timeout */
    if ( sys_sechr( ttyname, $CLMAX, terminal ) )
```

```

{
    perror( "sechr" );
    exit( 2 );
}
if ( sys_stom( ttyname, 0, atoi( argv[1] ) ) )
{
    perror( "stom" );
    exit( 1 );
}

while ( ( ch = getchar() ) != ERR )
    putchar( ch );
if ( ferror( stdin ) )
    printf("Timeout occurred.\n");

terminal[1] &= ~( 0100000 >> $CTO );
/* remove timeout bit */
sys_sechr( ttyname, $CLMAX, terminal );
}

```

A call to the program test with

x test 1

generates

Timeout occurred.

fflush function

Flushes the output buffer of the specified stream.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;  
...  
fflush(fd);
```

where

fd is a pointer to an I/O packet

The fflush function assumes that the file is open.

Returns

The function returns -1 if an error occurs, 0 otherwise.

Related Functions

See also the fclose, freopen functions.

Example

```
/* program test for the fflush() function */  
  
#include <stdio.h>  
  
main( argc, argv )  
{  
    int    argc;  
    char   *argv[];  
  
    int    i;  
  
    for ( i = 1; i < argc; i++ )  
        printf("%s_", argv[i] );  
}
```



```

        fflush( stdout );           /* insure characters */
                                     /* written out before */
        sleep( 2*60 );              /* sleeping          */
                                     /* go to sleep for    */
                                     /* 2 minutes         */
    }

```

A call to the program test with

```
x test a b c d e f
```

generates the output

```
a _ b _ c _ d _ e _ f _
```

before going to sleep for 2 minutes.

fgetc function

Gets a character from a general file.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;  
char    nextchar;  
...  
nextchar = fgetc( fd );
```

where

nextchar is the character to get

fd is a pointer to a FILE packet

The function assumes that the file is open.

The include file `stdio.h` defines this function.

If you define the macro `_NEXPAND`, the `stdio.h` macros `getc` and `putc` resolve to the library functions `fgetc` and `fputc`. This allows you to set a breakpoint at the routine while debugging.

Returns

The function returns EOF (-1) on end of file, or error.

Related Functions

See also the `getc`, `fputc`, `fgets`, `gets`, `getchar`, `getw`, `ungetc` functions.

Example

```
/* program test for the fgetc() function */

#include <stdio.h>

FILE    *fopen(), *fd;

char    c, *type;
int     i = 1, n;

main(argc, argv)
{
    int     argc;
    char    *argv[];

    type = "r";

    while ( i < argc )
    {
        fd = fopen( argv[i], type );
        printf("%c", c = fgetc( fd ) );
        i++;
    }
}
```

A call to program test with

x test [char_in]

where char_in contains the valid filenames

char_in_a &
char_in_b &
char_in_c &
char_in_d

and where each file char_in_x contains just instances of x
generates the output

abcd

fgets function

Reads a string from a specified file.

Calling Sequence

#include <stdio.h>

```
FILE    *fd;  
char    stringbuf[], *fgets();  
int     max;  
...  
fgets(stringbuf, max, fd);
```

where

stringbuf is a pointer to a user-provided buffer

int specifies the maximum number of characters to read

fd is a pointer to a file packet

The read halts at max characters or a delimiter, whichever comes first, reading max-1 characters into the buffer and appending a NULL.

The include file stdio.h defines fgets.

Returns

The function returns 0 (NULL) at end of file, or error, or stringbuf if no error occurs.

Related Functions

See also the fputs, getc, fgetc, getchar, getw, ungetc functions.

Example

```
/* program test for the fgets() function */

#include <stdio.h>

FILE    *fopen(), *fd;

char    string[6], *type, *fgets(), err;
int     i = 1, max;

main(argc, argv)

        int     argc;
        char    *argv[];
{

    max = 6;
    type = "r";

    while ( i < argc )
    {

        fd = fopen( argv[i], type );
        err = fgets( string, max, fd );
        printf("%s\n", string );
        i++;

    }

}
```

A call to the program test with

```
x test [char_in]
```

where char_in contains the valid filenames

```
char_in_a &
char_in_b &
char_in_c
```

which contain respectively

```
aaaaaaa
bbbbbb
ccc
```

generates the output

```
aaaaa
bbbbbb
ccc
```

```
*****
fileno                                                    macro
```

Returns the UNIX emulated file number of the specified open file.

```
*****
```

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;
int      no;
...
no = fileno(fd);
```

where

no is the channel number

fd is a pointer to a FILE packet

The include file stdio.h defines this function.

Example

```
/* program test for the fileno() function */
```

```
#include <stdio.h>
```

```
FILE    *fopen(), *fd;

char     c, *type;
int      i = 1, n;

main(argc, argv)

    int      argc;
    char     *argv[];
{
    type = "r";
    while ( i < argc )
    {
        fd = fopen( argv[i], type );
        printf("File %s is open on file %d.\n",
               argv[i], n = fileno( fd ) );
        i++;
    }
}
```

A call to the program test with

x test [char_in]

where the file char_in contains the valid filenames

char_in_a &
char_in_b &
char_in_c &
char_in_d &
char_in_e &
char_in_f &

generates the output

File char_in_a is open on file 3.
File char_in_b is open on file 4.
File char_in_c is open on file 5.
File char_in_d is open on file 6.
File char_in_e is open on file 7.
File char_in_f is open on file 8.

```

*****
floor                                                    function

Returns the floor of the number passed to it.
*****

```

Calling Sequence

```

double m, n, floor();
...
m = floor(n);

```

where

m and n are of type double

The floor of a positive floating-point number f.p is f.
The floor of a negative floating-point number -f.p is -(f+1).

Related Functions

See also the ceil, min, fmin functions.

Example

```

/* program test for the floor() function */

#include <stdio.h>
#include <math.h>

int      i = 1;
double m, n, atof(), floor();

main(argc, argv)
{
    int      argc;
    char     *argv[];

    while ( i < argc )
    {
        m = atof( argv[i] );
        printf("Floor of %f = %f\n", m, n = floor( m ));
        i++;
    }
}

```


A call to the program test with

x test [input_nos]

where the file input_nos contains

4.5 &
4.0 &
-3.5 &
.007e2

generates the output

Floor of 4.500000 = 4.000000
Floor of 4.000000 = 4.000000
Floor of -3.500000 = -4.000000
Floor of 0.700000 = 0.000000

```
*****
fmax                                     $builtin function
```

Returns the floating maximum of two numbers.

```
*****
```

Calling Sequence

```
double a, b, c, fmax();
...
```

or

```
$builtin double fmax();
double a, b, c;
...
a = fmax( b, c );
```

where

a, b, and c are of type double

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

The include file math.h defines this function, and uses the \$builtin version.

Related Functions

See also the fmin, max, min, ceil, floor functions.

Example

```
/* program test for the fmax() function */

#include <stdio.h>
#include <math.h>

int i = 1;
double m, n, o, atof(), fmax();

main(argc, argv)
    int argc;
    char *argv[];
{
```

```

while ( i+1 < argc )
{
    m = atof( argv[i] );
    n = atof( argv[i+1] );
    printf("fmax of %f and %f = %f\n", m, n, o = fmax( m, n ));
    i++;
}
}

```

A call to the program test with

x test [input_nos]

where the file input_nos contains

```

6.78e-5 &
.0001 &
-3.1416 &
.0098e3

```

generates the output

```

fmax of 0.000067 and 0.000100 = 0.000100
fmax of 0.000100 and -3.141600 = 0.000100
fmax of -3.141600 and 9.800000 = 9.800000

```

```
*****
fmin                                     $builtin function
```

Returns the floating minimum of two numbers.

```
*****
```

Calling Sequence

```
double a, b, c, fmin();
...
```

or

```
$builtin double fmin();
double a, b, c;
...
a = fmin( b, c );
```

where

a, b, and c are of type double

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

The include file math.h defines this function, and uses the \$builtin version.

Related Functions

See also the fmax, max, min, ceil, floor functions.

Example

```
/* program test for the fmin() function */

#include <stdio.h>
#include <math.h>

int i = 1;
double m, n, o, atof(), fmin();

main(argc, argv)
    int argc;
    char *argv[];
{
```

```

while ( i+1 < argc )
{
    m = atof( argv[i] );
    n = atof( argv[i+1] );
    printf("fmin of %f and %f = %f\n", m, n, o = fmin( m, n ));
    i++;
}
}

```

A call to the program test with

```
x test [input_nos]
```

where the file input_nos contains

```

6.78e-5 &
.0001 &
-3.1416 &
.0098e3

```

generates the output

```

fmin of 0.000067 and 0.000100 = 0.000067
fmin of 0.000100 and -3.141600 = -3.141600
fmin of -3.141600 and 9.800000 = -3.141600

```

```
*****
fmod                                     function

Returns the remainder of the first argument divided by the
second argument.
*****
```

Calling Sequence

```
double x, y, z, fmod();
...
y = fmod(x,z);
```

where

x, y, and z are of type double

y receives the remainder of the division of x by z

The include file math.h defines this function.

Related Functions

See also the `modf` function.

Example

```
/* program test for the fmod() function */

#include <stdio.h>
#include <math.h>

int    i = 1;
double m, n, o, atof(), fmod();

main(argc, argv)
{
    int    argc;
    char   *argv[];

    while ( i+1 < argc )
    {
        m = atof( argv[i] );
        n = atof( argv[i+1] );
        printf("fmod of %f and %f = %f\n", m, n, o = fmod( m, n ));
        i++;
    }
}
```

A call to the program test with

```
x test [input_nos]
```

where the file `input_nos` contains

```
21 &
13 &
8 &
5 &
3
```

generates the output

```
fmod of 21.000000 and 13.000000 = 8.000000
fmod of 13.000000 and 8.000000 = 5.000000
fmod of 8.000000 and 5.000000 = 3.000000
fmod of 5.000000 and 3.000000 = 2.000000
```

```
*****
fopen                                     function
Opens a file.
*****
```

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fopen();
char    *path, *type;
...
fopen(path, type);
```

where

path is a byte pointer to the pathname of the file

type is a byte pointer is a string specifying one of the standard opening types

```
|          "a"    Open for appending only (UNIX compatible).
|          "r"    Open for reading only (UNIX compatible).
|          "w"    Open for writing only (UNIX compatible).
```

or one of the additional Data General opening types

```
|          "a+"   Open for appending (read and write).
|          "b"    Open for block I/O.
|          "j"    Open for binary read.
|          "k"    Open for binary write.
|          "r+"   Open for read/write access. Return an error
|                  if the file does not exist.
|          "s"    Open for shared page I/O.
|          "u"    Open for updating.
|          "x"    Open file exclusively for updating.
|          "w+"   Delete, recreate, and open a file for read/
|                  write access.
```

```
| Provided your file is a disk file, you need not take any special
| action to alternate between reading and writing. The I/O routines
| correctly switch from one mode to the other. (If your program
| has to run under other C compilers, you may have to use an
| fseek( stream, 0L, 1 ) when switching I/O types.)
```


If the file does not exist, the function tries to create it, unless the mode is "r". On the "w" option, the function always deletes the file first and then recreates it.

If a file is opened under any option except "s" and is updated by more than one process, there can be no guarantee of the file's contents. This is because the AGENT buffers files in its address space, and not in the system-wide address space of the kernel.

If you use the "s" option, you must reserve one or more pages with the /RESERVE= switch on the CCL link macro.

If you are running under AOS/RT32, you must use either the "b" option for block I/O, or the "s" option for shared page I/O, unless the file is a device like the console. For more details, consult the "AOS/RT32 System Programmer's Reference".

Returns

The function returns a NULL if the opening fails, otherwise it returns a pointer to pass to the other I/O routines.

Related Functions

See also the fclose, fdopen, freopen functions.

Example

```
/* program test for the fopen() function */

#include <stdio.h>
#define CHARMAX 80
#define STRINGMAX 5

FILE    *fdi, *fdo, *fopen();
char    stringbuf[CHARMAX];
int      i = 0;

main(argc, argv)

    int      argc;
    char    *argv[];
{
    fdi = fopen( argv[1], "r" );
    fdo = fopen( argv[2], "a" );

    while ( i < STRINGMAX )
    {
        fgets( stringbuf, CHARMAX, fdi );
        printf("%s", stringbuf );
        fputs( stringbuf, fdo );
        i++;
    }
}
```

Suppose the file infile contains

This is line 1
This is line 2
This is line 3
This is line 4
This is line 5
This is line 6
This is line 7

and the file outfile contains

You did not append this line or the next.
The following lines are from infile:

A call to the program test with

x test infile outfile

generates the terminal output

This is line 1 -
This is line 2
This is line 3
This is line 4
This is line 5

and the file outfile now contains

You did not append this line or the next.
The following lines are from infile:
This is line 1
This is line 2
This is line 3
This is line 4
This is line 5

```
*****
fprintf                                     function
*****

Writes to a file.
*****
```

Calling Sequence

```
#include <stdio.h>

FILE    *fd;
...
fprintf( fd, format, arg1, ...)
```

where

fd is a pointer to the output FILE packet

each arg1 is a valid C variable or constant

format is a string containing two kinds of objects:

ordinary characters that the function copies directly

conversion specifications that start with '%'

The function replaces the format specifications with the successive arg1, and writes the result to the file. For details of conversion specification, see the printf function.

Returns

| The function returns -1 if an error occurs, and otherwise the number
| of characters output.

Related Functions

See also the fscanf, printf, sprintf functions.

Example

```
/* program test for the fprintf() function */
```

```
#nolist
```

```
#include <stdio.h>
```

```
#include <math.h>
```

```
#list
```

```
FILE    *fd, *fopen();
```

```
int      i = 2, ibuf, atoi();
```

```
double   dbuf, atof();
```

```
main(argc, argv)
```

```
    int    argc;  
    char   *argv[];
```

```
{  
    fd = fopen( argv[1], "a" );  
    fprintf(fd, "This program prints numerical data to a file in a\n");  
    fprintf(fd, "variety of formats.\n\n");  
    while ( i < argc )  
    {  
        dbuf = atof( argv[i] );  
        fprintf(fd, "%e in scientific notation =\n", dbuf );  
        fprintf(fd, "%f in normal notation =\n", dbuf );  
        ibuf = atoi( argv[i] );  
        fprintf(fd, "%d decimal =\n%o octal =\n%x hexadecimal.\n\n",  
                ibuf, ibuf, ibuf );  
        i++;  
    }  
}
```

A call to the program test with

```
x test result_nos [input_nos]
```

when the file result_nos is empty and the file input_nos contains

```
144 &  
3.1416 &  
-99.99e4
```

generates in the file result_nos the output

This program prints numerical data to a file in a variety of formats.

```
1.440000e+02 in scientific notation =  
144.000000 in normal notation =  
144 decimal =  
220 octal =  
90 hexadecimal.
```

```
3.141600e+00 in scientific notation =  
3.141600 in normal notation =  
3 decimal =  
3 octal =  
3 hexadecimal.
```

```
-9.999000e+05 in scientific notation =  
-999900.000000 in normal notation =  
-99 decimal =  
3777777635 octal =  
ffffff9d hexadecimal.
```

fputc function

Puts a character out to the specified stream.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;  
char    nextchar;  
...  
fputc(nextchar, fd);
```

where

nextchar is the character to put out

fd is a pointer to a FILE packet

The function assumes that the file is open.

The include file `stdio.h` defines this function.

If you define the macro `NEXPAND`, the `stdio.h` macros `getc` and `putc` resolve to the library functions `fgetc` and `fputc`. This allows you to set a breakpoint at the routine while debugging.

Related Functions

See also the `fgetc`, `putc`, `putchar`, `puts`, `putw` functions.

Example

```
/* program test for the fputc() function */  
  
#include <stdio.h>  
  
FILE    *fopen(), *fd;  
int      i = 0;  
char     *type;  
  
main(argc, argv)  
  
    int      argc;  
    char     *argv[];
```

```

{
    type = "a";
    fd = fopen( argv[1], type );
    while ( argv[2][i] != NULL )
    {
        fputc( argv[2][i], fd );
        i++;
    }
}

```

Three calls to the program test with

```

x test apfile datum1
x test apfile datum2
x test apfile datum3

```

generate

```

datum1datum2datum3

```

as the contents of apfile (if apfile had no prior contents).

fputs function

Writes a string to a specified file.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;  
char    *stringbuf;  
int      fputs();  
...  
fputs( stringbuf, fd );
```

where

stringbuf is a pointer to a user-provided buffer

fd is a pointer to a FILE packet

The function writes until it encounters a null byte. It does not append a NEW LINE.

The include file stdio.h defines fputs.

Returns

The function returns -1 on error, 0 otherwise.

Related Functions

See also the fgets, puts, printf, fprintf functions.

Example

```
/* program test for the fputs() function */
```

```
#include <stdio.h>
```

```
FILE    *fd;  
int      i = 2;  
char    *type;
```

```

main( argc, argv )
    int     argc;
    char    *argv[];
{
    type = "a";
    fd = fopen( argv[1], type );
    while ( i < argc )
    {
        fputs( argv[i], fd );
        i++;
    }
}

```

A call to the program test with

```
x test input_str mul ti syl lab ic
```

appends to the file input_str the string

```
multisyllabic
```

fread function

Reads n records of m bytes each from a specified file.

Calling Sequence

```
#include <stdio.h>
```

```
char    *buffer;  
FILE    *fd;  
int      numrec, numbyt, numread;  
...  
numread = fread( buffer, numbyt, numrec, fd );
```

where

buffer is a byte pointer to a user-provided buffer

numbyt is the number of bytes per record

numrec is the number of records to read

fd is a pointer to a FILE packet

numread is the number of records actually read

The include file `stdio.h` defines this function.

Since the hardware representation for byte pointers differs from that for word pointers, `stdio.h` defines a macro `fread` to coerce the first argument to a byte pointer.

Returns

The function returns the number of whole records read. If an error or end of file occurs, 0 is returned.

Related Functions

See also the `fwrite` function.

Example

```
-----  
/* program test for the fread() function */  
  
#include <stdio.h>  
  
FILE    *fopen(), *fd;  
char     *type, buffer[100];  
int      atoi(), numrec, numbyt, numread;  
  
main( argc, argv )  
  
    int      argc;  
    char     *argv[];  
{  
  
    type = "r";  
    fd = fopen( argv[1], type );  
    numrec = atoi( argv[2] );  
    numbyt = atoi( argv[3] );  
  
    printf("Number of records read = %d\n",  
           numread = fread( buffer, numbyt, numrec, fd ) );  
  
}
```

A call to the program test with

x test input_nos 8 2

where the file input_nos contains

0123456789

generates the output

Number of records read = 5

```
*****
free                                     $builtin function

Releases memory allocated through alloc, calloc, malloc.
*****
```

Calling Sequence

```
-----

#include <stdio.h>

    char    *fp, *free();
    void     free();
    ...

or

    $builtin void    free();
    char    *fp;
    ...
    free(fp);
```

where

fp is a pointer to the allocated memory

This function frees the storage that the pointer points to. You must have previously allocated the area with alloc, calloc, malloc, or realloc. |

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers. The include file stdio.h defines this entry, and uses the \$builtin version.

Since the hardware representations for byte and word pointers differ, stdio.h defines a macro named free. It converts the argument to a byte pointer before doing the call.

Example

```
/* program test for the free() function */

#include <stdio.h>

char    *p, *alloc();

main()

{

    p = alloc( 100 );
    printf("Allocated %o\n", p );
    free( p );
    printf("Area released.\n");

}
```

A call to the program test generates the output

```
Allocated 34003703624
Area released.
```

freopen function

Closes a channel and reopens a file or device.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *oldfile, *freopen();  
char    *newfile, *type;  
...  
freopen(newfile, type, oldfile);
```

where

newfile is a byte pointer to the new pathname

oldfile is a pointer to the I/O packet of the currently open file

type is a byte pointer to a string specifying the standard opening types

"a"	Open for appending only (UNIX compatible).	
"r"	Open for reading only (UNIX compatible).	
"w"	Delete, recreate, and open for writing only (UNIX compatible).	

or one of the additional Data General opening types

"a+"	Open for appending (read and write).	
"b"	Open for block I/O.	
"j"	Open for binary read.	
"k"	Open for binary write.	
"r+"	Open for read/write access. Return an error if the file does not exist.	
"s"	Open for shared page I/O.	
"u"	Open for updating.	
"x"	Open file exclusively for updating.	
"w+"	Delete, recreate, and open a file for read/write access.	

Provided your file is a disk file, you need not take any special action to alternate between reading and writing. The I/O routines correctly switch from one mode to the other. (If your program has to run under other C compilers, you may have to use an fseek(stream, 0L, 1) when switching I/O types.)

Returns

The function returns NULL if an opening error occurs, otherwise it returns the pointer oldfile.

Example

```
-----
/* program test for the freopen() function */

#include <stdio.h>

FILE    *fd, *fopen(), *freopen();
char    *type;
int     i = 1;

main( argc, argv )
{
    int     argc;
    char    *argv[];

    type = "r";
    fd = fopen( argv[i], type );
    putchar( fgetc( fd ) );

    while ( i+1 < argc )
    {
        fd = freopen( argv[i+1], type, fd );
        putchar( fgetc( fd ) );
        i++;
    }
}
-----
```

A call to the program test with

x test [char_in]

where each element of the file char_in is a valid filename
generates the output

abcde

| if the first file begins with a, and each successive file begins
| with the next letter of the alphabet.


```
*****
frexp                                                    function
```

Returns a double mantissa $x \leq 1$ and stores an integer n indirectly through a pointer where the original value is $x*2^n$.

Calling Sequence

```
double frexp(), value, result;
int     *eptr;
...
result = frexp( value, eptr);
```

Example

```
/* program test for the frexp() function */

int     i, exp;
double  number, result, atof(), frexp();

main( argc, argv )
{
    int     argc;
    char    *argv[];

    for ( i = 1; i < argc; i++ )
    {
        number = atof( argv[i] );
        result = frexp( number, &exp );
        printf("Number %g has an exponent ", number );
        printf("of %d, mantissa of %g\n", exp, result );
    }
}
```

A call to the program test with

x test [input_nos]

where the file input_nos contains

```
.0031416 &
3.1416 &
3141.6
```

generates the output

Number 3.1416e-03 has an exponent of -8, mantissa of 0.804249
Number 3.1416 has an exponent of 4, mantissa of 0.19635
Number 3141.6 has an exponent of 12, mantissa of 0.766992

fscanf function

Reads from a file.

Calling Sequence

#include <stdio.h>

```
FILE    *fd;  
...  
fscanf(fd, format, arg1 ... );
```

where

fd is a pointer to the input file

each argi is a valid C variable or constant

format is a string containing three kinds of objects:

spaces, NEW LINES, and TABs, which the function ignores

ordinary characters that must match the next nonspace character directly

conversion specifications that start with '%'

The function attempts to read successive characters from a file and convert the characters according to a specified format, updating the variables the successive arguments point to. For details of conversion specification, see the scanf function.

Returns

The return value from fscanf is the number of items successfully matched and assigned, or -1 if an end of file or error is found and no characters were read.

Related Functions

See also scanf, sscanf, fprintf.

Example

```
-----  
/* program test for the fscanf() function */  
  
#include <stdio.h>  
#define MAXCLASS 5  
#define MAXNAME 16  
  
FILE    *fd, *fopen();  
char    *type;  
char    name[MAXNAME];  
int     i = 1, grade;  
  
main( argc, argv )  
  
    int     argc;  
    char    *argv[];  
{  
  
    type = "r";  
    fd = fopen( argv[1], type );  
    printf("Class:  %s\n\n", argv[1]);  
    printf("NAME\t\t\tGRADE\n\n");  
    while ( i <= MAXCLASS )  
    {  
  
        fscanf( fd, "%s %d", &name, &grade );  
        printf("%s\t\t\t%d\n", name, grade );  
        i++;  
  
    }  
  
}
```

A call to the program test with

x test Latin.1

where the file Latin.1 contains

```
Brown 75  
Jones      86  
Smith  
12  
Varro 92  
Vergil 57
```

generates the output

Class: Latin.1

NAME	GRADE
Brown	75
Jones	86
Smith	12
Varro	92
Vergil	57

fseek function

Positions the file to a specified byte.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fp;  
long    disp;  
int      mode, error, fseek();  
...  
error = fseek(fp, disp, mode);
```

where

fp is a pointer to a file descriptor packet
for an open file

disp is a byte displacement

mode uses disp as an offset from

0	beginning of file
1	current position
2	end of file

The include file `stdio.h` defines this function.

Returns

The function returns 0 in err if no error occurs, and
-1 otherwise.

Related Functions

See also `lseek` in the UNIX emulation document.

Example

```
/* program test for the fseek() function */

#include <stdio.h>

FILE    *fd, *fopen();
char     next, *type;
long     disp = 0, mode = 2;

main( argc, argv )

    int     argc;
    char    *argv[];
{

    type = "r";
    fd = fopen( argv[1], type );
    while ( fseek ( fd, disp, mode ) == 0 )
    {

        next = getc( fd );
        putchar( next );
        disp--;

    }

}
```

A call to the program test with itself as argument

```
x test test.c
```

generates as output the reversed file

```
    }
    }

    ;--psid
    ;) txen (rahctup
    ;) df (cteg = txen

    {
    ) 0 == ) edom ,psid ,df ( keesf ( elihw
    ;) epyt ,]l[vgra (nepof = df
    ;"r" = epyt

    {
    ;][vgra*      rahc
    ;cgra   tni

    ) vgra ,cgra (niam

    ;2 = edom ,0 = psid      gnol
    ;epyt* ,txen   rahc
    ;)(nepof* ,df*  ELIF

    >h.oidts< edulcni#

    /* noitcnuf )(keesf eht rof c.tset margorP */
```

ftell function

Returns the current position in a file.

Calling Sequence

#include <stdio.h>

```
FILE    *fd;  
long    position, ftell();  
...  
position = ftell( fd );
```

where

position is the current byte position within a file

fd is a pointer to an i/o packet

The include file stdio.h defines this function.

Example

/* program test for the ftell() function */

#include <stdio.h>
#define EOF -1

FILE *fd, *fopen();
char *type;
int next;
long position, ftell();

main(argc, argv)

 int argc;
 char *argv[];

 {

 type = "r";
 fd = fopen(argv[1], type);
 while ((next = getc(fd)) != EOF)
 {

 putchar(next);
 printf(" %ld ", position = ftell(fd));

 }

 }

}

A call to program test with

x test input_str

where the file input_str contains

My last byte.

generates the output

M 1 y 2 3 1 4 a 5 s 6 t 7 8 b 9 y 10 t 11 e 12 . 13

ftoa function

Converts a floating-point number to a string.

Calling Sequence

```
float   fnum;  
char    *string, mode;  
int     precision;  
...  
ftoa(fnum, string, precision, mode);
```

where

fnum is the floating-point number to convert

string is a byte pointer to the beginning of the string

precision is the number of digits following the decimal point

mode specifies

'e' for scientific notation

'f' for normal notation

'g' for scientific or normal notation, whichever is
smaller, with nonsignificant trailing 0s removed

'E' is like 'e' with exponent E uppercase

'G' is like 'g' with exponent G uppercase

The maximum size of the character array string is 160 characters
if you convert 1e75 with mode = 'f' and precision >= 79.

The include file stdio.h defines this function.

The include file stdio.h also defines a #define macro L_ftoa
which is the maximum number of characters that ftoa returns
(160).

Example

```
/* program test for the ftoa() function */

#include <stdio.h>

FILE    *fd, *fopen();
float    fnum1, fnum2;
char    alf[160], *type, mode = 'g';
int      i = 1, precision = 32;

main( argc, argv )

    int    argc;
    char   *argv[];
{
    type = "w";
    fd = fopen( argv[i], type );
    i++;
    while ( i+1 < argc )
    {
        fnum1 = atof( argv[i] );
        fnum2 = atof( argv[i+1] );
        fnum1 *= fnum2;
        ftoa( fnum1, alf, precision, mode );
        fputs( alf, fd );
        fputc( '\n', fd );
        i++;
    }
}
```

A call to the program test with

```
x test mulfloat [input_nos]
```

when the input file input_nos contains

```
3.14159e3 &
.00002 &
54321.23456 &
6.9e4 &
13e-4 &
-3.33333
```

generates in the output file mulfloat

```
6.28318190574646e-02
1.086424827575684
3.74816512e+09
89.6999969482422
-4.333328455686569e-03
```

fwrite function

Writes n records of m bytes each to a specified file.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;  
char    *buffer;  
int      numrec, numbyt, numwrit;  
...  
numwrit = fwrite( buffer, numbyt, numrec, fd );
```

where

buffer is a byte pointer to a user-provided buffer

numrec is the number of records to write

numbyt is the number of bytes for each record

fd is a user-defined output buffer

numwrit is the number of records actually written

The include file `stdio.h` defines this function.

Since the hardware representations for byte and word pointers differ, `stdio.h` defines `fwrite` as a macro which converts the `databuf` pointer to a byte pointer.

Returns

The function returns the number of whole records written (0 for an error that wrote nothing).

Example

```
-----
/* program test for the fwrite() function */

#include <stdio.h>
#define MAXBYT 64

FILE    *fdi, *fdo, *fopen();
char *typei, *typeo, stringbuf[MAXBYT];
int     atoi(), numread, numrec, numbyt, numwrit;

main( argc, argv )

    int     argc;
    char    *argv[];
{
    typei = "r";
    fdi = fopen( argv[1], typei );
    typeo = "w";
    fdo = fopen( argv[2], typeo );
    numrec = atoi( argv[3] );
    numbyt = atoi( argv[4] );

    numread = fread( &stringbuf, numbyt, numrec, fdi );
    numwrit = fwrite( stringbuf, numbyt, numrec, fdo );

    printf("You specified %d records of %d bytes.\n",
           numrec, numbyt );
    printf("Number of records actually read = %d.\n",
           numread );
    printf("Number of records actually written = %d.\n",
           numwrit );
}
-----
```

A call to the program test with

```
x test fin fout 2 3
```

where the input file fin contains

```
9876543210
```

generates in the output file fout

```
987654
```

and at the terminal

```
You specified 3 records of 2 bytes.
Number of records actually read = 3.
Number of records actually written = 3.
```

getc macro

Reads a single character from a file.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;  
int      c;  
...  
c = getc(fd);
```

where

c = next character, or -1 if an error or end of file occurred

Since getc is a macro, it produces more in-line code than the corresponding call to fgetc, but the code executes faster. Since it is evaluated more than once, fd must not cause side effects. You may not pass getc as a parameter to a function or take its address, since it is a macro and not a function. Use fgetc instead.

If you define the macro `NEXPAND`, the `stdio.h` macros `getc` and `putc` resolve to the library functions `fgetc` and `fputc`. This allows you to set a breakpoint at the routine while debugging.

Example

```
/* program test for the getc() function */  
  
#include <stdio.h>  
#define EOF      -1  
  
FILE    *fd, *fopen();  
char     *type;  
int      c;  
  
main( argc, argv )  
  
    int      argc;  
    char     *argv[];
```

```

{
    type = "r";
    fd = fopen( argv[1], type );
    while( ( c = getc( fd ) ) != EOF )
        putchar( c );
}

```

A call to the program test with

```
x test input_str
```

replicates the file input_str at the terminal:

```
This is a test input file.
```

getchar macro

Reads a single character from the standard input.

Calling Sequence

```
#include <stdio.h>
```

```
int    c;  
...  
c = getchar();
```

where

c = next character, or -1 if an error occurred

Since getchar is a macro, it produces more in-line code than the corresponding call to fgetc with argument stdin, but the code executes faster. You may not pass getchar as a parameter to a function or take its address, since it is a macro and not a function. Use fgetc instead.

If you define the macro NEXPAND, the stdio.h macros getc and putc resolve to the library functions fgetc and fputc. This allows you to set a breakpoint at the routine while debugging.

Example

```
/* program test for the getchar() macro */  
  
#include <stdio.h>  
#define EOF      '\n'  
  
char    c;  
  
main( argc, argv )  
  
    int    argc;  
    char   *argv[];  
  
    {  
        putchar( c = '?' );  
        while ( c != EOF )  
        {  
            putchar( c = ' ' );  
            putchar( c = getchar() );  
        }  
    }
```

If you call the program test and respond to the prompt with

? Stretch me out.

you generate the output

S t r e t c h m e o u t .

```
*****
getdir                                     function
Returns a byte pointer to the pathname of the current working
directory.
*****
```

Calling Sequence

```
char    *cwd;
...
cwd = getdir();
```

where

cwd is a byte pointer to the current working directory
pathname

The include file dglib.h defines this function.

Example

```
/* program test for the getdir() function */

#include <stdio.h>
#include <dglib.h>

char    *cwd;

main()
{
    cwd = getdir();
    printf("\tYour current working directory is\n%s\n", cwd );
}
```

A call to the program test generates the output

```
        Your current working directory is
:UDD3:NITRAM:C:CDOC
```

```
*****
getenv                                         function

Gets a pointer to environment variables.
*****
```

Calling Sequence

```
char    *name, *value, *getenv();
...
value = getenv( name );
```

where

name is a byte pointer to a null-terminated string
containing the environment variable to search for

The getenv function gets a pointer to environment variables
that are passed with the UNIX emulated routines `execle` and `execve`,
which either initiate a son process (if the `vfork` routine is
called first), or kill the current process to chain to a new
process.

If both the caller of `execle/execve` and the new target
process are C programs (the `.PR` file is of type `?FUNX`, which is
set automatically under AOS/VS 2.00 with `CCL.CLI`), the environment
vector is passed from father to son, and the son process can use
`getenv` to assess the environment vector.

When the father process is not of type `?FUNX` (that is, the `CLI`),
the only environment variable is `LOGNAME`, which gives the user
name.

Returns

The function returns a `NULL` pointer if the variable was not
found. Otherwise, it returns a byte pointer to a null-terminated
string holding the value of the environment value.

```
*****
getfp                                     $builtin function
```

Returns the frame pointer of the father's environment.

```
*****
```

Calling Sequence

```
-----
```

```
int      *getfp(), *curfp;
...
```

or

```
$builtin int      *getfp();
short int        *curfp;
...
curfp = getfp();
```

The first version uses a procedure. The second may generate smaller or faster code.

Example

```
-----
```

```
/* program test for the getfp() function */

int      *getfp(), *curfp;

main()
{
    printf("My frame pointer = %o.\n", curfp = getfp() );
}
```

A call to the program test generates the output

```
My frame pointer = 16000012650.
```

getlogin function

Returns a byte pointer to the user name.

Calling Sequence

```
char    *name, *getlogin();  
...  
name = getlogin( );
```

The pointer returned is in the static area.

Returns

If the standard input for the process is not a terminal, the function returns a NULL pointer.

gets function

Reads from the standard input file until it encounters a NEW
LINE.

Calling Sequence

```
char    *stringbuf, *gets();  
...  
gets(stringbuf);
```

where

stringbuf is a pointer to a user-provided character array

This function removes NEW LINE from the line, unlike fgets, which includes it. This routine is provided strictly for compatibility with previous versions of the C language. You may wish to use the fgets function instead, because the bounds of the array may be exceeded.

The include file stdio.h defines this function.

Returns

The function returns NULL (0) on end of file or error; otherwise it returns the pointer to the buffer.

Related Functions

See also the getc, getchar, fgets functions.

Example

```
-----  
        /* program test for the gets() function */  
  
#include <stdio.h>  
#define STRMAX 80  
  
char    stringbuf[STRMAX], *gets();  
int      last, strlen();  
  
main()  
{  
  
    printf("? ");  
    gets( stringbuf );  
    printf("Length of your string = %d.\n",  
           last = strlen( stringbuf ) );  
  
}
```

A response to the prompt of

? This string may be a little longer than you thought.

generates the output

Length of your string = 52.


```
Retrieves the value stored at the WSB (wide stack base).
*****
```

The first version uses a procedure. The second may generate smaller or faster code.

A call to the program test generates the output

Value at WSB = 16000012504.

```
*****
gets1                                     $builtin function
```

Retrieves the value of the current WSL (wide stack limit).

Calling Sequence

```
short int      *getsl(), *curwsl;
...
```

or

```
$builtin short *getsp();
short int      *curswl;
...
curswl = gets1();
```

The first version uses a procedure. The second may generate smaller or faster code.

Example

```

/* program test for the gets1() function */

#include <stdio.h>
#include <dglib.h>

short int      *curwsl, *gets1();

main()
{
    printf("Value at WSL = %o.\n", curwsl = gets1() );
}

```

A call to the program `test` generates the output

Value at WSL = 16001743672.

getsp

\$builtin function

Retrieves the value of the WSP (wide stack pointer).

Calling Sequence

```
        short int      *getsp(), *cursp;
        ...
or
        $builtin short  *getsp();
        short int      *cursp;
        ...
        cursp = getsp();
```

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

Example

```
        /* program test for the getsp() function */

#include <stdio.h>
#include <dglib.h>

        short int      *curwsp, *getsp();

main()
{
    printf("Value at WSP = %o.\n", curwsp = getsp() );
}
```

A call to the program test generates the output

Value at WSP = 16000012570.

```
*****
getw                                                    function
```

Gets an integer value from the input buffer.

```
*****
```

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;
int      num;
...
num = getw( fd );
```

where

fd is a pointer to an I/O packet

num is an integer variable

This function assumes that the file is open.

The include file `stdio.h` defines this function.

Returns

The function returns EOF on end of file, but since that is a legal integer, use `feof` to test for end of file.

Related Functions

See also the `getc`, `gets`, `putw` functions.

Example

```
/* program test for the getw() function */

#include <stdio.h>
#include <dglib.h>

FILE    *fd, *fopen();
char     *type;
int      num, sum = 0, c = 0;
```

```

main( argc, argv )

    int     argc;
    char    *argv[];

{
    type = "r";
    fd = fopen( argv[1], type );
    for( ;; )
    {

        num = getw( fd );
        if( feof(fd) )
            break;

        sum += num;
        c++;

    }

    printf("%d = sum of integers in %s.\n", sum, argv[1] );
    printf("Number of integers:  %d\n", c );

}

```

A call to the program test with

x test input_nos

where the file input_nos contains binary representations of the first six digits generates the output

```

21 = sum of integers in input_nos.
Number of integers:  6

```

hypot function

Finds the hypotenuse of a right triangle.

Calling Sequence

```
#include <math.h>

double  hypot(), x, y, z;
...
z = hypot( x, y );
```

Example

```
/* program test for the hypot() function */

#include <stdio.h>
#include <math.h>

double  atof(), x, y, z, hypot();

main( argc, argv )

    int      argc;
    char     *argv[];

{
    x = atof( argv[1] );
    y = atof( argv[2] );
    printf("Hypotenuse = %f.\n", z = hypot( x, y ) );
}
```

A call to the program test with

x test 3.3 4.4

generates the output

Hypotenuse = 5.500000.

infos function

Make an INFOS II request for the user.

Calling Sequence

```
#include <packets:infos.h>
```

```
int    err, infos();  
...  
err = infos( &packet, request );
```

where

packet is an appropriate INFOS II packet

request is the number of an INFOS request

For details, consult the "INFOS II System User's Manual (AOS/VS)"

The include file packets:infos.h defines INFOS II packets.

Returns

The function returns 0 if no error occurred, and otherwise the INFOS II error code.

Example

You may examine a complete example of the infos function in the subdirectory C:EXAMPLES under the filename build_isam.c.

insert_after function

Inserts an element into a queue after another queue element.

Calling Sequence

```
#include <queue.h>
```

```
struct queue que;  
struct element *ptr, *elem;  
...  
ptr = insert_after(&que, ptr, elem);
```

This function inserts an element after a specific element in a doubly-linked list. The hardware representation for queues is used. For more detail, consult "The ECLIPSE MV/8000 Principles of Operation." The first two double words of the element are reserved to hold the forward and backward pointers.

Returns

The pointer inserted is returned.

insert_before function

Inserts an element into a queue before another queue element.

Calling Sequence

```
#include <queue.h>

struct queue *que;
struct element *ptr, *elem;
...
ptr = insert_before(&que, ptr, elem);
```

This function inserts an element before a specific element in a doubly-linked list. The hardware representation for queues is used. For more detail, consult "The ECLIPSE MV/8000 Principles of Operation." The first two double words of the element are reserved to hold the forward and backward pointers.

Returns

The pointer inserted is returned.

insert_head function

Inserts an element at the head of a queue.

Calling Sequence

#include <queue.h>

```
    struct queue  que;  
    struct element *ptr;  
    ...  
    ptr = insert_head(&que, ptr);
```

This function inserts an element into the head of a doubly-linked list. The hardware representation for queues is used. For more detail, consult "The ECLIPSE MV/8000 Principles of Operation." The first two double words of the element are reserved to hold the forward and backward pointers.

Returns

The pointer inserted is returned.

insert_tail function

Inserts an element at the tail of a queue.

Calling Sequence

```
#include <queue.h>
```

```
    struct queue  que;  
    struct element *ptr;  
    ...  
    ptr = insert_tail(&que, ptr);
```

This function inserts an element into the tail of a doubly-linked list. The hardware representation for queues is used. For more detail, consult "The ECLIPSE MV/8000 Principles of Operation." The first two double words of the element are reserved to hold the forward and backward pointers.

Returns

The pointer inserted is returned.

```
*****
intso                                     function
```

Returns 1 if the standard output file is a terminal.

```
*****
```

Calling Sequence

```
int    intso(), isterm;
...
isterm = intso();
```

Example

```
/* program test for the intso() macro */

#include <stdio.h>

int    isterm, intso();

main()
{
    printf("Is standard output a terminal?\n");
    printf("%s.\n", ( isterm = intso()) == 1?"Yes":"No" );
}
```

A call to the program test generates the output

```
Is standard output a terminal?
Yes.
```

intss function

Returns a 1 if the standard input file is a terminal.

Calling Sequence

```
int    intss(), isterm;  
...  
isterm = intss();
```

Example

```
/* program test for the intss() macro */  
  
#include <stdio.h>  
  
int    isterm, intss();  
  
main()  
{  
  
    printf("Is standard input a terminal?\n");  
    printf("%s.\n", ( isterm = intss()) == 1?"Yes":"No" );  
  
}
```

A call to the program test generates the output

```
Is standard input a terminal?  
Yes.
```

```

*****
isalnum                                     macro

Returns nonzero if character is alphanumeric.
*****

```

Calling Sequence

```

#include <stdio.h>

    or

#include <ctype.h>

    int    c, result;
    ...
    result = isalnum( c );

```

The isalnum macro is the same as isalphanum.

Alphanumerics are 0...9, A...Z, and a...z.

You may get unexpected results if you try to redeclare this macro.

Example

```

/* program test for the isalnum() macro */

#include <stdio.h>

int    i = 1, result;

main( argc, argv )

    int    argc;
    char    *argv[];

{

    while ( i < argc )
    {

        printf("Is character %c alphanumeric? ", argv[i][0] );
        printf("%s.\n", ( result = isalnum( argv[i][0] ) )
            == 0?"No":"Yes" );
        i++;
    }
}

```

A call to the program test with

x test 9 . p *

generates the output

Is character 9 alphanumeric?	Yes.
Is character . alphanumeric?	No.
Is character p alphanumeric?	Yes.
Is character * alphanumeric?	No.

```
*****
isalpha                                          macro
```

Returns nonzero if character is alphabetic.

```
*****
```

Calling Sequence

```
-----
#include <stdio.h>
```

or

```
#include <ctype.h>
```

```
int    c, result;
...
result = isalpha( c );
```

Alphabets are A...Z and a...z.

You may get unexpected results if you try to redeclare this macro.

Example

```
-----
/* program test for the isalpha() macro */

#include <stdio.h>

int    i = 1, result;

main( argc, argv )

    int    argc;
    char    *argv[];

{
    while ( i < argc )
    {
        printf("Is character %c alphabetic? ", argv[i][0] );
        printf("%s.\n", ( result = isalpha( argv[i][0] )
                                == 0?"No":"Yes" ));
        i++;
    }
}
```


A call to the program test with

x test a B 3 d

generates the output

Is character a alphabetic?	Yes.
Is character B alphabetic?	Yes.
Is character 3 alphabetic?	No.
Is character d alphabetic?	Yes.

```
*****
isalphanum                                     macro

Returns nonzero if character is alphanumeric.
*****
```

Calling Sequence

```
-----
#include <stdio.h>
```

```
    or
```

```
#include <ctype.h>
```

```
    int    c, result;
    ...
    result = isalphanum( c );
```

isalphanum is the same as isalnum.

Alphanumerics are 0...9, A...Z, and a...z.

You may get unexpected results if you try to redeclare this macro.

Example

```
-----
/* program test for the isalphanum() macro */

#include <stdio.h>

int    i = 1, result;

main( argc, argv )

    int    argc;
    char    *argv[];

{
    while ( i < argc )
    {
        printf("Is character %c alphanumeric? ", argv[i][0] );
        printf("%s.\n", ( result = isalphanum( argv[i][0] ))
            == 0?"No":"Yes" );
        i++;
    }
}
```

A call to the program test with

x test & 7 \ g

generates the output

Is character & alphanumeric?	No.
Is character 7 alphanumeric?	Yes.
Is character \ alphanumeric?	No.
Is character g alphanumeric?	Yes.

```
*****
isascii                                     macro
```

Returns nonzero if argument is an ASCII character or EOF.

```
*****
```

Calling Sequence

```
#include <stdio.h>

    or

#include <ctype.h>

    int    c, result;
    ...
    result = isascii( c );
```

Since `isascii` is a macro, and the argument is evaluated more than once in the macro, it should not contain side effects.

You may get unexpected results if you try to redeclare this macro.

Example

```
/* program test for the isascii() macro */

#include <stdio.h>

int    i = 1, result;

main( argc, argv )

    int    argc;
    char    *argv[];

{
    while ( i < argc )
    {
        printf("Is %c an ascii character? ", argv[i][0] );
        printf("%s.\n", ( result = isascii( argv[i][0] ))
            == 0?"No":"Yes" );
        i++;
    }
}
```

A call to the program test with

x test 0 p :

generates the output

Is 0 an ascii character? Yes.
Is p an ascii character? Yes.
Is : an ascii character? Yes.

```

*****
isatty                                     function

Returns a 1 if the I/O file is using a terminal for I/O.
*****

```

Calling Sequence

```

#include <stdio.h>

FILE    *iop;
int      isatty(), isterm;
...
isterm = isatty( fileno( iop ) );

```

Example

```

/* program test for the isatty() function */

#include <stdio.h>

FILE    *fd;
int      isterm, isatty();

main()
{
    printf("Is standard I/O the terminal?\n");
    printf("%s.\n",
        ( isterm = isatty( fileno( stdin )))
        == 1?"Yes":"No" );
}

```

A call to the program test gets the response

```

Is standard I/O the terminal?
Yes.

```

isctrl macro

Returns nonzero if argument is a control character.

Calling Sequence

#include <stdio.h>

or

#include <ctype.h>

```
int    c, result;  
...  
result = isctrl( c );
```

Control characters are '\0'...' \037' and '\177'.

You may get unexpected results if you try to redeclare this macro.

Example

```
/* program test for the isctrl() macro */  
  
#include <stdio.h>  
  
int    i = 1, result;  
main( argc, argv )  
  
    int    argc;  
    char    *argv[];  
  
    {  
  
        while ( i < argc )  
        {  
  
            printf("Is %c a control character? ", argv[i][0] );  
            printf("%s.\n", ( result = isctrl( argv[i][0] ) )  
                == 0?"No":"Yes" );  
            i++;  
        }  
    }
```

A call to the program test with

x test 9 ^R "

generates the output

Is 9 a control character? No.

Is a control character? Yes.

Is " a control character? No.

(The CLI does not write out the control character.)


```

*****
isdigit                                         macro

Returns nonzero if character is the ASCII representation
of (0 ... 9).
*****

```

Calling Sequence

```

#include <stdio.h>

    or

#include <ctype.h>

    int    c, result;
    ...
    result = isdigit( c );

```

Digits are 0...9.

You may get unexpected results if you try to redeclare this macro.

Example

```

/* program test for the isdigit() macro */

#include <stdio.h>

int    i = 1, result;

main( argc, argv )

    int    argc;
    char    *argv[];

{

    while ( i < argc )
    {

        printf("Is %c a digit? ", argv[i][0] );
        printf("%s.\n", ( result = isdigit( argv[i][0] ) )
            == 0?"No":"Yes" );
        i++;
    }
}

```

A call to the program test with

x test 6 A x 9

generates the output

Is 6 a digit? Yes.
Is A a digit? No.
Is x a digit? No.
Is 9 a digit? Yes.

ishex macro

Results in nonzero if character is hexadecimal.

Calling Sequence

```
#include <stdio.h>
```

or

```
#include <ctype.h>
```

```
int    c, result;  
...  
result = ishex( c );
```

Hexadecimals are 0...9 and a...f or A...F.

The ishex macro is the same as the isxdigit macro.

You may get unexpected results if you try to redeclare this macro.

Example

```
/* program test for the ishex() macro */  
  
#include <stdio.h>  
  
int    i = 1, result;  
  
main( argc, argv )  
  
    int    argc;  
    char   *argv[];  
  
    {  
  
        while ( i < argc )  
        {  
  
            printf("Is %c a hexadecimal digit? ", argv[i][0] );  
            printf("%s.\n", ( result = ishex( argv[i][0] ) )  
                == 0?"No":"Yes" );  
            i++;  
  
        }  
  
    }
```

A call to the program test with

x test A f g H

generates the output

Is A a hexadecimal digit? Yes.
Is f a hexadecimal digit? Yes.
Is g a hexadecimal digit? No.
Is H a hexadecimal digit? No.

isline macro

Returns nonzero if character is a line terminator.

Calling Sequence

```
#include <stdio.h>

    or

#include <ctype.h>

    int    c, result;
    ...
    result = isline( c );
```

Line terminators are NEW LINE, form feed, and carriage return (CR).

Returns

You may get unexpected results if you try to redeclare this macro.

Example

```
/* program test for the isline() macro */

#include <stdio.h>

int    i = 1, result;

main( argc, argv )

    int    argc;
    char    *argv[];

{

    while ( i < argc )
    {

        printf("Is %c a line terminator? ", argv[i][0] );
        printf("%s.\n", ( result = isline( argv[i][0] ) )
            == 0?"No":"Yes" );

        i++;

    }

}
```

A call to the program test with

x test [!ascii 273]

generates the output

Is ; a line terminator? No.

islower macro

Returns nonzero if character is lowercase.

Calling Sequence

```
#include <stdio.h>
```

or

```
#include <ctype.h>
```

```
int    c, result;  
...  
result = islower( c );
```

Lowercase characters are a...z.

You may get unexpected results if you try to redeclare this macro.

Example

```
/* program test for the islower() macro */  
  
#include <stdio.h>  
  
int    i = 1, result;  
main( argc, argv )  
  
    int    argc;  
    char    *argv[];  
  
    {  
  
        while ( i < argc )  
        {  
  
            printf("Is %c a lowercase character? ", argv[i][0] );  
            printf("%s.\n", ( result = islower( argv[i][0] ) )  
                == 0?"No":"Yes" );  
            i++;  
        }  
    }
```

A call to the program test with

x test Z z ? /

generates the output

Is Z a lowercase character? No.
Is z a lowercase character? Yes.
Is ? a lowercase character? No.
Is / a lowercase character? No.


```
*****
ispunct                                     macro
```

Returns nonzero if character is a punctuation mark.

```
*****
```

Calling Sequence

```
-----
```

```
#include <stdio.h>
```

or

```
#include <ctype.h>
```

```
int      c, result;
...
return = (ispunct( c ));
```

Punctuation marks, for this macro, are '.', ',', ';', ':', '!', '?', ' '.

You may get unexpected results if you try to redeclare this macro.

Example

```
-----
```

```
/* program test for the ispunct() macro */

#include <stdio.h>
#define EOF      -1

FILE      *fd, *fopen();
char      *type;
int       c, result;

main( argc, argv )
{
    int     argc;
    char    *argv[];

    {
        type = "r";
        fd = fopen( argv[1], type );
        while ( ( c = fgetc( fd ) ) != EOF )
        {
            printf("Is %c a punctuation character? ", c );
            printf("%s.\n", ( result = ispunct( c ) )
                == 0?"No":"Yes" );
        }
    }
}
```

A call to the program test with

with x test in_punct

where the file in_punct contains just

/.,;:-!

generates the output

Is / a punctuation character?	No.
Is . a punctuation character?	Yes.
Is , a punctuation character?	Yes.
Is ; a punctuation character?	Yes.
Is : a punctuation character?	Yes.
Is - a punctuation character?	No.
Is ! a punctuation character?	Yes.

isspace macro

Returns nonzero if the argument is a space character.

Calling Sequence

```
#include <stdio.h>
```

or

```
#include <ctype.h>
```

```
int    c, result;  
...  
result = isspace( c );
```

Space characters are space, TAB, NEW LINE, form feed, and carriage return (CR).

Returns

This function does not return true for '\000'.

You may get unexpected results if you try to redeclare this macro.

Example

```
/* program test for the isspace() macro */  
  
#include <stdio.h>  
#define EOF    -1  
  
FILE    *fd, *fopen();  
char    *type;  
int     c, result;  
  
main( argc, argv )  
  
    int     argc;  
    char    *argv[];  
  
    {  
  
        type = "r";  
        fd = fopen( argv[1], type );
```

```

while ( ( c = fgetc( fd )) != EOF )
{
    printf("Is %c a space character? ", c );
    printf("%s.\n", ( result = isspace( c ))
        == 0?"No":"Yes" );
}
}

```

A call to the program test with

x test in_space

where the file in_space contains just

```

1 2    3
4

```

generates the output

```

Is 1 a space character? No.
Is  a space character? Yes.
Is 2 a space character? No.
Is  a space character? Yes.
Is 3 a space character? No.
Is
a space character? Yes.
Is 4 a space character? No.

```

isupper macro

Returns nonzero if character is uppercase.

Calling Sequence

```
#include <stdio.h>
```

or

```
#include <ctype.h>
```

```
int    c, result;  
...  
result = isupper( c );
```

Uppercase characters are A...Z.

You may get unexpected results if you try to redeclare this macro.

Example

```
/* program test for the isupper() macro */  
  
#include <stdio.h>  
  
int    i = 1, result;  
main( argc, argv )  
    int    argc;  
    char   *argv[];  
{  
    while ( i < argc )  
    {  
        printf("Is %c an uppercase character? ", argv[i][0] );  
        printf("%s.\n", ( result = isupper( argv[i][0] ) )  
            == 0?"No":"Yes" );  
        i++;  
    }  
}
```

A call to the program test with

x test % 5 ? /

generates the output

Is % an uppercase character? No.
Is 5 an uppercase character? No.
Is T an uppercase character? Yes.
Is t an uppercase character? No.

isxdigit macro

Results in nonzero if character is hexadecimal.

Calling Sequence

```
#include <stdio.h>
```

or

```
#include <ctype.h>
```

```
int    c, result;  
...  
result = isxdigit( c );
```

Hexadecimals are 0...9 and a...f or A...F.

The isxdigit macro is the same as the ishex macro.

You may get unexpected results if you try to redeclare this macro.

Example

```
/* program test for the isxdigit() macro */
```

```
#include <stdio.h>
```

```
int    i = 1, result;
```

```
main( argc, argv )
```

```
int    argc;  
char   *argv[];
```

```
{
```

```
while ( i < argc )  
{
```

```
    printf("Is %c a hexadecimal digit? ", argv[i][0] );  
    printf("%s.\n", ( result = isxdigit( argv[i][0] ))  
            == 0?"No":"Yes" );  
    i++;
```

```
}
```

```
}
```

A call to the program test with

x test a % F H

generates the output

Is a a hexadecimal digit? Yes.
Is 5 a hexadecimal digit? Yes.
Is F a hexadecimal digit? Yes.
Is H a hexadecimal digit? No.

itoa function

Converts an integer to a character string.

Calling Sequence

```
int    num;  
char   *string;  
...  
itoa( num, string );
```

where

num is of type int

string is a byte pointer to a character array of at least
13 characters

The string contains a minus sign if num is negative, at most 11
digits, and terminates in a null.

The include file stdio.h defines this function.

The include file stdio.h also defines a macro L_itoa which
is replaced by 13, the maximum number of characters itoa returns.

Example

```
/* program test for the itoa() function */  
  
#include <stdio.h>  
#define STRINGMIN 13  
  
int    i = 1, num;  
char   string[STRINGMIN];  
  
main( argc, argv )  
  
    int    argc;  
    char   *argv[];  
  
{
```

```

while ( i < argc )
{
    num = atoi( argv[i] );
    num *= i;
    itoa( num, string );
    printf("%s\n", string );
    i++;
}

```

A call to the program test with

```
x test 1 2 3 4
```

generates the output

```

1
4
9
16

```

lasterror function

Returns the code for the most recent error.

Calling Sequence

```
int    lasterror(), err;  
...  
err = lasterror();
```

Example

```
/* program test for the lasterror() function */  
  
#include <stdio.h>  
  
int    err, lasterror();  
  
main( argc, argv )  
  
    int    argc;  
    char   *argv[];  
  
{  
  
    printf("Last error = %d.\n", err = lasterror() );  
  
}
```

A call to the program test generates the output

Last error = 0.

ldexp function

Returns a value times the nth power of 2.

Calling Sequence

```
#include <math.h>

double ldexp(), result, value, exp;
...
result = ldexp( value, exp );
```

Example

```
/* program test for the ldexp() function */

#include <stdio.h>
#include <math.h>

double result, value, exp2, atof(), ldexp();
int i = 1;

main( argc, argv )
    int argc;
    char *argv[];
{
    while ( i+1 < argc )
    {
        value = atof( argv[i] );
        exp2 = atof( argv[i+1] );
        printf("Value %f to power %f of 2 = %f\n",
            value, exp2, result = ldexp( value, exp2 ) );
        i += 2;
    }
}
```

A call to the program test with

x test 7.7 6.6 5.5 4.4 3.3 2.2

generates the output

Value 7.700000 to power 6.600000 of 2 = 746.945123

Value 5.500000 to power 4.400000 of 2 = 116.116696

Value 3.300000 to power 2.200000 of 2 = 15.162818

```

*****
log                                     $builtin function

Returns the natural log of the number passed to it.
*****

```

Calling Sequence

```

        double m, n, log();
        ...
or
        $builtin double log();
        double m, n;
        ...
        m = log( n );

```

where

m and n are of type double

A natural log is a log to the base e, which has the approximate value 2.718....

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers. The include file math.h defines this entry, and uses the \$builtin version.

```

| Returns
| -----
|
| If this routine gets an illegal input, it returns 0.0
| and sets errno to EDOM (defined in errno.h).

```

Example

```
/* Program test for the log() function */

#include <math.h>

double  atof(), m, n, log();
int      i = 1;

main( argc, argv )

    int      argc;
    char     *argv[];

{
    while ( i < argc )
    {
        n = atof( argv[i] );
        printf("Natural log of %f is %f.\n", n, m = log( n ));
        i++;
    }
}
```

A call to the program test with

x test [input_nos]

where the file input_nos contains

```
2 &
2.718 &
10 &
1024
```

generates the output

```
Natural log of 2.000000 is 0.693147.
Natural log of 2.718000 is 0.999896.
Natural log of 10.000000 is 2.302585.
Natural log of 1024.000000 is 6.931471.
```

```

*****
log10                               $builtin function

Returns the log based 10 of the number passed to it.
*****

```

Calling Sequence

```

-----

double m, n, log10();
...
or
$builtin double log10();
double m, n;
...
m = log10( n );

```

where

m and n are of type double

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers. The include file math.h defines this entry, and uses the \$builtin version.

```

| Returns
| -----
|
| If this routine gets an illegal input, it returns 0.0
| and sets errno to EDOM (defined in errno.h).

```


Example

```
/* Program test for the log10() function */

#include <math.h>

double  atof(), m, n, log10();
int     i = 1;

main( argc, argv )
        int     argc;
        char    *argv[];
{
    while ( i < argc )
    {
        n = atof( argv[i] );
        printf("Log to base 10 of %f is %f.\n", n, m = log10( n ));
        i++;
    }
}
```

A call to the program test with

x test [input_nos]

where the file input_nos contains

```
2 &
2.718 &
10 &
1024
```

generates the output

```
Log to base 10 of 2.000000 is 0.301029.
Log to base 10 of 2.718000 is 0.434249.
Log to base 10 of 10.000000 is 1.000000.
Log to base 10 of 1024.000000 is 3.010299.
```

longjmp function

Performs a nonlocal jump in the flow of control.

Calling Sequence

```
#include <setjmp.h>

int    val, save[8];
...
longjmp( &save, val );
```

To use this function, you must first call the setjmp function, which saves the return address following the setjmp() location. The longjmp function, called in a procedure nested arbitrarily deeply, retrieves successive frame pointers until it recovers the location following the matching setjmp function.

Execution continues as if the call to setjmp had just returned the value val to the function that called setjmp. That function must not have returned in the meantime. All accessible register variables and local data have values as of the time the longjmp function is called.

The include file setjmp.h defines this entry.

Related Functions

See the setjmp function.

Example

```
/* Program test for the longjmp() function */

#include <setjmp.h>

jmp_buf label;
int count = 0;
```

```

main()
{
    printf("\nSetjmp returned %o\n\n", setjmp( label ));
    if( ++count == 3 )
        exit( 0 );
    recurse( 10 );
    printf("Error\7:  recurse should not have returned.\n");
}

recurse( number )
{
    printf("Recurring, level %2d\n", number );
    if( number > 0 )
        recurse( number - 1 );
    else    longjmp( label, 1 );

    printf("Error\7:  longjmp should not have returned.\n");
}

```

A call to the program test generates the output

Setjmp returned 0

```

Recurring, level 10
Recurring, level 9
Recurring, level 8
Recurring, level 7
Recurring, level 6
Recurring, level 5
Recurring, level 4
Recurring, level 3
Recurring, level 2
Recurring, level 1
Recurring, level 0

```

Setjmp returned 1

```

Recurring, level 10
Recurring, level 9
Recurring, level 8
Recurring, level 7
Recurring, level 6
Recurring, level 5
Recurring, level 4
Recurring, level 3
Recurring, level 2
Recurring, level 1
Recurring, level 0

```

Setjmp returned 1

main function

Enters a C program.

Calling Sequence

```
main( [argc, argv] )
    int    argc;
    char    *argv[];
    {
    .
    .
    .
    }
```

where

argc is the number of arguments, including the program name, with which you invoke the C program

argv[i] is a pointer to a character array representing the ith argument to the program

Since argv[0] is the program name, the value of argc is always at least 1. If you wish to manipulate these character arrays as something other than strings, you must make explicit use of such functions as atof or atoi.

The main function must be present to initiate a C program. The initializer calls main and returns to the system when main returns.

If your program does not process the command line arguments, you may enter the function simply as

```
main()
```

Under AOS/RT32, which does not accept command line arguments, argv[0] is dummy_prog.

Example

```
/* Program test for the main() function */

int    i = 1;

main( argc, argv )

    int    argc;
    char   *argv[];

{

printf("You called the program %s.\n", argv[0] );
while ( i < argc )
{

    printf("Argument %d for this run is %s.\n",
           i, argv[i] );
    i++;

}

}
```

A call to the program test with

x test alpha beta gamma

generates the output

```
You called the program test.
Argument 1 for this run is alpha.
Argument 2 for this run is beta.
Argument 3 for this run is gamma.
```

malloc function

Allocates blocks of memory, but does not zero them out.

Calling Sequence

```
char    *ptr, *malloc();  
int      size;  
...  
ptr = malloc( size );
```

where

size is the size in bytes of the object

Since hardware representations for pointers to characters differ from pointers to other types, use the type-change operation to convert the result to a pointer of the appropriate type, such as

```
ip = ( int * ) malloc( sizeof ( int ) );
```

The include file `stdio.h` defines this entry.

Returns

The function returns 0 if no core was allocated.

Related Functions

See also `alloc`, `calloc`.

Example

```
/* program test for the malloc() function */  
  
#include <stdio.h>  
  
char    *ptr, *malloc();  
int      size;
```

```

main( argc, argv )

    int     argc;
    char    *argv[];
{

    size = atoi( argv[1] );
    ptr = malloc( size );
    printf("size = %d bytes in the block %o.\n", size, ptr );

}

```

A call to the program test with an argument of 512 generates the output

```
size = 512 bytes in the block 34003702770.
```

(The address will probably vary.)

```
*****
max                                         $builtin function
```

Returns the integer maximum of two numbers.

```
*****
```

Calling Sequence

```
int      a, b, c, max();
...
```

or

```
$builtin int      max();
int      a, b, c;
...
a = max( b, c );
```

The first version uses a procedure. The second may generate smaller or faster code. The include file math.h defines this entry, and uses the \$builtin version.

Related Functions

See also the fmax, min functions.

Example

```
/* program test for the max() function */

#include <math.h>

int      i = 1, a, b, c, max();

main( argc, argv )

    int      argc;
    char     *argv[];
{
    while ( i+1 < argc )
    {
        b = atoi( argv[i+1] );
        printf("Maximum of %d and %d is %d.\n",
               a, b, c = max( a, b ) );
        i++;
    }
}
```


A call to the program test with

x test 64 16 32 8

generates the output

Maximum of 64 and 16 is 64.

Maximum of 16 and 32 is 32.

Maximum of 32 and 8 is 32.

```
*****
min                                     $builtin function
```

Returns the integer minimum of two numbers.

```
*****
```

Calling Sequence

```
int      a, b, c, min();
...
```

or

```
$builtin int  min();
int      a, b, c;
...
a = min( b, c );
```

The first version uses a procedure. The second may generate smaller or faster code. The include file math.h defines this entry, and uses the \$builtin version.

Related Functions

See also the max, fmin functions.

Example

```
/* program test for the min() function */

#include <math.h>

int      i = 1, a, b, c, min();

main( argc, argv )

    int      argc;
    char     *argv[];
{
    while ( i+1 < argc )
    {
        a = atoi( argv[i] );
        b = atoi( argv[i+1] );
        printf("Minimum of %d and %d is %d.\n",
               a, b, c = min( a, b ) );
        i++;
    }
}
```

A call to the program test with

x test 64 16 32 8

generates the output

Minimum of 64 and 16 is 16.

Minimum of 16 and 32 is 16.

Minimum of 32 and 8 is 8.

mktemp function

Returns a complete unused temporary filename.

Calling Sequence

```
char    *char6;  
...  
mktemp( char6 );
```

where

char6 is at least six characters long

The function replaces the last six characters with the PID number and a unique sequence of alphabetic characters.

Related Functions

See also the tmpfile, tmpnam functions.

modf function

Returns the fractional part of a value as a double quantity,
storing the integer part indirectly.

Calling Sequence

```
#include <math.h>

double modf(), frac, value, *mant;
...
frac = modf( value, mant );
```

The include file math.h defines this function.

Related Functions

See also the fmod function.

Example

```
/* program test for the modf() function */
```

```
#include <math.h>

double frac, value, *mant, atof(), modf();
int i = 1;

main( argc, argv )
{
    int argc;
    char *argv[];

    while ( i < argc )
    {
        value = atof( argv[i] );
        printf("Result on %f = %f\n", value,
              frac = modf( value, mant ) );
        i++;
    }
}
```

A call to the program test with

```
x test [input_nos]
```

where the file input_nos contains

```
3.1416 &  
2.718 &  
.0001 &  
4e-4
```

generates the output

```
Result on 3.141600 = 0.141600  
Result on 2.718000 = 0.718000  
Result on 0.000100 = 0.000100  
Result on 0.000400 = 0.000400
```

```
*****
Returns the number of arguments for the executing routine.
*****
```

or

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers.

See also the argc argument at the entry for the main function.

093-000264-01 Licensed Material - Property of Data General Corp. 10-203

```

        printf("Number of arguments to main = %d\n",
               argnum = nargs() );

        printf("Last argument to main is %s.\n", argv[argc-1] );
    }

    subord( a )

    char    a[];

    {

        printf("Argument to subord on call %d is %s.\n", i, a );
        printf("Number of arguments to subord = %d\n\n",
               argnum = nargs() );
    }

```

A call to the program test with

x test alpha beta gamma

generates the output

```

Argument to subord on call 1 is alpha.
Number of arguments to subord = 1

Argument to subord on call 2 is beta.
Number of arguments to subord = 1

Argument to subord on call 3 is gamma.
Number of arguments to subord = 1

Number of arguments to main = 3
Last argument to main is gamma.

```



```
*****
otoa                                     macro
```

```
Converts an integer into an octal character string.
*****
```

Calling Sequence

```
int      num;
char     *octnum;
...
otoa( num, octnum );
```

Example

```
/* program test for the otoa() macro */

int      i = 1, num;
char     octnum[12];

main( argc, argv )

    int     argc;
    char    *argv[];

{

    printf("\tDecimal\t\tOctal\n\n");
    while ( i < argc )
    {

        num = atoi( argv[i] );
        printf("\t%d\t=\t%s\n", num,
                otoa( num, octnum ) );
        i++;

    }

}
```

A call to the program test with

```
x test 66 77 88 99
```

generates the output

Decimal		Octal
66	=	102
77	=	115
88	=	130
99	=	143

pathname function

Returns a pointer to the complete pathname of a file or directory.

Calling Sequence

```
char    *fname, *pname, *pathname();  
...  
pname = pathname( fname );
```

where

fname and pname are pointers to char

pname points to the pathname you wish to complete

Example

```
/* program test for the pathname() function */  
  
int     i = 1;  
char    *pname, *pathname();  
  
main( argc, argv )  
  
    int     argc;  
    char    *argv[];  
  
    {  
  
        printf("\tPATHNAMES:\n\n");  
        while ( i < argc )  
        {  
  
            printf("%s\n", pname = pathname( argv[i] ) );  
            i++;  
  
        }  
  
    }
```

A call to the program test with

x test [input_str]

where the file input_str contains

f302640009.d1 &
test.c &
c.prelim

generates the output

PATHNAMES:

:UDD3:NITRAM:C:CDOC:F302640009.D1
:UDD3:NITRAM:C:CDOC:TEST.C
:UDD3:NITRAM:C:CDOC:C.PRELIM

```
*****
perror                                                    function

Prints a string and the last error message on the error file.
*****
```

Calling Sequence

```
char    *string;
...
perror( string );
```

Example

```
/* Program test for the perror() function */

#include <stdio.h>

FILE    *fopen();

main( argc, argv )
    int    argc;
    char    *argv[];
{
    if ( fopen( argv[1], "w" ) == NULL )
    {
        perror( argv[1] );
        exit();
    }
}
```

A call to the program test with

```
x test c.temp
```

when the file c.temp has permanence ON generates the message

```
c.temp:0324 CANNOT DELETE PERMANENT FILE
```

```

*****
pow                                     $builtin function

Returns the value of the first argument raised to the power of
the second argument.
*****

```

Calling Sequence

```

double m, n, o, pow();
...
or

```

```

$builtin double pow();
double m, n, o;
...
m = pow(n,o);

```

where

m, n o are of type double

m is the result of raising n to the power o

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers. The include file math.h defines this entry, and uses the \$builtin version.

```

| Returns
| -----
|
| If floating exceptions are raised, this function returns the
| largest possible floating point number and sets errno to ERANGE
| (defined in errno.h).

```

Example

```

/* program test for the pow() function */

#include <math.h>

int i = 2;
double m, n, o, atof(), pow();

```

```

main( argc, argv )

    int     argc;
    char    *argv[];

{

    n = atof( argv[1] );

    while ( i < argc )
    {

        o = atof( argv[i] );
        printf("%f\tto the power %f = %f\n", n, o,
               m = pow( n, o ) );
        i++;

    }

}

```

A call to the program test with

x test 1.999999 [input_nos]

where the file input_nos contains

```

10 &
5 &
2.5 &
1.25 &
0.625

```

generates the output

```

1.999999      to the power 10.000000 = 1023.994880
1.999999      to the power 5.000000  = 31.999920
1.999999      to the power 2.500000  = 5.656847
1.999999      to the power 1.250000  = 2.378412
1.999999      to the power 0.625000  = 1.542210

```

printf function

Writes to standard output.

Calling Sequence

#include <stdio.h>

printf(format, arg1, ...)

where

each argi is a valid C variable or constant

format is a string containing two kinds of objects:

ordinary characters that the function copies directly

conversion specifications that start with '%'

The function replaces the format specifications with the successive argi, and writes the result to the file.

After the '%' character come specifications for precision, signs, and the like, which are optional but which you must enter in the following relative order:

Zero or more flag characters:

'\$' adds a floating dollar sign before d, e, E, f, g, and 'G' formats (a DGC feature)

'-' specifies left adjustment instead of right

'+' specifies that d, e, E, f, g, G formats have a sign, even if it is positive

(blank) specifies that if no sign is produced, a space prepends to the field (If '+' is also used, the blank is ignored.)

'#' specifies an alternate format: %o prepends a 0 (for octal), %x and %X prepend 0x and 0X respectively (for hexadecimal), and in floating conversions the decimal point always appears whether or not there are trailing digits.

A call to the program test with

```
x test 123 -45 6789
```

generates the output

Here are some ways to print out integers:

```
0000123 $123    123    +123    0173    0x7b
-000045 $-45    -45    -45    03777777723    0xffffffffd3
0006789 $6789   6789   +6789   015205   0x1a85
```

2. /* program test for the printf() function on floats */

```
#include <stdio.h>
```

```
double n, atof();
int i = 1;
```

```
main( argc, argv )
```

```
    int    argc;
    char    *argv[];
```

```
{
```

```
    printf("Here are some ways to print out");
    printf(" floating-point numbers:\n\n" );
    while ( i < argc )
    {
```

```
        n = atof( argv[i] );
```

```
        printf("%04f\t%e\t%.3f\n%#o\t%#x\t#g\n\n",
               n, n, n, n, n, n, n);
        i++;
```

```
    }
```

```
}
```

A call to program test with

```
x test .12345 12.345 1234.5
```

generates the output

Here are some ways to print out floating-point numbers:

0.123450	1.234500e-01	0.123
012054171174	0x401f9a6b	0.12345
12.345000	1.234500e+01	12.345
027024365605	0x41c5851e	12.345
1234.500000	1.234500e+03	1234.500
00	0x434d2800	1234.5

3. /* program test for the printf() function on strings */

```
#include <stdio.h>
```

```
int    ch1 = '\032', ch2 = '\132';
int    i = 1;
```

```
main( argc, argv )
```

```
    int    argc;
    char    *argv[];
```

```
{
```

```
    printf("Here are some ways to print out strings:\n\n" );
    while ( i < argc )
    {
```

```
        printf("%s\t", argv[i]);
        printf("%06s\t", argv[i]);
        printf("%.3s\n", argv[i]);
        printf("%5t%s\t", argv[i]);
        printf("%r\t",
            (ch1>040 && ch1<0176)?"%c":"\\%.3o", ch1 );
        printf("%r\n",
            (ch2>040 && ch2<0176)?"%c":"\\%.3o", ch2 );
        i++;
```

```
    }
```

```
}
```

A call to the program test with

x test Look at the last letters:

generates the output

Here are some ways to print out strings:

```
Look    00Look  Loo
      Look    \32    Z
at      0000at  at
      at  \32    Z
the     000the  the
      the \32    Z
last    00last  las
      last    \32    Z
letters:      letters:      let
      letters:  \32    Z
```

The second instance of each argument is extended by 0 to a length of 6. The third instance is truncated to a length of 3. The fourth is tabbed in to column 5. The first of the two characters is printed as an octal value because its value is less than 40.

putc macro

Writes a character to a given file.

Calling Sequence

#include <stdio.h>

```
FILE    *fd;  
char    nextchar;  
...  
putc(nextchar, fd);
```

where

nextchar is the character to write out

fd is a pointer to a FILE packet

Since this is a macro, the code is expanded in line. The code is faster than a function call such as fputc, though it takes a few more instructions and more code space. fd must not have any side effects. You may not pass putc as a parameter to a function or take its address, since it is a macro and not a function. Use fputc instead.

If you define the macro NEXPAND, the stdio.h macros getc and putc resolve to the library functions fgetc and fputc. This allows you to set a breakpoint at the routine while debugging.

Returns

The putc function returns the character or, if an error occurs, -1.

Related Functions

See also the getc, fputc, putchar, puts functions.

Example

```
/* program test for the putc() macro on strings */
```

```
#include <stdio.h>
```

```
FILE    *fdi, *fdo, *fopen();
```

```
int      c, i = 0;
```

```
main( argc, argv )
```

```
    int      argc;  
    char     *argv[];
```

```
{
```

```
    fdi = fopen( argv[1], "r" );  
    fdo = fopen( argv[2], "w" );  
    while ( ( c = getc( fdi ) ) != EOF )  
    {
```

```
        putc( c, fdo );  
        i++;
```

```
    }  
    printf("Number of bytes copied from %s to %s = %d",  
           argv[1], argv[2], i );
```

```
}
```

A call to the program test with

```
x test in_char out_char
```

generates the output

```
Number of bytes copied from in_char to out_char = 101
```

when the file in_char contains 101 bytes.

putchar macro

Writes a character to the standard output.

Calling Sequence

```
#include <stdio.h>
```

```
    char    nextchar;  
    ...  
    putchar( nextchar );
```

where

nextchar is the character to write out to the standard output

You may not pass putchar as a parameter to a function or take its address, since it is a macro and not a function. Use fputc instead.

Returns

The putchar macro returns the character written or, if an error occurred, -1.

Related Functions

See also the getchar, fputc, putc, puts functions.

Example

```
/* program test for the putchar() macro on strings */
```

```
#include <stdio.h>
```

```
FILE    *fd, *fopen();  
int      c, i = 0;
```

```
main( argc, argv )
```

```
    int    argc;  
    char   *argv[];
```

```
{
```

```
    fd = fopen( argv[1], "r" );  
    while ( ( c = getc( fd ) ) != EOF )  
    {
```

```
        putchar( c );  
        i++;
```

```
    }
```

```
    printf("Bytes copied from %s to standard output = %d",  
           argv[1], i );
```

```
}
```

A call to the program test with

```
x test to_term
```

generates the output

Please write this file to the terminal, if the terminal
is the standard output file.

Bytes copied from to_term to standard output = 85

when the file to_term contains just the sentence beginning with
"please".

```
*****
puts                                                    function
```

Writes to the standard output file until it encounters a NULL byte.

```
*****
```

Calling Sequence

```
char    *stringbuf, puts();
...
puts( stringbuf );
```

where

stringbuf is a pointer to a user-provided character array

The function appends a NEW LINE to the output characters, unlike fputs.

The include file stdio.h defines this function.

Related Functions

See also the gets, fputs functions.

Example

```
/* program test for the puts() macro on strings */
```

```
#include <stdio.h>
```

```
char    store[132], *strcpy();
int      i = 1;

main( argc, argv )
{
    int    argc;
    char   *argv[];
    while( i < argc )
    {
        strcpy( store, argv[i] );
        puts( store );
        i++;
    }
}
```


A call to the program test with

x test This is a complete listing of your strings.

generates the output

This
is
a
complete
listing
of
your
strings.

putw function

Puts an integer value into the output buffer.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;  
int      num;  
...  
putw( num, fd );
```

where

fd is a pointer to an I/O packet

num is an integer variable

Returns

The putw function returns num if there was no error, and
-1 if an error occurred.

Related Functions

See also the getw, putc, puts functions.

Example

```
/* program test for the putw() function */  
  
#include <stdio.h>  
  
FILE    *fd, *fopen();  
char     *type;  
int      sum = 0, c;  
  
main( argc, argv )  
  
        int      argc;  
        char     *argv[];
```

```

{

type = "w";
fd = fopen( argv[1], type );
for( c = 0 ; c < 6 ; c++ )
{

    putw( c, fd );
    sum += c;

}

printf("%d = sum of integers in %s.\n", sum, argv[1] );
printf("Number of integers:  %d\n", c++ );

}

```

A call to the program test with

x test input_nos

generates the output

```

15 = sum of integers in input_nos.
Number of integers:  6

```

```

*****
qsearch                                     function

Searches for an element in a queue meeting a given condition.
*****

```

Calling Sequence

```

#include <queue.h>

struct element *qsearch(), *ptr, *first;
struct queue   *header;
int            type, offset, value;
...
ptr = qsearch( header, type, offset, value, first );

```

where

header is a pointer to a queue header built with one of the
insert_ functions

ptr is the address of the element which either meets the search
condition or is 0 (NULL) if thte search condition fails

type is an integer value from 0 to 31 specifying the type of
search to perform (consult "The ECLIPSE MV/8000 () Principles
of Operation" manual for details)

offset is a value of the word offset offset from the beginning
of the structure whose value the search is to test against

value is an integer to test against the word at the offset in
each test

first is a pointer to the first element to be searched, or
-1 (QNULL) if the search is to start at the first element of the
list

Related Functions

See also the insert_head, insert_tail, insert_after, insert_before,
remove_head, remove_element, remove_tail functions.

```

*****
rand                                                    function

Creates and returns a random number in the range 0 to 32767.
*****

```

Calling Sequence

```

double rand();
int      m;
...
m = rand();

```

where

m is of type int

To be compatible with other C implementations, rand returns a number ≥ 0 and ≤ 32767 . The random number is based on a seed that is originally +1.

The include file math.h defines this entry.

Related Functions

See also the rand1, srand functions.

Example

```

/* program test for the rand() function */

#include <math.h>

int      result, i, c;

main( argc, argv )
{
    int      argc;
    char     *argv[];

    {
        c = atoi( argv[1] );
        printf("List of %d random numbers:\n\n", c );
        for ( i = 1; i<=c; i++ )
            printf("%d\n", result = rand() );
    }
}

```

A call to the program test with

x test 4

generates the output

List of 4 random numbers:

30819

12099

10924

22643

(These will probably vary.)

randl function

Creates and returns a random number in the range 0 to 1.

Calling Sequence

```
double m, randl();  
...  
m = randl();
```

where

m is of type double

The random number is based on a seed that is originally +1.

The include file math.h defines this entry.

Returns

The return value is ≥ 0.0 and < 1.0 .

Related Functions

See also the rand, srand functions.

Example

```
/* program test for the randl() function */  
#include <math.h>  
  
int    i, c;  
double result;  
  
main( argc, argv )  
  
    int    argc;  
    char   *argv[];  
  
    {  
        c = atoi( argv[1] );  
        printf("List of %d random numbers:\n\n", c );  
        for ( i = 1; i<=c; i++ )  
            printf("%f\n", result = randl() );  
    }
```

A call to the program test with

x test 4

generates the output

List of 4 random numbers:

0.242561

0.546138

0.541656

0.836353

realloc function

Changes the size of a previously allocated area.

Calling Sequence

```
char    *ptr, *realloc();
int      size;
...
ptr = realloc( ptr, size );
```

where

ptr is a pointer to the a previously allocated area

size is the number of bytes for the object

If size > current block size, a new pointer is allocated and the contents are copied to the new block. The old block is then freed up.

The include file stdio.h defines this function.

Related Functions

See also the alloc, alloca, calloc, malloc functions.

Example

```
/* program test for the realloc() function */

#include <stdio.h>

char    *ptr, *alloc(), *realloc();
int      size;

main(argc, argv)

    int    argc;
    char   *argv[];
```

```

{

size = atoi( argv[1] );
ptr = alloc( size );
printf("Size = %d bytes were in the block %o.\n", size, ptr );
size = atoi( argv[2] );
ptr = realloc( ptr, size );
printf("Size = %d bytes are now in the block %o.\n",
        size, ptr );

}

```

A call to the program test with

```
x test 512 1024
```

generates the output

```

Size = 512 bytes were in the block 34003702770.
Size = 1024 bytes are now in the block 34003674750.

```

remove_element function

Deletes an element from a queue.

Calling Sequence

```
#include <queue.h>
```

```
struct queue que;  
struct element *ptr;  
...  
ptr = remove_element(&que, ptr);
```

This function removes a specified element in a doubly-linked list. The hardware representation for queues is used. For more detail, consult "The ECLIPSE MV/8000 Principles of Operation." The first two double words of the element are reserved to hold the forward and backward pointers.

Returns

The pointer to the element is returned.

Related Functions

See also the remove_head, remove_tail functions.

```
*****
remove_head                                     function
```

Deletes an element from the head of a queue.

```
*****
```

Calling Sequence

```
#include <queue.h>

    struct queue  que;
    struct element *ptr;
    ...
    ptr = remove_head(&que);
```

This function removes an element from the head of a doubly-linked list. The hardware representation for queues is used. For more detail, consult "The ECLIPSE MV/8000 Principles of Operation." The first two double words of the element are reserved to hold the forward and backward pointers.

Returns

The pointer to the element is returned.

Related Functions

See also the remove_element, remove_tail functions.

remove_tail function

Deletes an element from the tail of a queue.

Calling Sequence

#include <queue.h>

```
struct queue que;  
struct element *ptr;  
...  
ptr = remove_tail(&que);
```

This function removes an element from the tail of a doubly-linked list. The hardware representation for queues is used. For more detail, consult "The ECLIPSE MV/8000 Principles of Operation." The first two double words of the element are reserved to hold the forward and backward pointers.

Returns

The pointer to the element is returned.

Related Functions

See also the remove_element, remove head functions.

repeat function

Replicates a string n times within a character array.

Calling Sequence

```
char    *repeat(), *result, *string, *ptr;
int      ntimes;
...
ptr = repeat( result, string, ntimes );
```

where

result is a pointer to a character string to receive the result

string is a pointer to a character string ending in the null character ('\0')

ntimes is an integer specifying the number of times to repeat the string in the result

Returns

The return value is the value of the result.

Example

```
/* program test for the repeat() function */
```

```
#define MAX 136
```

```
char    istring[MAX], ostring[MAX], *strcpy(), *repeat();
int      i;
```

```
main( argc, argv )
```

```
    int    argc;
    char   *argv[];
```

```

{
    strcpy( istring, argv[1] );
    i = atoi( argv[2] );
    printf("\tYou want to repeat\n%s\n\t%d times?",
           istring, i );
    repeat( ostring, istring, i );
    printf(" Here it is:\n%s\n", ostring );
}

```

A call to the program test with

x test Copy_me! 6

generates the output

```

    You want to repeat
Copy_me!
    6 times? Here it is:
Copy_me!Copy_me!Copy_me!Copy_me!Copy_me!Copy_me!

```

rew macro

Rewinds a file to the beginning.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;  
...  
rew(fd);
```

where

fd is a pointer to a file name

The include file stdio.h defines this macro as a call to the fseek function.

Returns

The macro returns -1 if an error occurred and 0 otherwise.

Example

```
/* program test for the rew() macro */
```

```
#include <stdio.h>
```

```
#define MAX    136
```

```
FILE    *fd, *fopen();  
char    stringbuf[MAX], *strcat();  
int      i, c;
```

```
main( argc, argv )
```

```
    int    argc;  
    char   *argv[];
```

```
{
```

```
    fd = fopen( argv[1], "r" );  
    c = atoi( argv[2] );
```



```

for( i = 1; i<= c; i++ )
{
    fgets( stringbuf, MAX, fd );
    printf("String %d: %s", i, stringbuf );
}

rew( fd );
fgets( stringbuf, MAX, fd );
printf("The first string was: %s\n", stringbuf );
}

```

A call to the program test with

```
x test input_str 3
```

where the file input_str contains

```

I am alpha.
I am beta.
I am gamma.
I am delta.
I am epsilon.

```

generates the output

```

String 1: I am alpha.
String 2: I am beta.
String 3: I am gamma.
The first string was: I am alpha.

```

since the file rewinds after the third call to fgets.

```
*****
scale10                                     function
```

Multiplies a number by a power of 10.

```
*****
```

Calling Sequence

```
-----
```

```
double  sn, num, scale10();
int      pow;
...
sn = scale10( num, pow );
```

where

sn and num are of type double

pow is of type int

The include file math.h defines this entry.

Example

```
-----
```

```
/* program test for the scale10() function */
```

```
#include <math.h>
```

```
int      i = 1, pwr;
double  sn, num, atof(), scale10();

main( argc, argv )
{
    int      argc;
    char     *argv[];

    {
        while ( i+1 < argc )
        {
            num = atof( argv[i++] );
            pwr = atoi( argv[i] );
            printf("%f times 10 to the %d = %f\n",
                num, pwr, sn = scale10( num, pwr ) );
            i++;
        }
    }
}
```

A call to the program test with

x test [input_nos]

where the file input_nos contains

```
.00007 5 &  
.006 4 &  
.5 3 &  
4 2
```

generates the output

```
0.000070 times 10 to the 5 = 7.000000  
0.006000 times 10 to the 4 = 60.000000  
0.500000 times 10 to the 3 = 500.000000  
4.000000 times 10 to the 2 = 400.000000
```

scanf function

Reads from standard input.

Calling Sequence

```
#include <stdio.h>
```

```
scanf(format, arg1 ... );
```

where

each argi is normally a pointer to a C variable or array name (if '@' is used, that argi would be an integer expression)

format is a string containing three kinds of objects:

spaces, NEW LINES, and TABs, which the function ignores

ordinary characters that must match the next nonspace character directly

conversion specifications that start with '%'

The function attempts to replace the format specifications with the successive argi, and read the result into the argi.

After the '%' character come three optional fields, which you must enter in the following relative order:

A comma ',' specifying that commas are to be treated like spaces

An asterisk '*' specifying that the number or string is to be scanned, but normal assignment omitted

A digit string specifying the maximum field width (If '@' occurs in place of the digit string (a DGC feature), take the field width from the next argument.)

'l' or 'L' indicating that the corresponding item is a long int (or double if 'e' or 'f' conversions) instead of the normal int (or float)

'h' or 'H' indicates that the next item is a short int instead of a normal int

A conversion character, specifying how to convert the input. If the character is not known, the input is expected to match that character. In support of other versions of C, an uppercase conversions character is equivalent to an 'l' or 'L' prefix. Characters marked with an asterisk (*) automatically skip blanks, TABs, and NEW LINES.

- *'d' Argument is a decimal integer.
- *'o' Argument is an octal integer (with or without a leading 0).
- *'x' Argument is a hexadecimal integer (with or without leading 0x).
- *'u' Argument is an unsigned int.
- 'c' Argument is a single character.
- *'s' Argument is a string.
- *'e' Argument is a floating-point number (= 'f').
- *'f' Argument is a floating-point number (= 'e').
- 'r' Do not read from input: argument is a nested format (a DG feature).
- [...] Brackets denote a string not delimited by spaces. If the first character is a caret (^), the string read matches all characters except those in the brackets (and end of file). If the first character is not ^, the string read matches only the characters in brackets. Some C environments may not support this feature.

Returns

The return value from *scanf is the number of items successfully matched and assigned, or -1 if an error occurred before the first match.

Related Functions

See also fscanf, sscanf, printf.

Example

1. /* program test for the scanf() function - strings */

```
#include <stdio.h>
#define MAX      136

int      i = 1, c;
char     string[MAX];

main( argc, argv )

    int      argc;
    char     *argv[];

{

    printf("Number of strings?\n");
    scanf("%d\n", &c );
    printf("Enter them:\n");
    while ( i =< c )
    {

        scanf("%s", &string );
        printf("%s%c", string, (i == c) ? '\n' : ' ');
        i++;

    }

}
```

If you call the program test, respond to the first query with 5, and to the second with

 There are five strings here.

you generate the output

 There are five strings here.

The scanf function skips spaces and TABs, and reads in the five character strings. The printf function places a space between each pair of strings it writes out, and a NEW LINE after the last string.

2. /* program test for the scanf() function - delimiters */

```
#include <stdio.h>
#define MAX      136

int      i = 0, c;
char     dlmstart[32] = "%[^", dlm[24], dlmend[4] = "]",
         *strcat(), string[MAX];

main( argc, argv )

    int      argc;
    char     *argv[];

{

    printf("How many strings?\n");
    scanf("%d", &c );
    printf("Pick characters for delimiters (max 24):\n");
    scanf("%s", &dlm );
    strcat( dlmstart, dlm );
    strcat( dlmstart, dlmend );
    printf("Enter %d strings using %s as delimiters:\n",
           c, dlm );
    while ( i < c )
    {
        scanf("%r", dlmstart, &string );
        printf("%s\n", string );
        scanf("%*ls", &string );
        i++;
    }
}
```

If you call the program test, respond to the first query with 5,
to the second with

!?,.

and to the third with

Greetings, user! Some data, please? Thank you.

you generate the output

How many strings?
Pick characters for delimiters (max 24):
Enter 5 strings using !?., as delimiters:

Greetings
user
Some data
please
Thank you

The program stores the characters you enter in the variable dlm, and uses the strcat function to create a %[^...] format, which serves as a delimiter table. The program replaces the %r format specification with the %[^...] format stored in the argument dlmstart. It reads in each string, writes the string out, and then reads past the current delimiter with the %*ls format.

3. /* program test for the scanf() function - integers */

```
#include <stdio.h>

int    dec, oct, hex;

main()

{

    printf("Enter decimal, octal, and hexadecimal numbers:\n");

    scanf("%d%o%x", &dec, &oct, &hex );

    printf("\n=>\tDecml\tOctal\tHexdec\n\n");

    printf("Decml\t%d\t%o\t%x\n", dec, dec, dec );
    printf("Octal\t%d\t%o\t%x\n", oct, oct, oct );
    printf("Hexdec\t%d\t%o\t%x\n", hex, hex, hex );

}
```

If you call the program test, and respond to the query with

1000 1000 1000

you generate the output

=>	Decml	Octal	Hexdec
Decml	1000	1750	3e8
Octal	512	1000	200
Hexdec	4096	10000	1000

4. /* program test for the scanf() function - floats */

```
#include <stdio.h>
```

```
int    i = 0, c;  
double d;
```

```
main()
```

```
{
```

```
    printf("How many floating-point entries?\n");  
    scanf("%d", &c );  
    printf("Please enter them:\n");  
    while ( i < c )  
    {
```

```
        scanf("%f", &d );  
        printf("%f\t= %g\n", d, d );  
        i++;
```

```
    }
```

```
    printf("Input Value\tShortest Representation\n");
```

```
}
```

If you call the program test and respond to the first query with 4 and to the second with

.000005 .005 50000 5000000

you generate the output

0.000005	= 5.e-06
0.005000	= 5.e-03
50000.000000	= 50000
5000000.000000	= 5.e+06
Input Value	Shortest Representation

setbuf function

Assigns a buffer to a specified file.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;  
char    bufptr[BUFSIZ];  
...  
setbuf( fd, bufptr );
```

or

```
setbuf( fd, NULL );
```

If the buffer pointer bufptr is NULL (0), it specifies an unbuffered file and uses single byte I/O. Otherwise the routine assumes that the printer can hold at least BUFSIZ characters. (The include file stdio.h defines BUFSIZ.)

setfp function

Sets the WFP (wide frame pointer).

Calling Sequence

```
short int      *curfp;  
...  
setfp( curfp );
```

In general, you should avoid this routine, as C uses the frame to hold the current activation record.

setjmp \$builtin function

Provides for nonlocal flow of control.

Calling Sequence

#include <setjmp.h>

```
    int    save[8];  
    ...
```

or

```
    $builtin int    setjmp();  
    int    save[8];  
    ...  
    setjmp( &save );
```

This function sets the conditions for performing a goto to a procedure nested at an arbitrary depth. You must call this function before performing the jump with the longjmp function, in order to set up the return address, which is the instruction following the location of setjmp().

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers. The include file setjmp.h defines this entry, and uses the \$builtin version.

Related Functions

See the longjmp function.

Example

Consult the example for the longjmp function.

setsb function

Sets the WSB (wide stack base).

Calling Sequence

```
short int      *cursb;  
...  
setsb( cursb );
```

In general, you should avoid this routine, since routines is
LANG_RT depend on the stack base being constant.

setscrn function

Sets screen characteristics.

Calling Sequence

#include <stdio.h>

```
FILE    *fd;  
int      setscrn(), flags, err;  
...  
err = setscrn( fd, flags );
```

where

fd is a pointer to a buffered I/O descriptor (normally
'stdin')

flags is an integer value to be set in the ?ESFC field of the
screen-read packet (use the include file paru.h to define any
bits you use)

the function result is 0 if the flags were set, and -1 if not

The file is flushed before returning. The next time I/O is done,
the flag value will be used.

This function has no effect if the file is a disk file or a
hardcopy terminal.

setsl function

Sets the WSL (wide stack limit).

Calling Sequence

```
short int      *cursl;  
...  
setsl( cursl );
```

In general, you should avoid this routine, since the memory allocation routines in LANG_RT reset the stack limit.

```
*****
setsp                                     function
```

Sets the WSP (wide stack pointer).

```
*****
```

Calling Sequence

```
short int      cursp;
```

```
...
```

```
setsp( cursp );
```

In general, you should avoid this routine, since you might wipe out some of your local variables. If you need to allocate more words on the stack, use the `alloca` function.

setxy function

Sets the cursor position for the next screen read.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *fd;  
int      setxy(), x, y, err;  
...  
err = setxy( fd, x, y );
```

where

fd is a pointer to a buffered I/O descriptor (normally
'stdin')

the variables x and y give the column and row in which to set
the cursor the next time I/O is done

This function has no effect if the file is a disk file or a
hardcopy terminal.

Returns

The return value is -1 if an error occurs or the file is a disk file,
and 0 otherwise.

Example

```
/* Program test for the setxy() function */
```

```
#include <stdio.h>
```

```
FILE    *fd;  
int      x, y, err, setxy();  
char     string[80];
```

```

main()
{
    fd = stdin;
    printf("Column?\n");
    scanf("%d", &x );
    printf("Row?\n");
    scanf("%d", &y );
    if (( err = setxy( fd, x, y )) == -1 )
        printf("Error:  cannot position.\n");
    else
    {
        printf("Input?\n");
        scanf("%s", &string );
        printf("Read '%s' at Column %d, Row %d.\n",
            string, x, y );
    }
}

```

A call to the program test with a clear screen generated

```

) x test
Column?
16
Row?
7
Input?

```

```

Where _am _I?
Read 'Where _am _I?' at Column 16, Row 7.

```

signcmp function

Compares successive character values of two strings.

Calling Sequence

```
char    *firststring, *secstring;
int      result;
...
result = signcmp(firststring, secstring);
```

where

firststring and secstring are pointers to strings to compare

result gets

```
-1      if firststring < secstring
0       if firststring == secstring
+1      if firststring > secstring
```

The scan ends when it encounters unequal characters, or both strings are exhausted (that is, the character is '\0').

The include file stdio.h defines this function.

The comparison is made on characters as signed values.

Related Functions

See also signncmp, strncmp, strcmp.

Example

```
/* Program test for the signcmp() function */

#include <stdio.h>

int      result;
char     first[80], second[80];
```

```

main()
{
    printf("Enter first string:\n");
    scanf("%s", &first );
    printf("Enter second string:\n");
    scanf("%s", &second );
    result = signcmp( first, second );
    switch ( result )
    {
        case -1:      printf("First, '%s', is earlier.\n", first );
                      break;
        case 0:       printf("Strings are equal.\n");
                      break;
        case 1:       printf("Second, '%s', is earlier.\n", second );
    }
}

```

Successive calls to the program test with the entries

jack
Jack

Jack
Jill

Jill.1
Jill.2

generate the outputs

Second, 'Jack', is earlier.

First, 'Jack', is earlier.

First, 'Jill.1', is earlier.

signncmp function

Compares successive character values of two strings.

Calling Sequence

```
char    *firststring, *secstring;
int      max, result;
...
result = signncmp( firststring, secstring, max );
```

where

firststring and secstring are pointers to strings to compare

max specifies the maximum number of characters to scan

result gets

```
-1      if firststring < secstring
0       if firststring == secstring
+1      if firststring > secstring
```

The scan ends when it encounters unequal characters, or both strings are exhausted (that is, the character is '\0', or when max characters have been scanned).

The comparison is made on characters as signed values.

The include file stdio.h defines this function.

Related Functions

See also signcmp, strncmp, strcmp.

Example

```
-----

/* Program test for the signncmp() function */

#include <stdio.h>
#define MAX      80

int      result;
char     first[MAX], second[MAX];

main()

{

    printf("Enter first string:\n");
    scanf("%s", &first );
    printf("Enter second string:\n");
    scanf("%s", &second );
    result = signncmp( first, second, MAX );
    switch ( result )
    {

        case -1: printf("First, '%s', is earlier.\n", first );
                  break;
        case 0:  printf("Strings are equal.\n");
                  break;
        case 1:  printf("Second, '%s', is earlier.\n", second );

    }

}
```

Successive calls to the program test with the entries

alpha
GAMMA

alpha
gamma

gamma
gamma

generate the outputs

Second, 'GAMMA', is earlier.

First, 'alpha', is earlier.

Strings are equal.

sin \$builtin function

Returns the sine of the number passed to it.

Calling Sequence

```
double m, n, sin();  
...  
or  
  
$builtin double sin();  
double m, n;  
...  
m = sin(n);
```

where

m and n are of type double

n is in radians

The function expects an expression in radians.

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers. The include file math.h defines this entry, and uses the \$builtin version.

Example

```
/* Program test for the sin() function */  
  
#include <math.h>  
  
int i = 1;  
double m, n, atof(), sin();  
  
main( argc, argv )  
  
    int argc;  
    char *argv[];  
  
    {
```

```

while ( i < argc )
{
    n = atof( argv[i] );
    printf("Sin of %lf = %lf\n",
           n, m = sin( n ) );
    i++;
}
}

```

A call to the program test with

100 75 50 25

generates the output

```

Sin of 100.000000 = -0.506365
Sin of 75.000000 = -0.387781
Sin of 50.000000 = -0.262374
Sin of 25.000000 = -0.132351

```

sinh \$builtin function

Returns the hyperbolic sine of the number passed to it.

Calling Sequence

```
double m, n, sinh();  
...  
or
```

```
$builtin double sinh();  
double m, n;  
...  
m = sinh(n);
```

where

m and n are of type double

m gets the hyperbolic sine of n

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers. The include file math.h defines this entry, and uses the \$builtin version.

Example

```
/* Program test for the sinh() function */  
  
#include <math.h>  
  
int i = 1;  
double m, n, atof(), sinh();  
  
main( argc, argv )  
  
    int argc;  
    char *argv[];  
  
    {  
  
        while ( i < argc )  
        {  
            n = atof( argv[i] );  
            printf("Hyperbolic sin of %lf = %lf\n",  
                n, m = sinh( n ) );  
            i++;  
        }  
    }
```

A call to the program test with

.1 .75 .50 .25

generates the output

Hyperbolic sin of 0.100000 = 0.100166
Hyperbolic sin of 0.750000 = 0.822316
Hyperbolic sin of 0.500000 = 0.521095
Hyperbolic sin of 0.250000 = 0.252612

sleep function

Delays a task for a given number of seconds.

Calling Sequence

```
int    hold;  
...  
sleep( hold );
```

where

hold specifies the number of seconds to delay the task

Example

```
/* Program test for the sleep() function */  
  
int    hold;  
  
main()  
{  
  
    printf("How long a nap?\n");  
    scanf("%d", &hold );  
    sleep( hold );  
    printf("I'm awake again.\n");  
  
}
```

A call to the program test generates

I'm awake again.

after 5 seconds if you respond to the query with 5.

sprintf function

Writes to an internal string.

Calling Sequence

```
#include <stdio.h>
```

```
    char    *string;  
    ...  
    sprintf( string, format, arg1, ...)
```

where

string is a pointer to a character array

each argi is a valid C variable or constant

format is a string containing two kinds of objects:

ordinary characters that the function copies directly

conversion specifications that start with '%'

The function replaces the format specifications with the successive argi, and writes the result to the string. For details of conversion specification, see the printf function.

Related Functions

See also fprintf, printf, sscanf.

Example

```
    /* Program test for the sprintf() function */  
  
#include <stdio.h>  
#define MAX    24  
#define LIM    20  
  
    char    store[MAX][LIM];  
    int     i = 1;
```

```

main( argc, argv )

    int    argc;
    char    *argv[];

{
    while ( i < argc )
    {
        sprintf( store[i], "%s\n", argv[i] );
        i++;
    }
    while ( i-- > 0 )
        printf("%s", store[i] );
}

```

A call to the program test with

```
x test [cities]
```

where the file cities contains

```

Albany &
Atlanta &
Baltimore &
Boston &
Chicago &
Cleveland &
Dallas &
Denver &
Memphis &
Miami

```

generates the output

```

Miami
Memphis
Denver
Dallas
Cleveland
Chicago
Boston
Baltimore
Atlanta
Albany

```

```

*****
sqrt                                     $builtin function

Returns the square root of the number passed to it.
*****

```

Calling Sequence

```

double m, n,    sqrt();
...
or

$builtin double sqrt();
double m, n;
...
m = sqrt(n);

```

where

m and n are of type double

If you furnish a negative number to the sqrt function,
the program aborts.

The first version uses a procedure. The second may generate smaller
or faster code, but may not run under other C compilers. The include
file math.h defines this entry, and uses the \$builtin version.

Example

```

/* Program test for the sqrt() function */

#include <math.h>

double m, n, atof(), sqrt();
int i = 1;

main( argc, argv )
{
    int    argc;
    char   *argv[];
    while ( i < argc )
    {
        n = atof( argv[i] );
        printf("Square root of %lf = %lf.\n",
               n, m = sqrt( n ) );
        i++;
    }
}

```

A call to the program test with

x test [input_nos]

where the file input_nos contains

3.1416 &
2.7182 &
1.4142 &
0.0009

generates the output

Square root of 3.141600 = 1.772455.
Square root of 2.718200 = 1.648696.
Square root of 1.414200 = 1.189201.
Square root of 0.000900 = 0.030000.

```
*****
srand                                                    function
*****
```

Places a user-specified seed in both rand and randl.

```
*****
```

Calling Sequence

```
-----
```

```
int      m;
...
srand(m);
```

where

m is of type int

If you specify a positive seed, both rand and randl base the random numbers on the seed. If you specify a negative seed, both rand and randl use the current time as the seed. If you change the seed for either of rand or randl, you change it for the other.

The include file math.h defines this entry.

Example

```
-----
```

```
/* Program test for the sqrt() function */

#include <math.h>

int      seed, m, i = 0, c;

main()

{

    printf("Enter a seed:\n");
    scanf("%d", &seed );
    srand( seed );
    printf("How many numbers?\n");
    scanf("%d", &c );
    printf("%d random numbers using a seed = %d:\n\n",
           c, seed );
    while ( i < c )
    {
        printf("%d\n", m = rand() );
        i++;
    }
}
```


A call to the program test with entries 13 and 4 generates the output

4 random numbers using a seed = 13:

13338

2308

17310

17096

sscanf function

Reads from an internal string.

Calling Sequence

```
#include <stdio.h>
```

```
    char    *string;  
    ...  
    sscanf(string, format, arg1 ... );
```

where

string is a pointer to a character array

each argi is a pointer to a valid C variable or constant

format is a string containing three kinds of objects:

spaces, NEW LINES, and TABs, which the function ignores

ordinary characters that must match the next nonspace
character directly

conversion specifications that start with '%'

The function attempts to replace the format specifications with the
successive argi, and read the result from the string into the argi.
For details of conversion specification, see the scanf function.

Returns

The return value from sscanf is the number of items successfully
matched and assigned.

Related Functions

See also fscanf, scanf, sprintf.

Example

```
/* Program test for the sscanf() function */

#include <stdio.h>
#define MAX      5

FILE      *fd, *fopen();
char      instring[25], name[MAX][12], course[MAX][8];
double    grade[MAX];
int        i = 0, j = 0, k = 0, l = 0;

main( argc, argv )

    int      argc;
    char     *argv[];

{

    fd = fopen( argv[1], "r" );
    while ( i < MAX )
    {

        fgets( instring, sizeof(instring), fd );
        if( 3 != sscanf( instring, "%s %s %f", name[i],
                        course[i], &grade[i] ) )
            break;
        i++;
    }

    printf("\nNAMES\n\n");
    while ( j < MAX )
    {
        printf("%s\n", name[j] );
        j++;
    }

    printf("\nCOURSES\n\n");
    while ( k < MAX )
    {
        printf("%s\n", course[k] );
        k++;
    }

    printf("\nGRADES\n\n");
    while ( l < MAX )
    {
        printf("%f\n", grade[l] );
        l++;
    }
}
```

A call to the program test with

x test classrec

where the file classrec contains

Brown	Algebra	2.5
Jones	Biology	1.6
Roberts	English	3.4
Smith	Physics	3.2
Wilson	Spanish	3.7

generates the output

NAMES

Brown
Jones
Roberts
Smith
Wilson

COURSES

Algebra
Biology
English
Physics
Spanish

GRADES

2.500000
1.600000
3.399999
3.199999
3.699999

strcat
function

Concatenates one ASCII string onto the tail of another.

Calling Sequence

```
char    *mainstring, *adjunct, *strcat();  
...  
strcat(mainstring, adjunct);
```

where

mainstring is the string to concatenate to

adjunct is the string to add to the tail of mainstring

Strings in C end in a NULL character. The delimiter for the string adjunct is NULL.

The include file stdio.h defines this function.

Returns

The return value is the value of mainstring.

Related Functions

See also strncat.

Example

```
/* Program test for the sscanf() function */  
  
#include <stdio.h>  
#define MAX      80  
  
char    string[MAX], *strcat();  
int      i = 1;
```

```

main( argc, argv )

    int    argc;
    char    *argv[];

{
    while ( i < argc )
    {
        strcat( string, argv[i] );
        printf("%s\n", string );
        i++;
    }
}

```

A call to the program test with

```
x test Just sit back and watch me grow!
```

generates the output

```

Just
Justsit
Justsitback
Justsitbackand
Justsitbackandwatch
Justsitbackandwatchme
Justsitbackandwatchmegrow!

```

strchr function

Searches for first occurrence of a character in a string.

Calling Sequence

```
char    *search, template, *strchr();  
...  
strchr(search, template);
```

where

search is the character array to inspect

template is the character you wish to match

The include file stdio.h defines this function.

Returns

The function returns a NULL if the character does not occur in the string, or the pointer to the byte found.

Related Functions

See also strrchar.

Example

```
/* Program test for the strchr() function */  
  
#include <stdio.h>  
#define MAX    80  
  
char    c, string[MAX], *strchr();  
int     i = 1, loc;  
  
main( argc, argv )  
  
    int     argc;  
    char    *argv[];
```

```

{
    printf("Character template?\n");
    scanf("%c", &c );
    while ( i < argc )
    {
        sprintf( string, "%s", argv[i] );
        if ( ( loc = strchr( string, c ) ) == 0 )
            printf("Character '%c' does not occur in \n\t'%s'\n",
                c, string );
        else
            printf("First occurrence of '%c' in\n\t'%s'\nat %o.\n",
                c, string, loc );
        i++;
    }
}

```

A call to the program test with a response of a to the query generates the output

```

First occurrence of 'a' in
    'alphabet'
at 34000023356.
First occurrence of 'a' in
    'syllabary'
at 34000023362.
Character 'a' does not occur in
    'string'

```

(The locations may vary with execution.)

strcmp function

Compares successive character values of two strings.

Calling Sequence

```
char    *firststring, *secstring;  
int     result;  
...  
result = strcmp(firststring, secstring);
```

where

firststring and secstring are pointers to strings to compare

result gets

```
-1      if firststring < secstring  
0       if firststring == secstring  
+1      if firststring > secstring
```

The scan ends when it encounters unequal characters, or both strings are exhausted (that is, the character is '\0').

The comparison is made on characters as unsigned values. If you use the /SIGNEDCHAR switch on the compiler, it creates a macro to replace sign(n)cmp calls with calls to str(n)cmp.

The include file stdio.h defines this function.

Related Functions

See also signcmp, signncmp, strncmp.

Example

```
/* Program test for the strcmp() function */

#include <stdio.h>
#define MAX      80

char    c = 'y', string1[MAX], string2[MAX];
int     result;

main( argc, argv )

    int     argc;
    char    *argv[];

{

    while ( c == 'y' )
    {

        printf("Enter first string:\n");
        scanf("%s", string1 );
        printf("Enter second string:\n");
        scanf("%s", string2 );

        result = strcmp( string1, string2 );
        switch ( result )
        {

            case -1: printf("'s' is less than 's'.\n",
                        string1, string2 );
                    break;
            case 0:  printf("The strings are equal.\n");
                    break;
            case 1:  printf("'s' is greater than 's'.\n",
                        string1, string2 );

        }

        printf("Another comparison? Type y: ");
        scanf("%s", &c );

    }

}
```

A call to the program test generates

```
Enter first string:
facts
Enter second string:
data
'facts' is greater than 'data'.
Another comparison? Type y: y
Enter first string:
LETTER
Enter second string:
letter
'LETTER' is less than 'letter'.
Another comparison? Type y: y
Enter first string:
13
Enter second string:
13
The strings are equal.
Another comparison? Type y: n
```

strcpy function

Copies one string onto another.

Calling Sequence

```
char    *dest, *source, *strcpy();  
...  
strcpy( dest, source );
```

where

dest is the string you wish to overwrite

source is the string you wish to copy

The function stops copying when it encounters a NULL byte.

The include file stdio.h defines this function.

Returns

The return value is dest.

Related Functions

See also strncpy.

Example

```
/* Program test for the strcpy() function */  
  
#include <stdio.h>  
#define MAX    80  
  
char    c = 'y', dest[MAX], source[MAX];  
  
main( argc, argv )  
  
    int    argc;  
    char    *argv[];
```

```

{
    while ( c == 'y' )
    {
        printf("Enter overwrite:\n");
        scanf("%s", source );
        printf("'s'\n\tbecomes\n", dest );
        printf("'s'\n", strcpy( dest, source ) );
        printf("Another overwrite? Type y: ");
        scanf("%s", &c );
    }
}

```

A call to the program test generates

```

Enter overwrite:
original_string
      becomes
'original_string'
Another overwrite? Type y: y
Enter overwrite:
replacement_string
'original_string'
      becomes
'replacement_string'
Another overwrite? Type y: n

```

strlen function

Determines the length of a character string.

Calling Sequence

```
char    *line;  
int      finis, strlen();  
...  
finis = strlen( line );
```

where

line is a pointer to the string to inspect

int receives the result of the count

Character strings in C end in a NULL. This function does not count NULL as part of the string.

Since the system separates ACL entries with a single NULL character, and strlen stops at the first NULL, the strlen function does not report the length of an ACL entry if you use one as the argument to strlen. For more information concerning the ACL field, consult your "Advanced Operating System/Virtual Storage (AOS/VS) Programmer's Manual".

Example

```
/* Program test for the strlen() function */

#include <stdio.h>
#define MAX      80

char    line[MAX];
int     i = 1, length;

main( argc, argv )

    int     argc;
    char    *argv[];

{

    while ( i < argc )
    {

        sprintf( line, "%s", argv[i] );
        printf("Length of argv[%d] = %d characters.\n",
               i, length = strlen( line ) );
        i++;
    }

}
```

A call to the program test with

x test Count these characters quickly!

generates the output

```
Length of argv[1] = 5 characters.
Length of argv[2] = 5 characters.
Length of argv[3] = 10 characters.
Length of argv[4] = 8 characters.
```

strncat function

Concatenates one ASCII string onto the tail of another.

Calling Sequence

```
char    *mainstring, *adjunct, *strncat();  
int      max;  
...  
strncat(mainstring, adjunct, max);
```

where

mainstring is a pointer to the string to concatenate to

adjunct is a pointer to the string to add to the tail of mainstring

max is the maximum number of bytes to move into mainstring

Strings in C end in a NULL character. The function does not count the null byte of adjunct as part of max.

The include file stdio.h defines this function.

Returns

The return value is mainstring.

Related Functions

See also strcat.

Example

```
/* Program test for the strncat() function */

#include <stdio.h>
#define MAX      80

char    string[MAX], *strncat();
int     i = 1;

main( argc, argv )

    int     argc;
    char    *argv[];

{

    while ( i < argc )
    {

        strncat( string, argv[i], MAX );
        printf("%s\n", string );
        i++;

    }

}
```

A call to the program test with

x test This string is getting longer and longer.

generates the output

```
This
Thisstring
Thisstringis
Thisstringisgetting
Thisstringisgettinglonger
Thisstringisgettinglongerand
Thisstringisgettinglongerandlonger.
```

strncmp function

Compares successive character values of two strings.

Calling Sequence

```
char    *firststring, *secstring;
int      max, result;
...
result = strncmp( firststring, secstring, max );
```

where

firststring and secstring are pointers to strings to compare

max specifies the maximum number of characters to scan

result gets

```
-1      if firststring < secstring
0       if firststring == secstring
+1      if firststring > secstring
```

The scan ends when it encounters unequal characters, or both strings are exhausted (that is, the character is '\0', or when max characters have been scanned).

The comparison is made on characters as unsigned values. If you use the /SIGNEDCHAR switch on the compiler, it creates a macro to replace sign(n)cmp calls with calls to str(n)cmp.

The include file stdio.h defines this function.

Related Functions

See also signcmp, signncmp, strcmp.

Example

```
/* Program test for the strncmp() function */
```

```
#include <stdio.h>
```

```
#define MAX      80
```

```
char    c = 'y', string1[MAX], string2[MAX];
int     result;
```

```
main( argc, argv )
```

```
    int     argc;
    char    *argv[];
```

```
{
```

```
    while ( c == 'y' )
    {
```

```
        printf("Enter first string:\n");
        scanf("%s", string1 );
        printf("Enter second string:\n");
        scanf("%s", string2 );
```

```
        result = strncmp( string1, string2, MAX );
        switch ( result )
        {
```

```
            case -1:      printf("'s' is less than 's'.\n",
                           string1, string2 );
                           break;
            case  0:      printf("The strings are equal.\n");
                           break;
            case  1:      printf("'s' is greater than 's'.\n",
                           string1, string2 );
```

```
        }
```

```
        printf("Another comparison? Type y: ");
        scanf("%s", &c );
```

```
    }
```

```
}
```

A call to the program test generates

```
Enter first string:
abcd
Enter second string:
ABCD
'abcd' is greater than 'ABCD'.
Another comparison? Type y:  y
Enter first string:
abcd
Enter second string:
bcde
'abcd' is less than 'bcde'.
Another comparison? Type y:  n
```

strncpy function

Copies one string onto another.

Calling Sequence

```
char    *dest, *source, *strncpy();  
int      max;  
...  
strncpy( dest, source, max );
```

where

dest is a byte pointer to the string you wish to overwrite

source is a byte pointer to the string you wish to replace
dest with

max is the maximum number of bytes of source to write

The function copies source to dest, ending when it has moved
max bytes or encountered a NULL. If it does not encounter a
NULL, it does not append one. If the string length is less than
max, the string is padded with NULLs.

The include file stdio.h defines this function.

Returns

The return value is dest.

Related Functions

See also strcpy.

Example

```
/* Program test for the strncpy() function */

#include <stdio.h>
#define MAX      80

char    c = 'y', dest[MAX], source[MAX];

main( argc, argv )

    int    argc;
    char   *argv[];

{
    while ( c == 'y' )
    {
        printf("Enter overwrite:\n");
        scanf("%s", source );
        printf("'%s'\n\tbecomes\n", dest );
        printf("'%s'\n", strncpy( dest, source, MAX ) );
        printf("Another overwrite? Type y: ");
        scanf("%s", &c );
    }
}
```

A call to the program test generates

```
Enter overwrite:
initial_line
''
    becomes
'initial_line'
Another overwrite? Type y: y
Enter overwrite:
next_line
'initial_line'
    becomes
'next_line'
Another overwrite? Type y: n
```

strnsave function

Allocates an area long enough to hold a string from the heap,
moves the string into the area, and returns the pointer to
the area.

Calling Sequence

```
char    *string, *newstring;  
int      n;  
...  
newstring = strnsave( string, n );
```

where

string is a byte pointer to a character array that
ends in NULL

newstring gets the byte pointer which is allocated
to receive a copy of the string

n specifies the maximum number of characters to copy

The include file stdio.h defines this function.

Related Functions

See also strsave.

Example

```
-----

/* Program test for the strnsave() function */

#include <stdio.h>
#define MAX      80

char    *newloc;
int      i = 1;

main( argc, argv )

    int      argc;
    char    *argv[];

{

    while ( i < argc )
    {

        newloc = strnsave( argv[i], MAX );
        printf("\tStored argv[%d] at %o.\n",
                i, newloc );
        printf("argv[%d] = '%s'\n", i, newloc );
        i++;

    }

}
```

A call to the program test with

x test Find some addresses.

generates the output

```
        Stored argv[1] at 34003703644.
argv[1] = 'Find'
        Stored argv[2] at 34003677500.
argv[2] = 'some'
        Stored argv[3] at 34003677344.
argv[3] = 'addresses.'
```

(The locations vary with execution.)


```
*****
strpbrk                                     function
```

```
Returns a pointer to the first character in both strings.
*****
```

Calling Sequence

```
char    *strpbrk(), *match, *string1, *string2;
...
match = strpbrk( string1, string2 );
```

Example

```
/* Program test for the strpbrk() function */

#include <stdio.h>

char    *match, *strpbrk();
main( argc, argv )

int      argc;
char     *argv[];

{

    match = strpbrk( argv[1], argv[2] );
    printf("First matching character in '%s' and '%s'",
           argv[1], argv[2] );
    printf(" is '%c'.\n", *match );

}
```

Two successive calls to the program test, the second with the arguments reversed, generate the outputs

```
First matching character in 'copy' and 'typo' is 'o'.
First matching character in 'typo' and 'copy' is 'y'.
```

strchr
function

Searches for the last occurrence of a character in a string.

Calling Sequence

```
char    *search, ultima, *strchr(), *result;  
...  
result = strchr(search, ultima);
```

where

search is the string to inspect

ultima is the character you wish to match

The include file stdio.h defines this function.

Returns

The function returns a NULL if the character does not occur in the string. Otherwise it returns a pointer to the last occurrence of ultima.

Related Functions

See also strchr.

Example

```
/* Program test for the strchr() function */  
  
#include <stdio.h>  
#define MAX    80  
  
char    c, string[MAX], *strchr();  
int     result, i = 1;
```

```

main( argc, argv )

    int    argc;
    char    *argv[];

{

    printf("Character template?\n");
    scanf("%c", &c );
    while ( i < argc )
    {

        sprintf( string, "%s", argv[i] );
        if ( ( result = strchr( string, c ) ) == NULL )
            printf("Character '%c' does not occur in \n\t'%s'\n",
                   c, string );
        else
            printf("Last occurrence of '%c' in\n\t'%s'\nat %o.\n",
                   c, string, result );
        i++;
    }

}

```

A call to the program test with

```
x test [input_str]
```

where the file input_str contains

```

element &
digital &
execute

```

generates the output

```

Character template?
e
Last occurrence of 'e' in
    'element'
at 34000023362.
Character 'e' does not occur in
    'digital'
Last occurrence of 'e' in
    'execute'
at 34000023364.

```

```
*****
strsave                                         function
*****
```

Allocates an area long enough to hold the string from the heap, moves the string into the area, and returns a pointer to the area.

```
*****
```

Calling Sequence

```
char    *string, *save, *strsave();
...
save = strsave(string);
```

where

string is a byte pointer to a character array, terminated by a null

save is a byte pointer to an allocated area large enough to contain a copy of the string

The include file `stdio.h` defines this function.

Related Functions

See also `strnsave`.

Example

```
/* Program test for the strsave() function */

#include <stdio.h>

char    *newloc;
int      i = 1;

main( argc, argv )

    int      argc;
    char    *argv[];
{
    while ( i < argc )
    {
        newloc = strsave( argv[i] );
        printf("\tStored argv[%d] at %o.\n",
               i, newloc );
        printf("argv[%d] = '%s'\n", i, newloc );
        i++;
    }
}
```

A call to the program test with

x test Save these strings.

generates the output

```
        Stored argv[1] at 34003703760.
argv[1] = 'Save'
        Stored argv[2] at 34003677730.
argv[2] = 'these'
        Stored argv[3] at 34003677704.
argv[3] = 'strings.'
```

(The locations vary with execution.)

```
*****
swab                                     function

Swaps successive bytes pairwise.
*****
```

Calling Sequence

```
char    *string1, *string2;
int      num;
...
swab( string1, string2, num );
```

where

string1 contains the bytes to be swapped

string2 contains the result of swapping

num is the number of bytes to swap

The number of bytes to swap, num, should be even.

Example

```
/* Program test for the swab() function */

#include <stdio.h>
#define MAX      80

char    string[MAX], *swab();
int      length, strlen();
int      i = 1;

main( argc, argv )

    int      argc;
    char    *argv[];

{
    while ( i < argc )
    {
        length = strlen( argv[i] );
        swab( argv[i], string, length );
        printf( "'%s' ==> '%.s'\n", argv[i], length, string );
        i++;
    }
}
```

A call to the program test with
x test Swap these bytes.
generates the output

'Swap' ==> 'wSpa'
'these' ==> 'htsee'
'bytes.' ==> 'ybet.s'

```
*****
sys                                                    function

Provides an interface to the operating system.
*****
```

Calling Sequence

For integer arguments

```
int      error, cmd, ac0, ac1, ac2, sys();
...
```

or

```
$builtin int    sys();
int      error, cmd, ac0, ac1, ac2;
...
error = sys(cmd, &ac0, &ac1, &ac2);
```

For pointer arguments

```
int      error, cmd, *ac0, *ac1, *ac2, sys();
...
```

or

```
$builtin int    sys();
int      error, cmd, *ac0, *ac1, *ac2;
...
error = sys(cmd, ac0, ac1, ac2);
```

where

cmd is a system command number

Before issuing the command, sys loads the values of ac0, ac1, and ac2 into the specified hardware registers.

After the system call, the updated values are stored back into ac0, ac1, ac2. Sys returns 0 if there was no error, or the error code (Ac0) if an error occurred. For more information about what is expected, consult your "AOS/VS Programmer's Manual". ac0, ac1, ac2 can be of types int or pointer.

If you are using the AOS/RT32 operating system, you must restrict your program to that subset of the system calls which is valid under that operating system. For details, consult your "AOS/RT32 System Programmer's Reference".

The include file sysid.h defines the system commands.

The first versions use a procedure. The second may generate smaller or faster code, but may not run under other C compilers. The include file dglib.h defines this entry, and uses the \$builtin version.

Example

```
/* Program test for the sys() function */

#include <stdio.h>
#include <sysid.h>
#define MAX      255
#define SPACE    "\040"

char    message[MAX], *acl;
int     i = 2, error, ac0, ac2;

main( argc, argv )

    int     argc;
    char    *argv[];

{

    ac0 = atoi( argv[1] );
    while ( i < argc )
    {

        strcat( message, argv[i] );
        if ( i+1 < argc )
            strcat( message, SPACE );
        i++;
    }
    acl = message;
    ac2 = strlen( acl );
    error = sys( $SEND, &ac0, &acl, &ac2 );
    printf("\nError = %d\n", error );
}
```

A call to the program test with

```
x test -l Can you read this?
```

generates the output

```
From Pid 48 : Can you read this?
Error = 0
```

Compare the sys_send example under the sys-<system-call-mnem> function.

```
*****
system                                         function

Executes a CLI command.
*****
```

Calling Sequence

```
system( "<CLI-command>" );
```

or

```
char    *cmd;
...
cmd = "<CLI-command>";
system( cmd );
```

where

cmd is a byte pointer to the command you wish the CLI to execute

If you are using the AOS/RT32 operating system, you must restrict your program to that subset of the CLI commands which is valid under that operating system. For details, consult your "AOS/RT32 Command Line Interpreter".

Returns

	The function returns	
	0	if no error occurred
	1	if an error occurred in any command
	2	if the routine could not get status back from the CLI
	127	If the routine could not initialize the CLI

Example

```
/* Program test for the system() function */  
  
#include <stdio.h>  
  
char    *cmd;  
  
main()  
{  
  
    cmd = "sysid";  
    system( cmd );  
  
}
```

A call to the program test generates

Crosley AOS/VS 01.64.91.06

which identifies the machine and revision number.

```

*****
sys_<system-call-mnemonic>                                function
*****
Makes an AOS/VS system call from a C program.
*****

```

Calling Sequence

```

int      ac0, ac1, ac2, *rac0, *rac1, *rac2;
...
sys_<sys-call-mnem>( ac0, ac1, ac2, rac0, rac1, rac2 );

```

where

system-call-mnemonic is the name of an AOS/VS system call
in lowercase less the question mark (?) prefix

ac0, ac1, ac2 are input values, if any, that you furnish to
accumulators AC0, AC1, AC2

rac0, rac1, rac2 are result values that the call generates,
if any, in accumulators AC0, AC1, AC2

To use this function successfully, you must place the library
csyscalls.lib in your LINK command line.

Table 10-2 lists the system calls available and the arguments
they take. If you are using the AOS/RT32 operating system,
you must restrict your program to that subset of the system
calls which is valid under that operating system. For details,
consult your "AOS/RT32 System Programmer's Reference".

Table 10-2. System Calls from C

An entry of		Means that				
X		the system call takes this argument. For details of use and purpose, consult your "Advanced Operating System/Virtual Storage (AOS/VS) Programmer's Manual".				
-		the system call does not take this argument.				
SYSTEM CALL	INPUT			RESULT		
	ac0	ac1	ac2	rac0	rac1	rac2
_allocate	-	X	X	-	X	-
_assign	X	-	-	-	-	-
_blkpr	X	X	-	-	-	-
_bname	X	-	-	X	-	-
_brkfl	X	X	X	-	-	-
_cday	X	-	-	X	X	X
_cgnam	X	X	X	-	-	X
_chain	X	X	X	-	-	X
_cisnd	X	X	X	-	-	-
_ckvol	X	X	X	-	-	-
_close	-	-	X	-	-	X
_con	X	X	-	X	-	-
_cpmax	X	X	-	-	-	-
_create	X	-	X	-	-	X
_cruda	X	-	X	-	-	-
_cterm	X	X	-	-	-	-
_ctod	X	-	-	X	X	X
_ctype	X	X	X	-	-	-
_dacl	X	X	-	X	-	-
_dadid	X	-	-	-	X	-
_dcon	-	X	-	-	-	-
_ddis	-	-	-	-	-	-
_deassign	X	-	-	-	-	-
_debl	-	-	-	-	-	-
_debug	-	-	-	-	-	-
_delete	X	-	-	-	-	-
_dfrsch	-	-	-	X	-	-
_dir	X	-	-	-	-	-
_dqtsk	-	-	X	-	-	-
_drcon	-	X	X	-	-	-
_drsch	-	-	-	-	-	-
_enbrk	X	X	X	-	-	-
_enqueue	X	X	-	-	-	-
_ermmsg	X	X	X	X	X	-
_ersch	-	-	-	-	-	-
_exec	-	-	X	-	-	X
_expo	X	X	X	-	-	-
_fday	X	X	X	X	-	-

(continues)

Table 10-2. System Calls from C

SYSTEM CALL	INPUT			RESULT		
	ac0	ac1	ac2	rac0	rac1	rac2
_fedit	X	X	X	X	X	X
_flush	X	-	-	-	-	-
_feov	X	-	-	-	-	-
_fstat	X	X	X	-	-	-
_ftod	X	X	X	X	-	-
_gacl	X	X	-	-	-	-
_gbias	-	-	-	X	-	-
_gchr	X	X	X	-	-	-
_gclose	-	X	-	-	X	-
_gcpn	X	X	-	-	X	-
_gday	-	-	-	X	X	X
_gechr	X	X	X	[X]	-	-
_gfname	X	X	X	X	X	X
_ghrz	-	-	-	X	-	-
_glink	X	X	-	-	-	-
_glist	-	X	X	-	-	-
_gmem	-	-	-	-	X	-
_gname	X	X	X	-	-	X
_gnfn	-	X	X	-	-	X
_gopen	X	X	X	X	-	-
_gport	-	X	-	-	X	X
_gpos	-	-	X	-	-	X
_gprnm	X	-	X	-	-	-
_grname	X	X	X	-	-	X
_gshpt	-	-	-	X	X	-
_gsid	-	-	X	-	-	-
_gtacp	X	X	-	X	-	-
_gtmes	-	-	X	X	X	X
_gtnam	X	X	X	X	-	-
_gtod	-	-	-	X	X	X
_gtruncate	-	X	X	-	-	-
_gtsv	-	X	X	X	-	-
_gtsvl	-	X	X	X	-	-
_gunm	X	X	X	X	X	-
_gvpid	X	X	-	X	-	-
_hname	X	X	-	X	X	X
_idef	X	X	X	-	-	-
_idgoto	X	X	-	-	-	-
_idkil	-	X	-	-	-	-
_idpri	X	X	-	-	-	-
_idrdy	-	X	-	-	-	-
_idsus	-	X	-	-	-	-
_ifpu	X	-	-	-	-	-
_ilkup	X	X	-	-	X	X
_imerge	X	-	X	X	-	-
_img	-	-	X	-	X	-
_init	X	X	X	-	-	-
_intwt	-	-	-	-	-	-

(continued)

Table 10-2. System Calls from C

SYSTEM CALL	ac0	INPUT acl	ac2	rac0	RESULT rac1	rac2
_iqtsk	X	X	-	-	-	-
_irec	-	-	X	-	-	-
_irmv	X	-	-	-	-	-
_isend	-	-	X	-	-	-
_is_r	-	-	X	-	-	-
_isplit	X	-	-	X	X	X
_itime	-	-	-	X	X	-
_ixit	-	X	-	-	-	-
_ixmt	-	X	X	-	-	-
_khist	-	-	-	-	-	-
_kilad	X	-	-	-	-	-
_kill	-	-	-	-	-	-
_kintr	X	X	X	X	X	X
_label	-	-	X	-	-	-
_lefd	-	-	-	X	-	-
_lefe	-	-	-	X	-	-
_lefs	-	-	-	X	-	-
_lmap	X	X	-	-	-	-
_logcalls	X	X	X	-	-	-
_logev	X	X	X	-	-	-
_ltfunc	-	X	-	-	-	-
_mbfc	-	-	X	-	X	-
_mbfu	X	X	X	X	X	X
_mbtc	-	-	X	-	X	X
_mbtu	X	X	X	X	X	X
_mdump	X	-	X	-	-	-
_mem	-	-	-	X	X	X
_memi	X	-	-	-	X	-
_mytid	-	-	-	X	X	-
_nabr	X	X	X	-	X	X
_nrnt	X	X	X	X	X	X
_nsint	X	X	X	X	X	X
_odis	-	-	-	-	-	-
_oebl	-	-	-	-	-	-
_open	-	-	X	-	-	-
_oper	X	X	X	-	-	-
_pcnx	X	X	-	-	-	-
_pname	X	X	-	-	X	-
_prdb	-	X	X	-	X	-
_pri	X	-	-	-	-	-
_pripr	X	X	X	-	-	-
_prkil	X	-	-	-	-	-
_proc	-	-	X	-	X	-
_prrdy	X	-	-	-	-	-
_prsus	X	-	-	-	-	-
_pstat	X	X	X	-	-	-
_pwr	-	X	X	-	X	-

(continued)

Table 10-2. System Calls from C

SYSTEM CALL	ac0	INPUT ac1	ac2	rac0	RESULT rac1	rac2
_rclose	X	X	X	X	X	X
_rdb	-	X	X	-	X	-
_rduda	X	-	X	-	-	-
_read	-	-	X	-	-	-
_rec	X	-	-	-	X	-
_recnw	X	-	-	-	X	-
_recreate	X	-	-	-	-	-
_release	X	-	-	-	-	-
_rename	X	X	-	-	-	-
_resched	-	-	-	-	-	-
_resign	-	-	-	-	-	-
_return	X	X	X	-	-	-
_ringld	X	-	-	-	-	-
_rngpr	X	X	X	-	-	-
_rngst	X	-	-	-	-	-
_rpage	X	-	-	-	-	-
_runtm	X	X	X	-	-	-
_sacl	X	X	-	-	-	-
_satr	X	X	-	-	-	-
_sbias	X	-	-	-	-	-
_schr	X	X	X	-	-	-
_sclose	-	X	-	-	-	-
_sday	X	X	X	-	-	-
_sdbl	-	X	-	-	-	-
_sdlm	X	X	X	-	-	-
_sdpol	X	X	X	-	-	-
_sdrtr	X	X	-	-	-	-
_sebl	X	X	-	-	-	-
_sechr	X	X	X	[X]	-	-
_send	X	X	X	-	-	-
_sert	X	X	-	-	-	-
_serve	-	-	-	-	-	-
_sges	X	X	X	-	-	-
_sgnl	-	X	X	X	X	X
_sigwt	-	X	X	-	X	-
_sinfo	-	-	X	-	-	-
_slist	-	-	X	-	-	-
_sopen	X	X	X	-	X	-
_sopff	X	X	X	-	X	-
_spage	-	X	X	-	-	-
_spos	-	-	X	-	-	-
_srcv	X	X	X	-	-	-
_sshpt	X	X	-	-	-	-
_ssid	-	-	X	-	-	-
_ssnd	X	X	X	-	-	-
_stmap	X	X	X	-	X	-

(continued)

Table 10-2. System Calls from C

SYSTEM CALL	INPUT			RESULT		
	ac0	ac1	ac2	rac0	rac1	rac2
stod	X	X	X	-	-	-
stom	X	X	X	-	-	-
suproc	X	-	-	X	-	-
sus	-	-	-	-	-	-
suser	X	-	-	X	-	-
swst	X	X	X	X	X	X
sylog	X	X	-	X	-	-
sysd	X	X	X	X	X	X
task	-	-	X	-	-	X
term	X	X	X	-	-	-
tidstat	-	X	-	X	-	-
tlock	X	-	X	X	-	X
tpid	X	-	-	X	X	-
tport	-	-	X	-	X	-
trcon	X	-	-	-	X	-
trun	-	-	X	-	-	-
truncate	-	-	X	-	-	-
tunlock	-	X	-	-	-	-
ublpr	X	X	-	-	-	-
uidcall	X	X	X	-	-	-
uidost	-	X	-	-	-	-
uidrtn	X	-	-	-	-	-
unwire	X	X	-	-	-	-
update	-	X	-	-	-	-
valad	X	X	-	-	X	-
vcust	X	-	-	-	-	-
vcust	X	-	X	-	-	-
wdelay	X	-	-	-	-	-
whist	X	X	X	-	-	-
wire	X	X	-	-	-	-
wrb	-	X	X	-	X	-
write	-	-	X	-	-	-
wruda	X	-	X	-	-	-
xmt	X	X	X	-	-	-
xmtw	X	X	X	-	-	-

(concluded)

Returns

The function returns 0 if no error occurred, an error code otherwise.

Examples

```
1.      /* Program test for the sys_<system-call-mnem>() function */

#include <sys_calls.h>
#include <stdio.h>

int      ac0, ac1, ac2, rac0, rac1, rac2;

main()
{
    printf("\nWhat time do you have in hours (24-hour clock),");
    printf(" minutes, seconds?\n");

    scanf("%d%d%d", &ac2, &ac1, &ac0);

    sys_ftod(ac0, ac1, ac2, &rac0);

    printf("That time is %d biseconds since midnight.\n", rac0);

    sys_ctod(ac0, &rac0, &rac1, &rac2);

    printf("The time is %d:%d:%d.\n", ac2, ac1, ac0);
}
```

A call to the program test at 12:15:45 generates the output

```
What time do you have in hours (24-hour clock), minutes, seconds?
That time is 22073 biseconds since midnight.
The time is 12:15:45.
```

The functions `sys_ftod` and `sys_ctod` utilize the AOS/VS system calls `?FTOD` and `?CTOD`. The program first converts your input to biseconds with the function `sys_ftod` and then reconverts it to hours, minutes, seconds with the function `sys_ctod`.

```

2.      /* Program test for the sys_send() function */

#include <sys_calls.h>
#include <stdio.h>
#define MAX      255
#define SPACE    "\040"

char      message[MAX];
int       i = 2, error;

main( argc, argv )

int       argc;
char      *argv[];

{

    while ( i < argc )
    {

        strcat( message, argv[i] );
        if ( i+1 < argc )
            strcat( message, SPACE );
        i++;

    }
    error = sys_send( atoi(argv[1]), /* Ac0 */
                     message,        /* Ac1 */
                     strlen(message));/* Ac2 */
    printf("\nError = %d.\n", error );

}

```

A call to the program test with

```
x test -1 Please disregard this message.
```

generates

```

From Pid 23 : Please disregard this message.
Error = 0.

```

Compare the example under the sys function.

```
*****
tail                                                    macro
```

Returns a pointer to the null byte within a string.

```
*****
```

Calling Sequence

```
-----
```

```
#include <stdio.h>
```

```
char    *string, *last;
...
last = tail(string);
```

where

string is a character array

*last points to the null byte within a the string

The include file stdio.h defines this function.

Example

```
-----
```

```
/* Program test for the tail() macro */
```

```
#include <stdio.h>
#define MAX    80
```

```
char    *last;
int      i = 1;
```

```
main( argc, argv )
```

```
int      argc;
char     *argv[];
```

```
{
```

```
    while ( i < argc )
    {
```

```
        printf("argv[%d] = '%s'", i, argv[i] );
        printf("\nbegins at %o and ends at %o.\n",
               argv[i], last = tail( argv[i] ) );
        i++;
```

```
    }
```

```
}
```

A call to the program test with

x test Where are these lines?

generates the output

```
argv[1] = 'Where'
begins at 34000025473 and ends at 34000025500.
argv[2] = 'are'
begins at 34000025501 and ends at 34000025504.
argv[3] = 'these'
begins at 34000025505 and ends at 34000025512.
argv[4] = 'lines?'
begins at 34000025513 and ends at 34000025521.
```

```
*****
tan                                          $builtin function
```

Returns the tangent of the number passed to it.

```
*****
```

Calling Sequence

```
double m, n, tan();
...
or

$builtin double tan();
double m, n;
...
m = tan(n);
```

where

m and n are of type double

n is an expression in radians

The function expects an expression in radians.

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers. The include file math.h defines this entry, and uses the \$builtin version.

```
| Returns
| -----
|
| If floating exceptions are raised, this function returns the
| largest possible floating point number and sets errno to ERANGE
| (defined in errno.h).
```

Related functions

See also the tanh function.

Example

```
/* Program test for the tan() function */

#include <math.h>

double m, n, atof(), tan();
int i = 1;

main( argc, argv )

int argc;
char *argv[];

{

    while ( i < argc )
    {

        n = atof( argv[i] );
        printf("Tangent of %lf = %lf.\n", n, m = tan( n ) );
        i++;

    }

}
```

A call to the program test with

x test [input_nos]

where the file input_nos contains

10 &
1 &
0.1 &
0.01

generates the output

Tangent of 10.000000 = 0.648360.
Tangent of 1.000000 = 1.557407.
Tangent of 0.100000 = 0.100334.
Tangent of 0.010000 = 0.010000.

```

*****
tanh                                $builtin function

Returns the hyperbolic tangent of the number passed to it.
*****

```

Calling Sequence

```

-----

double m, n, tanh();
...
or

$builtin double tanh();
double m, n;
...
m = tanh(n);

```

where

m and n are of type double

The first version uses a procedure. The second may generate smaller or faster code, but may not run under other C compilers. The include file math.h defines this entry, and uses the \$builtin version.

Related Functions

See also the tan function.

Example

```

-----

/* Program test for the tanh() function */

#include <math.h>

double m, n, atof(), tanh();
int i = 1;

main( argc, argv )

int argc;
char *argv[];

```



```

{
    while ( i < argc )
    {
        n = atof( argv[i] );
        printf("Hyperbolic tangent of %lf = %lf.\n",
               n, m = tanh( n ) );
        i++;
    }
}

```

}
 A call to the program test with

x test [input_nos]

where the file input_nos contains

```

10 &
1 &
0.1 &
0.01

```

generates the output

```

Hyperbolic tangent of 10.000000 = 0.999999.
Hyperbolic tangent of 1.000000 = 0.761594.
Hyperbolic tangent of 0.100000 = 0.099667.
Hyperbolic tangent of 0.010000 = 0.009999.

```

tmpfile function

Creates and opens a temporary file.

Calling Sequence

```
#include <stdio.h>
```

```
FILE    *tmpfile(), *fd;  
...  
fd = tmpfile();
```

where

fd returns a pointer to an open file descriptor of a file
that will be deleted at process termination

Other processes, including sons, will not be able to refer
to this file.

Related Functions

See also the mktemp, tmpnam functions.

tmpnam function

Returns a pointer to a temporary filename.

Calling Sequence

```
char    *tmpnam(), *name;  
...  
name = tmpnam();
```

where

the new filename has the form

?<pid>.CFILE.xxxxx.TMP<0>

and xxxxx is a unique integer

Related Functions

See also the mktemp, tmpfile functions.

```

| *****
| toascii macro
|
| Strip the parity bit from the argument.
| *****

```

```

| Calling Sequence
| -----

```

```

| #include <ctype.h>
|
|     int    inp, out;
|     ...
|     out = toascii( inp );

```

```

| This macro converts a native character set to the ASCII character
| set.

```

```

| Example
| -----

```

```

|         /* Program test for the toascii() macro */
|
| #nolist
| #include <stdio.h>
| #include <ctype.h>
| #list
|
|     int    inp, out;
|
|     main()
|     {
|
|         printf("Enter a small octal number from -041 to -176: ");
|         scanf("%o", &inp );
|         out = toascii( inp );
|         printf("Strips %o and converts to '%c', or octal %o.\n",
|               inp, out, out );
|
|     }

```

```

| If you call program test and enter

```

```

|     17777101

```

```

| you will see

```

```

|     Strips 1777701 and converts to 'A', or octal 101.

```

tolower macro

Maps uppercase alphabetic to lowercase alphabetic.

Calling Sequence

```
#include <stdio.h>

    or

#include <ctype.h>

    int    c, result;
    ...
    result = tolower(c);
```

This macro is not defined for arguments for which the `isascii` macro is false. Since the argument is evaluated more than once in the macro, it should not contain side effects.

Related Functions

See also `toupper`, `_toupper`, `_tolower`.

Example

```
/* Program test for the tolower() macro */

#include <stdio.h>

    int    c, result;

    main( argc, argv )

        int    argc;
        char   *argv[];

    {

        while ( ( c = getchar() ) != EOF )
            putchar( result = tolower( c ) );

    }
```

If you call the program test and type in

CONVERT

you see

convert

_tolower macro

Maps uppercase alphabetic to lowercase alphabetic.

Calling Sequence

```
#include <stdio.h>
```

or

```
#include <ctype.h>
```

```
int    c, result;  
...  
result = _tolower(c);
```

This macro assumes the character to be alphabetic; however, this generates smaller code than the tolower macro.

Related Functions

See also tolower, toupper, __toupper.

Example

```
/* Program test for the _tolower() macro */  
  
#include <stdio.h>  
  
int    c, result;  
  
main( argc, argv )  
  
    int    argc;  
    char   *argv[];  
  
    {  
  
        while ( ( c = getchar() ) != EOF )  
            putchar( result = _tolower( c ) );  
  
    }
```

If you call the program test and type in

CONVERT

you see

convert


```
*****
toupper                                                    macro
```

Maps lowercase alphabetic to uppercase alphabetic.

```
*****
```

Calling Sequence

```
#include <stdio.h>
```

or

```
#include <ctype.h>
```

```
int    c, result;
...
result = toupper( c );
```

This macro is not defined for arguments for which the `isascii` macro is false. Since the argument is evaluated more than once in the macro, it should not contain side effects.

Related Functions

See also `tolower`, `_tolower`, `_toupper`.

Example

```
/* Program test for the toupper() macro */

#include <stdio.h>

int    c, result;

main( argc, argv )
{
    int    argc;
    char   *argv[];

    while ( ( c = getchar() ) != EOF )
        putchar( result = toupper( c ) );
}
```

If you call the program test and type in

transform

you see

TRANSFORM

```

*****
_toupper                                          macro

Maps lowercase alphabetic to uppercase alphabetic.
*****

```

Calling Sequence

```

#include <stdio.h>

    or

#include <ctype.h>

    int    c, result;
    ...
    result = _toupper( c );

```

This macro assumes the character to be alphabetic; however, this macro generates smaller code than the toupper macro.

Related Functions

See also toupper, tolower, _tolower.

Example

```

/* Program test for the _toupper() macro */

#include <stdio.h>

    int    c, result;

    main( argc, argv )

        int    argc;
        char   *argv[];

    {

        while ( ( c = getchar() ) != EOF )
            putchar( result = _toupper( c ) );

    }

```

If you call the program test and type in

transform

you see

TRANSFORM

```
*****
traceback                                     function
*****
```

Prints a stack traceback and continues.

Calling Sequence

```
error = traceback( err );
```

where

err is either an AOS/VS or AOS/RT32 error code, or 0 to
use the last error code

and execution continues.

Returns

The return value is the error code signal.

Example

```
#include <stdio.h>

main()
{
    printf("Calling secl.\n");
    secl();
}

secl()
{
    printf("Calling sec2.\n");
    sec2();
}
```

```

sec2()
{

    printf("Calling sec3.\n");
    sec3();

}

sec3()
{

    int    ch;

    while( (ch = getchar()) != EOF )
        putchar( ch );

    traceback( 0 );
        /* print traceback & End of File error */

}

```

A call to the program test and entry of

abc

generates

```

Calling sec1.
Calling sec2.
Calling sec3.
abc

```

```

ERROR          24.
from $traceback.

```

Call Traceback:

```

from fp=16000013030, pc=sec3+161,   line 663 of sec3
from fp=16000012776, pc=sec2+31,   line 651 of sec2
from fp=16000012744, pc=sec1+31,   line 643 of sec1
from fp=16000012712, pc=main+31,   line 635 of main
from fp=16000012560, pc=.MAIN.+73,  C initializer
from fp=          0, pc=I.INIT+46

```

END OF FILE

```
*****
ttyname                                     function

Returns a byte pointer to the name of a file.
*****
```

Calling Sequence

```
-----
char    *name, *ttyname();
int      fildes;
...
name = ttyname( fildes );
```

The pointer returned is in the static area, and will be overwritten on the next call to the ttyname function.

The name returned is in UNIX format, and is not acceptable as input to an open routine.

Returns

If the file is not opened at a terminal, the function returns a NULL pointer.

```
*****
ungetc                                                    function
*****
```

Returns a character to the input buffer.

```
*****
```

Calling Sequence

```
-----
#include <stdio.h>
```

```
FILE    *fd;
char    next;
...
ungetc(next[, fd]);
```

where

next is a character placed in the look-ahead byte of
the buffer to which fd points

The function uses the standard input file if you omit fd. You
lose any character already in the look-ahead buffer.

Example

```
-----
/* Program test for the ungetc() function */

#include <stdio.h>

char    digit;

main( argc, argv )
{
    int    argc;
    char    *argv[];

    {
        while ( ( digit = getchar() ) != EOF )
        {
            if ( ( '0' <= digit ) && ( digit <= '9' ) )
                putchar( digit );
            else
            {
                ungetc( digit );
                exit();
            }
        }
    }
}
-----
```


If you call the program test and enter

1776AD

you see

1776

since the program places the first nondigit back in the input
buffer and exits.

```

*****
unlink                                     function

Deletes a file.
*****

```

Calling Sequence

```

#include "usc.h"

char    *name;
...
unlink( name );

```

Example

```

/* Program test for the unlink() function */

main( argc, argv )
    int    argc;
    char    *argv[];
{
    unlink( argv[1] );
    printf("Deleted %s.\n", argv[1] );
}

```

A call to the program test with

```
x test DELFILE
```

when DELFILE is a deletable file generates the output

```
Deleted DELFILE.
```

```

*****
utoa                                     function

Converts an unsigned integer to a character string.
*****

```

Calling Sequence

```

    unsigned    num;
    char        *string;
    ...
    utoa(num, string);

```

where

num is the unsigned integer to convert

string is the byte pointer to the string to hold the number

The function puts a maximum of 11 digits and a trailing null into the string.

The include file `stdio.h` defines this function.

The include file `stdio.h` also defines the macro `L_utoa`, which is 12, the maximum number of characters that `utoa` returns.

Example

```

/* Program test for the utoa() function */

char    place[12];
unsigned    store, dim;

main( argc, argv )
{
    int    argc;
    char    *argv[];

    store = atou( argv[1] );
    dim = atou( argv[2] );
    while ( dim > 0 )
    {
        utoa( store--, place );
        puts( place );
        dim--;
    }
}

```

A call to the program test with

x test 1984 5

generates the output

1984
1983
1982
1981
1980

```

*****
verify                                                    function

Determines whether one string contains a character not in the
other.
*****

```

Calling Sequence

```

char    *string1, *string2;
int      verify(), offset;
...
offset = verify( string1, string2 );

```

where

string1 and string2 are pointers to character strings ending
in the null character ('\0')

The value of offset is -1 if all of the characters in string1 are in
string2. Otherwise the value is the offset of the first character
in string1 that is not in string2.

Example

```

/* Program test for the verify() function */

int      offset, verify();

main( argc, argv )

    int      argc;
    char     *argv[];

{

    offset = verify( argv[1], argv[2] );
    if ( offset == -1 )
        printf( "'%s' contains all of '%s'\n",
                argv[1], argv[2] );
    else
        printf( "Offset in '%s' is %d.\n",
                argv[1], offset );

}

```

A call to the program test with

x test humanize humanoid

generates the output

Offset in 'humanize' is 6.

```

*****
wdleng                                     function

Returns the integer value 32, the word length in bits of the
type int.
*****

```

Calling Sequence

```

    int    m, wdleng();
    ...
    m=wdleng();

```

where

m is of type int

Example

```

/* Program test for the wdleng() function */

int    m;

main()
{
    printf("The wdleng function returns %d.\n",
           m = wdleng() );
}

```

A call to the program test generates the output

The wdleng function returns 32.

```
*****
zero                                                    function
```

Zeroes a section of memory.

```
*****
```

Calling Sequence

```
-----
```

```
char    *loss;
int      size;
...
zero(loss, size);
```

where

loss is a byte pointer to the memory location to zero

size is the number of bytes to zero

Example

```
-----
```

```
/* Program test for the zero() function */
```

```
char    loss[] = "Zero out this string.";
```

```
int      size;
```

```
main()
```

```
{
```

```
    printf("%s\n", loss );
    size = strlen( loss );
    zero( loss, size );
    printf("'%s' zeroed out.\n", loss );
```

```
}
```

A call to the program test generates the output

```
Zero out this string.
```

```
'' zeroed out
```



```

*****
zeropad                                     function
Substitutes zeros for leading spaces.
*****

```

Calling Sequence

```

char    *digs;
...
zeropad(digs);

```

where

digs is a byte pointer to a character array

Example

```

/* Program test for the zeropad() function */

char    digs[] = "      7";
main()
{
    printf("%s\n", digs );
    zeropad( digs );
    printf("%s\n", digs );
}

```

A call to the program test generates the output

```

      7
0000007

```

End of Chapter

CHAPTER 11

OPERATING PROCEDURES

This chapter contains instructions and options to use when compiling, linking, and executing C programs. It describes C debugging and error handling.

Compiling C Program Modules

=====

This section outlines options available to you at compilation time, and sketches the use of certain preprocessor statements.

AOS/VS Compilation Command Line

The AOS/VS C programming language compilation command line has the form

```
XEQ[/S] CC[/global-switch-string] source-file-data<NL>
```

where

global-switch-string

is

global-switch[/global-switch-string]

and

source-file-data

contains

exactly one

<main-filename>

from zero to seven

<filename>/include

any number of

<identifier>/define

any number of

<identifier>/define=<value>

| and up to eight

|

| <filename>/search

The optional /S switch on the CLI XEQ command returns the program's termination message to STRING. If the program compiles successfully, STRING is empty. If the program does not compile, STRING contains an error message.

You may enter the components of source-file-data in any mixture and in any order. You may shorten XEQ to X, include to i, search to s, and define to d.

Conditional Compilation

The lines in your program source file following

```
#if <constant-exp><NL>
```

will compile if the constant-exp has a nonzero value. The number sign (#) must occur in column 1.

Lines following

```
#ifdef <identifier><NL>
```

will compile if the identifier was previously stipulated with a #define statement or a /define compilation switch.

Lines following

```
#ifndef <identifier><NL>
```

will compile if the identifier was not previously stipulated with a #define statement or a /define compilation switch.

An

```
#else<NL>
```

may follow any of the preceding three statements, in which case the lines following it will compile if and only if the lines preceding it do not compile.

Whether or not an #else occurs, you must use an

```
#endif<NL>
```

to specify the end of the conditional compilation.

You may limit the scope of a #define by specifying

```
#undef <identifier><NL>
```

after which identifier is no longer valid.

Macros

If you wish a macro definition to be valid from the beginning of a program, you may specify it in the compilation command line as

```
XEQ[/S] CC ... <identifier>/define=<value> ...<NL>
```

where identifier corresponds to the name in #define, and value corresponds to the string. You may place the macro definition anywhere in the command line following the global switches, and you may specify as many definitions as you like. You may define an identifier as itself with

```
XEQ[/S] CC ... <identifier>/define ...<NL>
```

This is helpful if you wish to compile conditionally, for example for debugging. You cannot define a macro that has arguments from the compilation line.

You may also define substitutions in your source file with C macro definitions. The macro names have the same form as C identifiers. You may redefine the macro for different sections of code. The substitutions are valid from the point of definition either to the end of the source file or to the point of redefinition. Substitution does not take place within quoted strings.

The macro terminates with a NEW LINE, not a semicolon. If you wish to include a NEW LINE within the macro definition, precede it with a \.

Macros defined from the command line in the form

```
MACRO_NAME/define
```

as well as the predefined macros AOSVS and aosvs take the value 1 instead of the identifier itself. The AOSVS or aosvs macro allows you to specify compilation conditional on an AOS/VS environment.

Simple macros have the form

```
#define <NAME> <string><NL>
```

where name has the form of an identifier, and string is any syntactically valid string of C code. If your file contains the line

```
#define MAX 1000<NL>
```

then the string 1000 replaces MAX everywhere (except in quoted strings) from the point of definition to the end of the file or the point of redefinition. It is customary to cite macro names in uppercase.

Include Files

If you wish to include the contents of a file as valid from the beginning of, and perhaps throughout, a program, you may specify their inclusion with a local switch in the compilation command line.

```
XEQ[/S] CC ... filename/include ...<NL>
```

You may specify up to seven include files in the compilation command line. The include filenames may occur anywhere after the global switches.

You may also include the contents of other files in the file you wish to compile. Your source file may contain either the line

```
#include "filename"<NL>
```

or the line

```
#include <filename><NL>
```

where filename is the file you wish to include in the compilation. The line terminates in a NEW LINE, not with a semicolon. The contents of filename are valid only from the point in the program at which you specify the inclusion. If you wish the contents to be valid throughout the program, specify the inclusion at the beginning of the source file. Include files can contain other include files to a nesting limit of seven.

Compilation Switches

Table 11-1 shows the global switches available for use in the compilation line. Table 11-2 shows local switches for use on arguments to the compilation command.

Table 11-1. Global Compilation Switches

GLOBAL SWITCHES (on the CC command name):	
Paired Switches	
Default Switch	Nondefault Switch
/NOCHECKOFFSET	/CHECKOFFSET
/NOCODE	/CODE
/CONDITIONAL*	/NOCONDITIONAL
/NODEBUG	/DEBUG
/NODISALLOWANCIENT	/DISALLOWANCIENT
/NOEXTL	/EXTL
/NOEXTERNMAP	/EXTERNMAP
/NOFIXEDOVERFLOW	/FIXEDOVERFLOW
/NOFLAGDEFINES	/FLAGDEFINES
/NOFRAMESIZE	/FRAMESIZE
/INCLUDES*	/NOINCLUDES
/NOLINEID	/LINEID
/NOLOWERCASE	/LOWERCASE
/MAP*	/NOMAP
/NOOPTIMIZE	/OPTIMIZE=n
/NOPOINTERCHECK	/POINTERCHECK
/NOPROCID	/PROCID
/NOSHAREDSTRING	/SHAREDSTRING
/NOSHORTARITHMETIC	/SHORTARITHMETIC
/NOSIGNEDCHAR	/SIGNEDCHAR
/NOSTATISTICS	/STATISTICS
/NOSUBCHECK	/SUBCHECK
/NOUNDECLARED	/UNDECLARED
/NOWARNINGS	/WARNINGS
/NOXREF	/XREF
* In all but these cases, the /NO<switch> is the default value.	

(continues)

Table 11-1. Global Compilation Switches

Single Switches
<pre> /BACKSPACE[=value] /E[=filename] /ERRORCOUNT=n /INCDIRECTORY=<directory-name> /L[=filename] /LINK /LS /N /O=filename /PARTCODE=partition /PARTSTATIC=partition /PARTSHORT=partition /REVISION=maj_rev[.min_rev[.update[.pass[.]]]]] /SHORTSTATIC=n /TEMPDIR=directory /UNIQUE=n </pre>

(concluded)

Table 11-2. Local Compilation Switches

LOCAL SWITCHES (on arguments to CC):
/DEFINE
/INCLUDE
/SEARCH

Global Switches

You may abbreviate switches up to ambiguity.

Many switches come in pairs /<switch>, /NO<switch> whose uses are mutually exclusive. One of the pair is the default. If you select neither member of the pair, the compiler assumes the default value. A d in the first column of the following list indicates that this switch is the default value of a pair of switches.

/BACKSPACE[=value]	
Evaluate '\b' with the specified value. If you do not specify a value, the compiler evaluates '\b' as '\10'. If you do not use this switch, the compiler evaluates '\b' as '\31'.	
/CHECKOFFSET	
Check whether members of structures or unions with the same name are at the same offset, and of the same type, in all structures or unions.	
/CODE	
Write out a code listing. You must use the /L switch to enable this switch.	
d /CONDITIONAL	
List all code, even code not conditionally compiled.	
/DEBUG	
Produce SWAT (TM) information.	
/DISALLOWANCIENT	
Do not evaluate assignment operators of the form =<op>.	
If you use this switch, the expression a = 2 becomes a = (-2) rather than a = a - 2.	
/E[=filename]	
Redirect the error listing to filename. If you use /E alone, the default error file is @OUTPUT. If you omit /E, the default error listing is @OUTPUT.	
/ERRORCOUNT=n	
Set the maximum number of errors to list to n. If you omit a value, the default maximum is 100. If you exceed the maximum number of errors, compilation aborts.	

/EXTERNMAP

On external names, convert names with underscores (_) to corresponding names with question marks (?) in calling the macroassembler.

/EXTL

Generate an external reference for each declaration with an explicit extern storage class declaration. Each reference must be satisfied in another module (as opposed to generating a partition definition block, which occupies the same storage as other partition definition blocks with the same name when the program is linked). This lets you pull default

initializations from a library. You could also use the switch to reference entry points defined in assembly language. Any declaration preceded by the keyword `extern`, but not initialized, is assumed to be defined in some other module.

`/FIXEDOVERFLOW`

Turn on fixed overflow checking for all functions defined within the current compilation.

`/FLAGDEFINES`

Set a flag to tell whether the compiler used a `#define` macro on the current line. You must use the `/L` switch to enable this switch.

`/FRAMESIZE`

Print on the listing and error files the size in words that each function needs from the stack. This does not include space needed by the `alloc`, `calloc`, or `malloc` functions.

`/INCDIRECTORY=<directory-name>`

Specify the directory to prefix to an `#include <filename>`. If the file does not exist in the directory, the compiler then attempts to open the file from the current directory using the current searchlist.

`d /INCLUDES`

Print include files on the listing file. You must use the `/L` switch to enable this switch.

`/L[=filename]`

Produce a listing in `filename`. If you use `/L` alone, the default filename is `@LIST`, and if you omit `/L`, the default is no listing. You must use the `/L` switch to enable the `/CODE`, `/FLAGDEFINES`, `/INCLUDES`, `/MAP`, or `/XREF` switches.

`/LINEID`

Generate code to save the line number of the statement. Use the saved number to report the location of runtime errors. Automatically set the `/PROCID` switch to identify functions and modules.

`/LINK`

This switch is not interpreted by the compiler, but rather by the `CC.CLI CLI` macro. If you use this switch, it will call the `CCL.CLI` macro to link the various C modules together, providing there were no errors in the compilation. If used, the following CCL switches are available, and passed on the `LINK`:

/LS
/NOIOSWITCH
/RESERVE=#
/HEAPSIZE=#
/STACKSIZE=#
/MODMAP
/MODSYM
/ALPHA
/NUMERIC
/MTOP=#
/TASKS=#
/RT32
/SHAREDIO
/NOSEA

/LOWERCASE

Map identifiers to lowercase.

/LS

This switch is not a compiler switch, but is used by the CC.CLI macro, and creates a listing file of the form <file>.LS, where <file> is the name of the program being compiled, without the .C extension.

d /MAP

Produce a map of storage on the listing file. You must use the /L switch to enable this switch.

/N

Do not produce an .OB file. If you omit this switch, you generate code.

|d /NOCHECKOFFSET

| Do not check members of structures or unions with the same
| name for offset or type.

d /NOCODE

Do not list generated code.

/NOCONDITIONAL

Do not list code that is not conditionally compiled.

d /NODEBUG

Omit generation of SWAT (TM) information.

|d /NODISALLOWANCIENT

| Treat assignment operators of the form =<op> as equivalent
| to operators of the form <op>=.

d /NOEXTERNMAP

Do not map underscores (__) to question marks (?) in external names.

d /NOEXTL

All definitions that are external, including definitions with an explicit extern declaration, generate a partition definition block. LINK will resolve all of the partition definition blocks with the same name to the same address.

d /NOFIXEDOVERFLOW

Do not check for fixed overflow.

d /NOFLAGDEFINES
Do not set a flag for #define macros used on the line.

d /NOFRAME SIZE
Do not display the stack space needed.

/NOINCLUDES
Do not print include files.

d /NOLINEID
Do not produce code to save line numbers of statements.

d /NOLOWERCASE
Do not map identifiers to lowercase automatically.

/NOMAP
Do not produce a storage map.

d /NOOPTIMIZE
Do not optimize the code. If you use this switch, your optimization level is 0.

d /NOPOINTERCHECK
Do not check pointers for validity.

d /NOPROCID
Do not produce code to save the procedure name.

d /NOSHAREDSTRING
Places quoted strings in unshared memory, where they can be modified.

d /NOSHORTARITHMETIC
Perform implicit conversions before carrying out arithmetic operations: convert short and char to int, short unsigned to unsigned, and float to double.

d /NOSIGNEDCHAR
Treat char variables as unsigned numbers in the range 0...255.

d /NOSTATISTICS
Do not generate compiler statistics.

|d /NOSUBCHECK
| Do not check for out-of-bounds array subscripts.

d /NOUNDECLARED
Give an error message if a structure or union is not defined in the declaration.

d /NOWARNINGS
Do not generate any warnings.

d /NOXREF
Do not generate a cross reference.

/O=filename

Redirect the object file to filename.OB. If you omit this switch, your object file is main-filename.OB.

/OPTIMIZE=n

Set the optimization level to n (n = 0, 1, 2, 3). If you omit a value, the default level is 3. If you omit the switch, the default level is 0.

Level	Optimization
-----	-----

0	Do not optimize.

1	Eliminate unreachable code, such as an unlabeled statement following an unconditional goto.

2	Include level 1 optimizations. In addition, eliminate redundant computations.

3	Include level 2 optimizations. In addition, remove invariant computations from do loops.

/PARTCODE=partition

Specify which partition shared code goes in. If you omit this switch, the compiler uses the default shared code partition. For more details, see the "AOS/VS Link and Library File Editor User's Manual."

/PARTSTATIC=partition

Specify which partition the static data goes in. If you omit this switch, the compiler uses the default unshared data partition. For more details, see the "AOS/VS Link and Library File Editor User's Manual."

/PARTSHORT=partition

Specify which partition the short static data goes in. If you omit this switch, the compiler uses UD.SHORT_STATIC. For more details, see the "AOS/VS Link and Library File Editor User's Manual."

/POINTERCHECK

Check all pointers for validity, and generate a stack traceback if the pointer is invalid. If a byte pointer (bit 0 = 0) occurs where a word pointer is expected, or a word pointer (bit 0 = 1) occurs where a byte pointer is expected, convert the pointer and continue execution. (The /LINEID switch will tell you the line number of the statement that uses the invalid pointer. Pointers are also checked at conversion time (byte vs. word) for validity, except that the NULL pointer is not given an error in this case.) There are some address traps that the /POINTERCHECK switch does not catch.

For most usages, the switch should help to identify places where byte and word pointers are used indiscriminately. To use this switch, your program must run in rings 4-7.

/PROCID

Produce code to save the procedure name. Use the saved name to report the offending function or module in runtime errors.

/REVISION=maj_rev[.min_rev[.update[.pass[.]]]]]

Specify the revision number for the module. Each component of the revision number must be less than 256. If you omit this switch, the default is the compiler revision number.

/SHAREDSTRING

Places quoted strings in shared (write-protected) memory. This may give a slight performance gain when running multiple versions of the same program file, and slightly smaller code.

/SHORTARITHMETIC

Use narrow (16-bit integer) arithmetic for types short and short unsigned. Use single-precision (32-bit floating) arithmetic for the type float. This switch produces shorter or faster code for the given types, but accuracy suffers.

/SHORTSTATIC=n

If the number of words in static or external data items is less than or equal to n, use the short addressing mode instead of the long addressing mode (that is, use an X-type instruction rather than an L-type instruction, and locate the data within the first 32K of memory).

/SIGNEDCHAR

Treat char variables as signed numbers in the range -128...127 rather than as unsigned numbers in the range 0...255. The compiler generates a warning for values outside the range -128...127. (Unsigned declarations such as unsigned char a; use values in the range 0...255.)

/STATISTICS

Generate compiler statistics, such as number of lines compiled.

| /SUBCHECK

| Check for out-of-bounds array subscripts. (You will not be
| able to use this switch if your program uses pointers instead
| of arrays.)

/TEMPDIR=directory

Specify an alternate directory for temporary files. If you omit this switch, the default is the current working directory.

/UNDECLARED

Do not give an error message if a structure or union is undefined. Since an undefined structure occupies no storage, you will not be able to use the structure in any expression, or other structure definitions. Other compilers might not

check for undefined structures until the structure is referenced.
This switch disables the check so that existing C programs
can run without errors if they use this feature.

/UNIQUE=n

Set the number of unique characters in identifiers (from 8 to 32).
If you omit this switch, the default number of unique characters
is 32.

/WARNINGS

List warnings.

/XREF

Generate a cross reference: append an alphabetical
cross-reference of all program entities, with line numbers,
to the listing file. You must use the /L switch to enable this
switch. The cross listing includes #define macros. If
any level 3 compilation errors occur, the compiler does not
generate any cross references, since they run off intermediate
code.

Local Switches

/DEFINE

Set up a #define from the command line.

/INCLUDE

Set up an include file from the command line.

/SEARCH

Search this directory for include files before a normal
search following AOS/VS rules (current working directory and
searchlist). If you specify the include file with
quotation marks (" ") instead of angle brackets (< >), the
directory containing the main file is searched first. You
may specify up to 8 directories for searching.

You may use the CC.CLI macro provided to you to compile a C program module. The CC.CLI macro provided as part of the C language product will compile each of the modules given that end in .C (or no extension), and pass any arguments with switches (ie, foo/DEFINE) to all modules. Any modules that end in .SR are assumed to be assembly language modules and are assembled. Modules that end in .OB or .LB are assumed to be object modules and are not compiled.

If the user also gives the /LINK switch, and there were no errors in the compilation or assembly, the CCL.CLI macro will be invoked to link all of the modules together (including .OB and .LB modules).

The /LS switch will produce listings for each compilation, assembly (with a .LS extension instead of .C or .SR), and a LINK map (with a .LM extension). The following switches are assumed to be LINK switches and are passed on to CCL:

Switch	Effect	
-----	-----	
/NOIOSWITCH	Do not recognize runtime switches.	
/RESERVE=#	Reserve # pages for ?MEMI or ?SSHPT.	
/HEAP=#	Use a heap of # words.	
/STACK=#	Use a stack of # words.	
/MODMAP	Put out module map.	
/MODSYM	Put out module symbols.	
/ALPHA	Sort symbols alphabetically.	
/NUMERIC	Sort symbols numerically.	
/MTOP=#	Specify program to take # megabytes.	
/TASKS=#	Specify # tasks.	
/DEBUG	Build .DS and .DL files (also used for compilation).	
/NOSEA	Turn off the AOS/VS searchlist, unless you execute the program with a local /SEA switch.	
/RT32	Link for AOS/RT32 instead of AOS/VS.	
/SHAREDIO	Force all disk files to use AGENT shared I/O. If all users of the file open it this way, the file is updated immediately, rather than being buffered in the AGENT.	

Three files are available for the system manager to edit to give default switches for a CC.CLI invocation.

- * The file CC.CSW.CLI specifies any default switches to be given to the CC.CLI command (such as /SIGNEDCHAR, /LS, /DEBUG, /OPTIMIZE, etc).
- * The file CC.MSW.CLI specifies any default switches to be given to MASM.
- * The file CC.LSW.CLI specifies any default switches to be given to CCL.CLI.

By default CC.CSW.CLI and CC.LSW.CLI specifies no switches, and CC.MSW.CLI specifies /ULC/SYMBOL=32. If you modify either file, take the precaution of placing the file in a directory which is in the user's searchlist before the C release directory, or turn permanence on, so that future releases will not override the defaults you select.

You may wish to construct your own compilation macro. A useful, relatively complex macro might have the form

```
[!EQUAL,%/LS%,/LS]
    DELETE/2=IGNORE/V %1%.LS
    X[!EQUAL,%/S%,/S]/S[!END] CC/L==%1%.LS%\S\LS% %1-%
[!ELSE]
    X[!EQUAL,%/S%,/S]/S[!END] CC%\S% %1-%
[!END]
```

A minimal but useful CLI macro for C compilation might be a file containing just

```
DELETE/2=IGNORE/V %1%.LS
X CC/L==%1%.LS%/ %1-%
```

which deletes any previous listing file, ensures that the

current listing file has the name of the current source file with a .LS extension, and permits you to enter other global switches as the need arises.

Linking C Programs

=====

This section provides you with a number of linking options which alter the way the runtime initializer configures memory. You can map additional UNSHARED pages into the program context with ?MEMI system calls that set up the stack and heap, changing the number of unshared pages in your program's logical address space. This information should be of interest if your program is linked for a very large address space and is designed to make ?MEMI system calls, or if your program is multitasked.

Terminology

The "heap" refers to the area in memory from which the C calloc, malloc, strsave, and realloc functions parcel out storage.

A "fixed stack" is a stack with a stack limit that does not change. The stack may grow until reaching this limit, at which time the program aborts.

A "dynamic stack" is a stack which uses the top of the UNSHARED address space as the stack limit. The stack fault handler will ?MEMI additional pages and reset the stack limit as the stack requirements for the program grow.

A "fixed heap" is a heap in which the total amount of storage available for allocation is fixed. Storage is allocated from the heap until exhausted, at which time the program aborts.

A "dynamic heap" is a heap that uses the top of the UNSHARED address space (highest addresses) for storage. The heap management runtimes issue ?MEMI calls for additional UNSHARED pages as the heap requirements for the program grow.

A "dynamic stack/heap" is a memory organization in which the stack and the heap share the same area of memory. The heap occupies the higher addresses and grows down as required. The stack occupies the lower addresses. The stack limit is effectively the current bottom of the heap area and serves as the boundary between the stack and the heap. The stack limit is adjusted down and up as the heap grows and shrinks, respectively. The program aborts if the stack ever reaches the stack limit or if the heap bottom (stack limit) would become less than the current stack pointer.

Multitasking programs that require a heap can use a dynamic stack/heap, provided only that the main task allocates and frees: the heap is a global resource (common to all tasks) but the stack limit is a local value (each task has its own).

C programs always require a heap for reading in initial arguments, and for all file opens, including the three default opens.

Use of the Switches

The stack and heap are set up by the runtime initializer according to whether or not the following symbols have been defined at linktime to have values other than -1 (which means that the value is not defined):

.RESERVE
.STKSIZE
.HPSIZE
.STORAGE

You set the symbols above by assigning values in the LINK command line or by linking with .OBs, produced by MASM, that define their values. All of these symbols will have a value of -1 by default if not otherwise defined.

.RESERVE

The value of .RESERVE (if other than -1) denotes the number of pages (1024. words) the initializer is to leave unmapped between the unshared and the shared partitions. These pages are available for ?MEMI calls by the program. ?SBOT-1024.*.RESERVE is the address of this unmapped area, and all addresses below this will be mapped as UNSHARED.

.STKSIZE

The value of .STKSIZE (if other than -1) denotes the size, in words, of a fixed stack.

.HPSIZE

The value of .HPSIZE (if other than -1) denotes the size, in words, of a fixed heap.

.STORAGE

The value of .STORAGE (if other than -1) denotes the size, in words, of an area of storage to be used as a dynamic stack/heap.

Link Example

```
XEQ LINK/<switches> main .STKSIZE/val=10000 ...
```

or in an .OB

```
.ENT      .STKSIZE
.STKSIZE = 10000.
```

Combinable Options

You can use only certain combinations of defined values of the symbols. For example, defining values other than -1 for .STKSIZE, .HPSIZE, and .STORAGE all at the same time is not an allowed combination, and results in an unpredictable initialization. The following list describes the allowed combinations and the resulting actions by the initializer. See Figure 11-1 at the end of this section. The description for <none> is the default behavior (none of the values are defined).

<none>

Make ?MEMI calls for all available pages as UNSHARED and use as a dynamic stack/heap. (Col. A)

.RESERVE

Make ?MEMI calls for all but .RESERVE pages as UNSHARED and use as a dynamic stack/heap. The program then has .RESERVE pages available for ?MEMI calls. (Col. A)

.STKSIZE

Make ?MEMI calls for just enough pages for a fixed stack of .STKSIZE words and an initial heap. Heap runtimes make ?MEMI calls for additional UNSHARED pages as needed for the heap. The program must not ?MEMI additional UNSHARED pages. (Col. D)

.STORAGE

Make ?MEMI calls for enough pages for a dynamic stack/heap of .STORAGE words. Remaining pages are available for ?MEMI calls by the program. (Col. A)

.HPSIZE

Make ?MEMI calls for enough pages for a fixed heap of .HPSIZE words and an initial stack. Stack fault handler makes ?MEMI calls for additional UNSHARED pages as required. The program must not make ?MEMI calls for additional UNSHARED pages. (Col. C)

.RESERVE, .STKSIZE

Make ?MEMI calls for all but .RESERVE pages. Use .STKSIZE words for a fixed stack and the rest as a fixed heap. .RESERVE pages are available for ?MEMI calls by the program. (Col. B)

.RESERVE, .HPSIZE

Make ?MEMI calls for all but .RESERVE pages. Use .HPSIZE words as a fixed heap and the rest as a fixed stack. .RESERVE pages are available for ?MEMI calls by the program. (Col. B)

.STKSIZE, .HPSIZE

Make ?MEMI calls for just enough pages for a fixed stack of .STKSIZE words and a fixed heap of .HPSIZE words. Remaining pages are available for ?MEMI calls by the program. (Col. B)

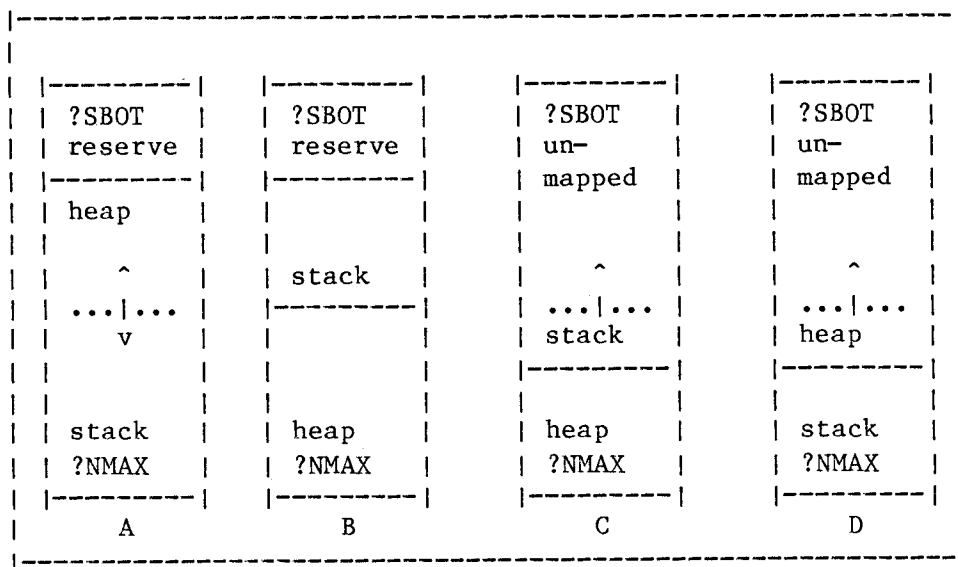


Figure 11-1. Memory Configurations

The diagrams represent the four basic memory configurations. The dotted lines (...) indicate boundaries that change in the directions indicated during the execution of the program.

- A Dynamic stack/heap, heap grows and shrinks by moving stack limit. Multitasking programs cannot use this configuration.
- B Fixed size stack and a fixed size heap.
- C Fixed size heap and a dynamic stack. Stack fault handler ?MEMIs additional pages as required.
- D Fixed size stack and a dynamic heap. Heap manager ?MEMIs additional pages as required.

CCL LINK Macro

The following CCL macro translates switches /STACK=, /HEAP=, /RESERVE=, and /STORAGE= into the correct <name>/val= arguments for LINK. You must not abbreviate these switches when you try to use them. If you want to direct the program not to recognize the /E=<filename>, /I=<filename>, and /L=<filename> switches, use the /NOIOSWITCH and the program's initializer will not check these switches. The /SHAREDIO switch directs the C runtime library to open all files as shared. This means that updates via writes are immediately reflected in other process reads. The CCL.CLI macro contains

```

[!EQUAL, %/LS%, /LS]
    DELETE/2=IGNORE/V %1\%.LM
[!END]
X LINK[!EQUAL, %/LS%, /LS]/L==%1\%.LM/V[!END]&
[!NEQUAL, %/RT32%, ]/NSLS[!END]&
%\LS\RESERVE\STORAGE\STACK\HEAP\NOIOSWITCH\RT32\SHAREDIO%&
    <UC,UD,SC,SD>/ALIGN=1&
    %1-%&
[!NEQUAL, %/DEBUG%, ]      SWATI                      [!END]&
[!NEQUAL, %/NOIOSWITCH%, ] $REGIO/VAL=0                [!END]&
[!NEQUAL, %/SHAREDIO%, ]   $SHAREDIO/VAL=1             [!END]&
[!NEQUAL, %/RESERVE=%, ]   .RESERVE/VAL=%/RESERVE=%    [!END]&
[!NEQUAL, %/STORAGE=%, ]   .STORAGE/VAL=%/STORAGE=%    [!END]&
[!NEQUAL, %/STACK=%, ]     .STKSIZE/VAL=%/STACK=%      [!END]&
[!NEQUAL, %/HEAP=%, ]      .HPSIZE/VAL=%/HEAP=%        [!END]&
&
    CSYSCALLS.LB DGLIB.LB STDIO.LB USC.LB GEN.LB LANG_RT.LB&
&
[!NEQUAL, %/RT32%, ]      AOSRT_URT32.LB              [!END]

```

For example, the link line

```
CCL/stack=5000 MY_PROG
```

resolves to

```
XEQ LINK MY_PROG .STKSIZE/VAL=5000 DGLIB.LB STDIO.LB USC.LB GEN.LB &
LANG_RT.LB
```

A minimal but useful CLI macro for linking C programs might be a file containing just

```

DELETE/2=IGNORE/V %1%.LM
X LINK/L==%1%.LM/% %1-% CENV.LB LANG_RT.LB

```

which deletes any previous listing file, ensures that the current listing file has the name of the current object file with a .LM extension, and permits you to enter other global switches as the need arises.

Executing C Programs

=====

To execute a successfully compiled and linked program file, type

```
XEQ[/S] <file0>[/I[=<file1>]][/L[=<file2>]][/E[=<file3>]]<NL>
```

where <file0>.PR is the name of the program file. You may use lowercase /i, /l, and /e as variants to the uppercase switch names. If you did not use the /NOIOSWITCH on the CCL macro, you can use the switches in Table 11-3 to specify input, output, and error files.

Table 11-3. Switches for Program Execution

standard files (stdio.h)			
switch	stdin /I	stdout /L	stderr /E
No switch	@INPUT	@OUTPUT	@OUTPUT
/SWITCH	@DATA	@LIST	@OUTPUT
/SWITCH=<filename>	<filename>	<filename>	<filename>

The optional /S switch on the CLI XEQ command returns the program's termination message to STRING. If the program runs successfully and has a normal return, STRING is empty. Otherwise, STRING contains an error message.

File Redirection

These switches on the program module allow you to specify file redirection at execution time:

Program Switch	File
/L=file	append to standard output
/O=file	recreate for standard output
/OO=file	append stdout to file without overwriting (same as /L=file)
/I=file	standard input
/E=file	standard error
/EE=file	append stderr to file without overwriting

Furnishing Arguments to Programs

To execute a program f0 with arguments, type

```
XEQ <f0>[/I[=<f1>]][/L[=<f2>]][/E[=<f3>]] <arg1> <arg2> ... <argn><NL>
```

where argi is the ith argument you wish to furnish to the program.

In order to invoke a program with arguments, you must call the main program with

```
main(argc, argv)      /* program with arguments */
int argc;
char *arg[];
{
.
.
.
}
```

or equivalently with

```
main(argc, argv)
int argc;
char **arg;
{
.
.
.
}
```

where the first argument to main, argc, is the number of arguments you invoke the program with. By convention, argv[0] is the name with which you invoke the program, so argc is at least 1. A single argument to a program is argv[1], and argc = 2. If there are n arguments to a program, argv[n] is the last argument, and argc == n+1.

The second argument to main, argv, is a pointer to an array of character strings. Each string contains a single one of the arguments to the program. To manipulate these strings as something other than character arrays, you must make explicit use of such runtime functions as atof or atoi.

Debugging Aids for C Programs

As you use the AOS/VS compiler, you have two choices of help in debugging a C program, the SWAT debugger, or the AOS/VS language debugger. You can use them singly or together.

SWAT Native Language Debugger

You can use SWAT, Data General's Native Language Debugger, which allows you to debug the program at the level of source code. Using SWAT, you can

- * set breakpoints at any executable statement, and have SWAT carry out a list of commands before executing the statement
- * display variables according to their type
- * set variables to general expressions
- * get the type of a variable, in terms of the language it is defined in
- * read SWAT commands from a file
- * list portions of the source, including include files
- * call the AOS/VS assembly language debugger
- * provide a runtime walkback at a breakpoint, to see which functions called the current function
- * use one terminal for program execution and a second terminal for debugging

To use SWAT, you must

- * compile each program module you wish to debug with a /DEBUG switch in the compilation line
- * link the program with a /DEBUG switch on the link line
- * place the SWAT interface file SWATI on the link line before any language library files

You call SWAT with

```
XEQ SWAT[/global switch ...] program[/switch...] [arg[/switch...]]...
```

where program is the name of the program you wish to debug, with any switches, and arg is any argument to your program, with any switches.

For more information about using SWAT, consult the "SWAT User's Manual".

AOS/VS Assembly Language Debugger

You can use the AOS/VS assembly language debugger to

- * gain access to the actual machine instructions, displayed in normal assembler mnemonics
- * modify the instructions if you wish
- * gain access to all the machine registers available to user programs

You do not have to give any special debugging command to the compiler or linker.

You may enter the debugger from the CLI with a

```
DEBUG <program-file> [<program-arg> ...]
```

command, or you may enter it from the program with a ?DEBUG system call from your assembly language program. For more information, consult the "Advanced Operating System/Virtual Storage (AOS/VS) Debugger and File Editor".

Error Handling for C Programs

The AOS/VS C compiler gives error diagnostics in plain, clear English. It prints the line in question, along with a little arrow indicating the token in error. If the error is in a macro expansion, it also prints out the macro name. For example, this program contains two errors

```
#define foo(a) a += 1

main( argc, argv )
    char    *argv[];
{
    auto    float    b;
    auto    int      b;        /* different declaration */

    foo(1);                    /* expands to 1 += 1 */
}
```

and generates the error messages

```
ERROR 179 SEVERITY 3 BEGINNING ON LINE 9 (File er.c)
    auto    int      b;        /* different declaration */
                ^
```

You have already declared "b" as a variable, enumeration, or typedef in the current block.

ERROR 376 SEVERITY 3 BEGINNING ON LINE 11 (Macro foo, File er.c)
a += 1
^

The lefthand sub-expression for the binary operator "+=" must be a variable or array element. It cannot be an expression, constant, array name, or typedef name.

When the C compiler encounters an error, the course of action it takes depends upon the severity, of the error. The levels, in increasing order of severity are

Level	Action
-----	-----
1	If you used the /WARNINGS compilation switch, the compiler reports a minor error or nonstandard use, but continues compilation. If you did not use the /WARNINGS switch, the compiler does not report the error.
2	The compiler found a serious error. It attempted to correct the error, and continued compilation.
3	The compiler found a serious error that it could not correct. It continues to process the file but does not generate code.
4	The compiler found a serious error that it could not correct. It aborts immediately.

Appendix A contains a complete listing of compiler error messages.

End of Chapter

APPENDIX A

COMPILATION ERROR MESSAGES

This appendix lists all compilation error messages and specifies their severity levels.

Severity Levels

=====

When the C compiler encounters an error, the action it carries out depends upon the severity level associated with the error. You can adjust error reporting with the /WARNINGS switch.

Level	Action

1	If you used the /WARNINGS compilation switch, the compiler reports a minor error or nonstandard use, but continues compilation. If you did not use the /WARNINGS switch, the compiler does not report the error.
2	The compiler found a serious error. It attempted to correct the error, and continued compilation.
3	The compiler found a serious error that it could not correct. It continues to process the file but does not generate code.
4	The compiler found a serious error that it could not correct. It aborts immediately.

Compiler Error Messages

=====

C error handling supplies actual values for arguments, switches, and the like in place of the template "\$" in this listing of the error messages. The first column contains the error number. The second contains the severity level of the error. Since error messages are grouped and numbered by type, some numbers are not used. The format of the message on the screen may differ somewhat from the format in this list, but the wording is identical.

Command Line Errors

1	3	The command line lacks an argument \$.
2	3	The command line contains a null switch.

- 3 3 The command line contains an unknown switch \$.
- 4 3 The command line contains a switch \$
that is not unique.
- 5 2 The command line contains multiple occurrences of a switch (\$).
The compiler used the first occurrence of the switch.
- 6 3 The command line lacks a local switch argument \$.
- 7 3 An argument on the command line has a null switch.
- 8 3 A command line argument has an unknown switch \$.
- 9 3 A command line argument has nonunique switch \$.
- 10 2 An argument on the command line has multiple occurrences of a
switch. The compiler used the first occurrence of the switch.
- 11 3 You must not use switch /\$ with a value
on the command line.
- 12 3 The switch /\$ must have a value on the
command line.
- 13 3 Switches \$ must not appear together on
the command line.
- 14 3 You must not use switch /\$ on an argument
with a value in the command line.
- 15 3 Switches \$ on a single argument must
not appear on the same argument in the command line.
- 16 3 The argument "\$" to either the /TEMPDIR= or /INCDIRECTORY=
switch on the command line must specify a directory.
- 17 4 You did not specify a source file on the command line.
- 18 3 The compiler found the source file \$ after
finding another source file.
- 19 3 The switch /\$ on the command line must have
an integer value from 1 to 32767.
- 20 3 If the switch /\$ has a value, it must be an
integer from 0 to 3.
- 21 3 The switch \$ on the command line must have an
integer value from 8 to 32.
- 22 2 The compiler truncated the partition name "\$"
to 32 characters.

- 23 3 \$ is an illegal revision number.
- 24 2 You cannot #define "\$" from the command line
since it is not an identifier. The compiler ignored it.
- 25 2 The body for the macro "\$" defined in the command
line exceeded the implementation limit of 136 characters. The
compiler ignored it.
- 26 3 The switch \$ on the command line must have
an integer value from 0 to 32767.
- 39 4 The compilation is aborting -- The command line contained serious
errors.

Preprocessor Errors

- 45 2 The compiler ignored the command #\$: unknown
to the preprocessor.
- 46 4 The nesting of the include file "\$" exceeds
the implementation limit of 7 files. The compiler is aborting.
- 49 2 The preprocessor statement ended before the compiler found the
NEW LINE. It skipped the rest of the line.
- 50 1 Warning: The preprocessor command #\$ is not portable.
- 51 2 The filename for a #include statement must begin either with
a left angle bracket (<) or a double quotation mark (").
- 52 2 The filename for a #include statement must end with a right angle
bracket (>) if it begins with a left angle bracket (<),
or a double quotation mark (") if it begins with a double
quotation mark.
- 53 2 The file name for a #include statement must be nonempty. The
compiler will ignore this #include.
- 54 2 The preprocessor statements #define, #ifdef, #ifndef, or #undef
require that an identifier follow the preprocessor command.
The compiler has ignored this preprocessor command.
- 55 2 You cannot use the reserved word "\$" as a macro
within a #define, #ifdef, #ifndef, or #undef
preprocessor command. The compiler has ignored this
preprocessor command.
- 56 2 You did not follow the identifier "\$" with
either a left parenthesis "(", or a whitespace character
(tab, space) in a #define preprocessor command. The
compiler has ignored the #define definition.

57 1 Warning: You did not #define the identifier "\$",
so the compiler ignored the #undef preprocessor command.

58 2 The compiler found an #else when there was no outstanding
#ifdef, #ifndef, or #if. The compiler ignored the #else.

59 2 The compiler found an extra #else after already matching an
#else to a #ifdef, #ifndef, or #if. The compiler ignored
the second #else.

60 2 The compiler found an #end when there was no outstanding
#ifdef, #ifndef, or #if. The compiler ignored the #end.

63 1 Warning: The compiler found a NEW LINE in the middle of a
comment inside a #define definition. The compiler skipped the
character inside the comment and continued processing the
#define macro.

64 3 You tried defining a macro "\$" with an argument
which was not an identifier. The compiler did not define
the macro.

65 2 You tried defining a macro "\$" but omitted
a comma or right parenthesis after an argument. The
compiler assumed correct punctuation and continued
processing.

66 2 You tried defining a macro "\$" with a number of arguments
exceeding the implementation limits of 255. The
compiler ignored the excess arguments.

67 2 You have already defined the argument "\$" as
a macro argument. The compiler ignored the second definition and
assumed the first.

68 1 Warning: you should not use the reserved word "\$" as a macro
argument name.

69 1 You defined the macro "\$" previously. The old
definition will not be used until an #undef is done.

70 3 The assertion was false.

71 2 In expanding the macro "\$", you did not give enough
arguments. The compiler set the remaining arguments to null.

72 2 In expanding the macro "\$", you gave more arguments than
you defined the macro with. The compiler ignored the excess
arguments.

73 3 The compiler found an End of File (or a NEW LINE within an #if or
#assert) while scanning a macro, or an argument to a macro.

- 74 2 You gave the macro "\$" no arguments. The compiler set the arguments to null.
- 75 2 A line number was not used in a #line preprocessor statement. The #line statement was ignored.
- 76 2 A line number greater than 32767 or less than or equal to 0 was used in a #line preprocessor statement. The #line statement was ignored.
- 77 2 A line number which was a floating-point number was used in a #line preprocessor statement. The #line statement was ignored.

Token Errors

-
- 100 2 The compiler found an illegal character "\$" (octal) in the source and skipped the character.
- 101 3 The compiler found an End of File while scanning a comment. Check to see whether you inadvertently omitted a */ comment delimiter.
- 102 1 Warning: The compiler skipped a preprocessor statement because the line was inside a comment.
- 103 1 Warning: The C language does not support nested comments.
- 104 2 The int constant \$ could not fit into 32 bits without a loss of precision. The compiler truncated it on the left.
- 105 2 While scanning \$, the compiler detected a floating-point underflow. The value used may be meaningless.
- 106 2 While scanning \$, the compiler detected a floating-point overflow. The value used may be meaningless.
- 107 2 The compiler found a character constant '\\$' larger than 0377 (octal), and truncated it to 8 bits of precision.
- 108 2 You exceeded the maximum string length of 256 characters. The compiler truncated the string on the right.
- 109 1 Warning: The compiler found a # in column 1 while scanning a quoted string. the compiler did not treat it as a preprocessor statement, but instead put it inside the string.
- 110 3 The compiler found an End of File while scanning a character string. Check to see that you properly terminated all strings with "".
- 111 3 The compiler found an End of File while scanning a character constant. Check to see that you properly terminated all character constants with '.

- 112 1 Warning: The compiler found a zero-length character constant, and assumed '\000'.
- 113 2 A character constant has more than 4 characters. The compiler ignored all characters but the last 4.
- 114 1 Warning: Multicharacter constants are nontransportable.
- 115 1 Warning: The compiler found a # in column 1 while scanning a character constant. The compiler did not treat it as a preprocessor statement, but instead put it inside the constant.
- 116 2 The floating-point constant \$ has at least 2 periods in it. The compiler used the first period to scale the number.
- 117 2 The hex constant \$ contained a period. The compiler ignored the period.
- 118 2 The floating-point constant \$ lacked a correct exponent, so 0 was used to scale the number.
- 119 2 The indicated token has been replaced with "\$".
- 120 2 The compiler found an end of line terminator in a string, without a preceding '\\' character. A double quotation mark (") has been inserted at this point.
- 121 2 The compiler found an end of line terminator in a character constant, without a preceding '\\' character. A single quotation mark (') has been inserted at this point.

Code Generation Errors

- 140 3 Previous severity 3 errors prevent generating code or producing the cross reference listing.
- 141 1 There was nothing to compile. You may have not closed a comment properly, failed to meet a conditional compilation condition, or specified an empty file.
- 142 3 A possible compiler error (\$) has occurred.
Please correct all other errors, and resubmit the compilation.
If this message persists, please submit a Software Trouble Report to Data General Corporation.
- 143 4 A compiler stack fault occurred during the compilation.
You may have nested expressions too deeply.
- 144 4 The system error \$
caused the compiler to abort the compilation.

Parser Errors

- 170 2 You used more than one type (int, float, char, etc.) in this declaration. The compiler ignored all but the first type.
- 171 2 You used more than one storage class (extern, static, auto, register, typedef) in this declaration. The compiler ignored all but the first storage class declaration.
- 172 2 You used more than one language type in this declaration (\$asm, \$p11, \$pascal, \$fortran). The compiler ignored all but the first language.
- 173 2 You used a storage class declaration (extern, static, auto, register, typedef) on a parameter, or on a member of a structure or union. The compiler ignored the storage class declaration.
- 174 2 You declared the array "\$" with a negative or zero bound. The compiler assumed the value 1.
- 175 3 The identifier "\$" does not begin a function definition.
- 176 3 You declared the identifier "\$" as an array with a function element type, which is illegal in C. You can, however, declare an array of pointers to functions.
- 177 3 You declared the identifier "\$" as a function that returned either an array or a function, which is illegal in C. You can, however, declare a function that returns a pointer to either an array or a function.
- 178 2 You declared the variable "\$" with storage type auto or register outside a function. The compiler assumed the extern storage class.
- 179 3 You have already declared "\$" as a variable, enumeration, or typedef in the current block.
- 180 2 You did not declare "\$" as a variable, enumeration, or typedef in any of the current blocks. The compiler gave it the attributes of an integer.
- 181 2 You have already declared "\$" as a structure, union, or enumeration in the current block. The compiler ignored this subsequent declaration.

- 182 1 You have not declared "\$" as a structure, union, or enumeration in any of the blocks within the current scope. You will not be able to reference any variable declared to be that structure, union, or enumeration.
- 183 3 You tried to define "\$" as a structure or union recursively (as a structure within itself). You can, however, define a pointer to a structure or union.
- 184 3 You tried to declare "\$" as a function within a structure or union, which is impossible. You may, however, declare a pointer to a function within a structure.
- 185 2 You did not specify the array bounds for "\$", either directly or implicitly within an initialization. You may omit only the first bound of an extern declaration or the first bound after a pointer. The compiler used a bound of 1.
- 186 2 You have already declared "\$" as a member of the current structure or union.
- 187 3 You have already declared the tag name "\$" for the current structure, union, or enumeration as the tag for a different type of object.
- 188 2 You declared a type without variables, but it was not a structure, union, or enumeration. This is meaningless and the compiler ignored it.
- 189 2 You declared a structure, union, or enumeration without a field name within an enclosing structure or union. The declaration was put in the scope of the current block.
- 190 3 You have already declared "\$" with a different storage attribute in the current block. You can only duplicate the declaration if the type and storage match that in the current block. The only mixture of storage types allowed are extern and \$builtin.
- 191 3 You have already defined the function "\$" in this compilation.
- 192 3 The member "\$" was not an integer type, and you specified a size field in bits. The compiler ignored the field size.
- 193 3 The bit field for "\$" was not between 1 and 32. The compiler ignored the size field.
- 194 3 You specified bit field padding beyond the bounds 0 through 32. The compiler ignored the padding.
- 195 2 You specified bit padding within a union. The compiler ignored it.

- 196 1 In the declaration of "\$", which was a member of a union, you specified a bit field length. The normal use of bit fields is within a struct, since the union members all share the same storage.
- 197 3 In the declaration of "\$", you declared an assembly language function to return a non-integer. This is illegal since the implementation returns 0 or 1 depending on whether the function skipped.
- 198 3 You declared the identifier "\$" as void when it was not a function. Only functions can be declared of type void.
- 199 3 You declared the local function "\$" to be an assembly language function. This implementation does not support functions written in C that are called with the assembly language calling sequence.
- 200 2 You declared "\$" to be a builtin function, but the compiler does not know about it. The compiler has assumed it is an extern instead.
- 201 3 You declared "\$" to be a builtin function, but you did not declare it to be a function.
- 202 3 You declared "\$" to be a builtin function, but your declaration has the function returning a different value than an int.
- 203 3 You declared "\$" to be a builtin function, but your declaration has the function returning a different value than a double.
- 204 3 You declared "\$" to be a builtin function, but your declaration has the function returning a different value than a pointer to a char.
- 205 3 You declared "\$" to be a builtin function, but your declaration has the function returning a different value than a pointer to a word-aligned item.
- 206 3 You declared "\$" to be a builtin function, but your declaration has the function returning a value when it should be a void function.
- 207 3 You cannot declare "\$" as a structure member, since it is defined to be a structure that hasn't been declared yet.
- 250 2 The compiler found a "default:" or "case <constant expression>:" label on a statement that is not within a "switch" statement. The compiler ignored the label.
- 251 3 You have already declared the label "\$" in the current function.

- 252 3 You have already declared the label "\$" in the same function as a variable, enumeration, or typedef type with the same name.
- 253 1 Warning: You defined the label "\$" within a local block that is not in the scope of the current local block. The compiler will not do any auto initializations for the block the label is defined in.
- 254 3 You tried to branch to an identifier "\$" defined as a variable, enumeration, or typedef type.
- 255 3 You did not define the label "\$" within the current function.
- 256 2 You declared the function "\$" as an auto or register function. Since it is impossible to compile functions on the stack or in a register, the compiler assumed an extern storage class instead.
- 257 2 You declared the function "\$" with a typedef storage class. Since it has a function body, the compiler assumed an extern storage class.
- 258 3 The conditional test for a while statement must evaluate to a scalar expression.
- 259 3 The conditional test for a do statement must evaluate to a scalar expression.
- 260 3 The conditional test for a for statement must evaluate to a scalar expression.
- 261 2 The compiler found a break not nested within a do, for, or while loop, or within a switch statement. The compiler ignored the break.
- 262 2 The compiler found a continue not nested within a do, for, or while loop. The compiler ignored the continue.
- 263 3 The parameter starting with "\$" exceeds the implementation limit of 255 parameters.
- 264 2 You have already listed the parameter "\$" in this procedure declaration. The compiler ignored the second instance.
- 265 2 You declared the variable "\$" in the parameter section, but did not list it in the parameter list. The compiler converted it to type auto.
- 266 2 You have already defined the case "\$" for the current switch. The compiler ignored this subsequent definition.
- 267 2 You have already defined a default case for the current switch statement. The compiler ignored this subsequent default.

- 268 3 The expression for this switch statement must have an integer or character value.
- 269 2 You did not define any cases for the switch statement.
- 270 1 Warning: The statement following the switch is not a compound statement.
- 271 2 You nested the case "\$" in a block within the statement after the switch. The compiler did not initialize any auto or register variables you specified.
- 272 3 The member "\$" does not have the same offset, size, or type in all of the structures or unions in the current scope.
- 273 3 The binary operator "\$" requires that the expression on the left side be of an integer or pointer data type.
- 274 3 "\$" is not a member within any structure or union in the current scope.
- 275 3 You cannot take the size of a function.
- 276 3 You cannot take the size of an array with unknown bounds.
- 277 3 You cannot take the size of a label.
- 278 3 You cannot pass more than 3 arguments to an assembly language procedure.
- 279 2 You attempted to pass an argument to an assembly language procedure that was not an integral or pointer type. The given value was ignored, and 0 passed instead.
- 280 2 You attempted to return a value from a void function. The return value was ignored.
- 281 1 Warning: The member name \$ was not a member of the structure on the left hand side of the period (.). This construct allows support of other versions of C, since the member name is at the same offset, and of the same type in all structures.
- 282 1 Warning: The member name \$ was not a member of the structure pointed to by the pointer on the left hand side of the ->. This construct allows support of other versions of C, since the member name is at the same offset, and of the same type in all structures.
- 283 3 You attempted to take the structure member of an item aligned on a byte or bit boundary. This compiler restricts all structures to be on word boundaries.

- 284 2 The member field \$ will not show up on the cross reference because the left hand side of the period operator was not a structure.
- 285 3 You attempted to use an expression evaluating to struct "\$" and it was never declared.
- 286 1 The identifier \$ was used in an #if or #assert preprocessor statement, and it was not defined. A 0 was used instead.
- 287 3 You cannot take the size of a struct/union that has not been declared.

Initialization Errors

- 310 3 You cannot initialize "\$" because it is a type definition.
- 311 3 The initialization for "\$" is nonconstant.
All static scalar initializations, and all initializations of arrays and structures must be constant.
- 312 3 In the initialization of "\$", you tried to assign an address constant to a double or float, which is illegal. You may have specified too many or too few initializers before the current one.
- 313 1 Warning: In the initialization of "\$", you tried to assign a floating-point constant to an integer or pointer value. The compiler converted the constant to an integer.
- 314 3 In the initialization of "\$", you tried to assign an address constant to a short int or short unsigned int, which cannot hold the full value of the address.
- 315 3 In the initialization of "\$", you tried to assign an address constant to a bit field within a structure or union.
- 316 3 In the initialization of "\$", you tried to initialize a structure where you have not yet declared the structure body.
- 317 3 In the initialization of "\$", you tried to assign an address constant to a variable of type char, which cannot hold the full value of the address.
- 318 2 In the initialization of "\$", you tried to assign the address of a char to an integer pointer. The offset used was on an odd byte boundary. The address holds the character before the character you intended.

- 319 1 In the initialization of "\$", you used a { and it is not an array or struct. The compiler effectively ignored the {.
- 320 2 You supplied \$ more initial values than there were variables or fields to initialize. The compiler ignored the excess elements.
- 321 3 You cannot initialize the function "\$".
- 322 2 In the initialization of "\$", you tried to assign more elements to the array than the declared bounds allowed. Since you declared the array inside another array, a struct, a union, or as a type definition, the compiler could not extend the bounds, and ignored the excess elements.
- 323 1 Warning: In the initialization of "\$", you assigned more elements to the array than the declared bounds allowed. The compiler extended the declared bounds to receive all of the elements initialized.
- 324 1 Warning: Other compilers may not give "\$" an initialization, since it involves initializing an array, or structure or union which has an auto or register storage class.
- 325 3 In the initialization of "\$", you declared the array without bounds, and the current initializer did not begin with a '{', so the compiler had no way of calculating the bounds. The compiler has skipped the rest of the initialization.
- 326 1 You initialized only the field "\$" when you initialized the union. Other C compilers may not allow the initialization of unions.
- 327 3 You have already initialized "\$".

Conversion Errors

- 350 3 The unary operator "\$" works only on integer, character, floating-point, and pointer data types.
- 351 3 The unary operator "\$" works only on character and integer data types.
- 352 3 The unary operator "\$" works only on character, integer, and floating-point data types.
- 353 3 The unary operator "\$" works only on pointer data types.
- 354 3 The unary operator "\$" requires that the data it operates on be a variable or array element, rather than an expression, constant, typedef name or array name.

- 355 3 You specified an illegal conversion.
- 356 2 The unary operator "&" would not ordinarily work on variables stored in registers. This implementation, however, treats the register storage class as equivalent to auto, so the operation succeeds.
- 357 1 Warning: Other C implementations may not allow you to take the address of a function or array name, since the name by itself gives the address.
- 358 3 You cannot apply the unary operator "&" to variables which are bit fields within a struct.
- 370 3 The binary operator "\$" works only on integer, floating-point and pointer data types.
- 371 3 The binary operator "\$" works only on integer or floating-point types.
- 372 3 The binary operator "\$" works only on integer data types.
- 373 3 The binary operator "+" allows only the combinations
pointer + integer
integer + pointer
when one of the operands is a pointer.
- 374 3 The binary operator "-" allows only the combinations
pointer - integer
pointer - pointer
when one of the operands is a pointer.
- 375 3 You specified an illegal conversion in an assignment expression.
- 376 3 The lefthand sub-expression for the binary operator "\$" must be a variable or array element. It cannot be an expression, constant, array name, or typedef name.
- 377 3 The binary operator "\$" allows only integer data types on the righthand side when the lefthand side is a pointer.
- 378 3 The subscript for an array must be an integer type.
- 379 3 The subscript operator "[]" requires that the left side of the expression be an array or a pointer value.
- 380 3 The function call operator "()" requires that the left side of the expression be a function name, or * of a pointer to a function.
- 381 2 A function call has more than the implementation limit of 255 arguments. The compiler ignored the remaining arguments.
- 382 3 This expression is too complicated for C to handle.

383 3 "\$" is not a structure or union.

384 3 "\$" is not a member of the structure or union.

385 3 You cannot convert one type of structure or union into another structure or union.

386 3 You tried to subtract 2 pointers, but the pointers did not point to the same object. This is meaningless, since C defines the difference of two pointers to be the number of objects between them.

387 1 Warning: You used an operation on a pointer valued expression that is not defined for pointers (|, &, ^, ~, <<, or >>). This compiler converted the pointer to unsigned first.

400 1 While converting a constant, the compiler detected a fixed-point overflow. The converted constant may not be accurate.

401 2 While converting a constant, the compiler detected a floating-point hardware error. The converted constant may not be accurate.

402 1 While converting a constant, the compiler attempted to convert an unsigned value larger than 65535 to the short unsigned data type.

403 1 While converting a constant, the compiler attempted to convert an unsigned value larger than 255 to the char data type.

404 1 While converting a constant, the compiler found an unsigned value larger than the bit field. It ignored the excess bits.

405 3 You cannot convert a data item to a function.

406 3 You cannot convert either a void function result or something previously converted to void to any other type.

407 1 While converting a constant, the compiler attempted to convert a value outside of the range -128..127 to the char type (and the /SIGNEDCHAR switch was used).

420 2 The context requires a constant integer expression. The constant "\$" was inserted by the compiler instead of the expression given.

421 2 While handling a constant subscript or adding a constant to a pointer, the compiler found a fixed-point overflow. The constant was too large.

440 2 While folding a constant int expression, the compiler noticed a fixed-point overflow and used the lower 32 bits of the expression for the result.

- 441 2 While folding a constant double expression, the compiler noticed a floating-point overflow. The result used may be meaningless.
- 442 2 While folding a constant double expression, the compiler noticed a floating-point underflow. The result used may be meaningless.
- 443 2 While folding a constant double expression, the compiler noticed a floating-point division by 0. The result used may be meaningless.
- 470 2 An argument to the builtin function "\$" was supposed to be an integer type, and it wasn't. The compiler will attempt to convert it to be an integer.
- 471 2 An argument to the builtin function "\$" was supposed to be a floating-point type, and it wasn't. The compiler will attempt to convert it to be a double.
- 472 3 An argument to the builtin function "\$" was supposed to be a pointer to a character, and it wasn't.
- 473 3 An argument to the builtin function "\$" was supposed to be a pointer to a word aligned item the size of an integer (2 words), and it wasn't.
- 474 3 There were not enough arguments to builtin function "\$".
- 475 3 There were too many arguments to builtin function "\$".
- 500 4 The compiler found a premature End of File.
- 501 4 The compiler found a NEW LINE unexpectedly in an #if or #assert.

Global Errors

- 502 4 The compiler found a syntax error.
- 503 4 Previous errors have lead to an unrecoverable state. Please correct all errors and resubmit the compilation. If this message persists, submit a STR (Software Trouble Report) to Data General Corporation.
- 504 3 The compiler has found an unrecognizable statement. It has continued compilation at the indicated token.

End of Appendix

APPENDIX B

A SYNTAX GRAPH FOR THE C LANGUAGE

/* In the following description of the syntax of the C programming language, brackets indicate an option, and braces indicate selection from a list of options separated by bars. Angle brackets enclose strings for which you must supply actual values. Lowercase indicates a C keyword. Read the colon as the "consists-of" relation. Parentheses, colons, commas, periods, brackets, and curly braces are also part of the C language, and as such are cited in quotation marks. The symbols surrounding this paragraph are the standard comment delimiters for the C language. Comments do not nest. */

Statement Syntax

=====

```
STATEMENT-LIST      :      STATEMENT [STATEMENT-LIST]

STATEMENT           :      [<identifier>":" ] STATEMENT

STATEMENT           :      { COMPOUND-STATEMENT
                           | CONDITIONAL-STATEMENT
                           | WHILE-STATEMENT
                           | DO-STATEMENT
                           | FOR-STATEMENT
                           | SWITCH-STATEMENT
                           | BREAK-STATEMENT
                           | CONTINUE-STATEMENT
                           | RETURN-STATEMENT
                           | GOTO-STATEMENT
                           | NULL-STATEMENT
                           | EXPRESSION-STATEMENT}

COMPOUND-STATEMENT   :      "{" [DECLARATION-LIST]
                           [STATEMENT-LIST] "}"

CONDITIONAL-STATEMENT :      if "(" EXP ")" STATEMENT
                           [else STATEMENT]

WHILE-STATEMENT      :      while "(" EXP ")"
                           STATEMENT

DO-STATEMENT         :      do STATEMENT while "(" EXP ")" ";"

FOR-STATEMENT        :      for "(" [INITIAL] ";" [TEST] ";" [INCR] ")"
                           STATEMENT

        INITIAL      :      EXP

        TEST         :      EXP
```

```

        INCR                :      EXP

SWITCH-STATEMENT          :      switch "(" EXP ")"
                           COMPOUND-STATEMENT

```

where the keyword

```

        default ":"

```

may precede at most one STATEMENT of the compound, and labels
with distinct constant expressions

```

        [case CONSTANT-EXP ":" ]

```

may precede other STATEMENTS of the compound.

```

BREAK-STATEMENT          :      break ";"

CONTINUE-STATEMENT       :      continue ";"

RETURN-STATEMENT         :      return [ "(" EXP ")" ] ";"

GOTO-STATEMENT           :      goto <identifier> ";"

NULL-STATEMENT           :      ";"

EXPRESSION-STATEMENT     :      EXP ";"

```

Expression Syntax =====

```

EXP                      :      { PRIMARY-EXP
                                | UNARY-EXP
                                | BINARY-EXP
                                | CONDITIONAL-EXP
                                | EXP "," EXP }

PRIMARY-EXP              :      { <constant>
                                | <string>
                                | PRIMARY-EXP "(" [ EXP-LIST ] ")"
                                | "(" EXP ")"
                                | <identifier>
                                | LVALUE }

EXP-LIST                 :      [ EXP-LIST "," ] EXP

```

```

UNARY-EXP      :      { "*" EXP
                       | "&" LVALUE
                       | "-" EXP
                       | "!" EXP
                       | "~" EXP
                       | "++" LVALUE
                       | "--" LVALUE
                       | LVALUE "++"
                       | LVALUE "--"
                       | "(" TYPE-NAME ")" EXP
                       | sizeof EXP
                       | sizeof "(" TYPE-NAME ")" }

BINARY-EXP     :      { EXP BINARY-OP EXP | LVALUE ASSGN-OP EXP}

LVALUE         :      { <identifier>
                       | PRIMARY-EXP "[" EXP "]"
                       | LVALUE "."<identifier>
                       | PRIMARY-EXP ":"<identifier>
                       | "*"EXP
                       | "(" LVALUE ")" }

BINARY-OP      :      { "*" | "/" | "%"
                       | "+" | "-"
                       | ">>" | "<<"
                       | "<" | ">" | { "<=" | "<" } | { ">=" | ">" }
                       | "==" | "!="
                       | "&"
                       | "^"
                       | "|"
                       | "&&"
                       | "||" }

CONDITIONAL-EXP :      EXP?"EXP":EXP

ASSGN-OP       :      { "="
                       | { "+=" | "+=" }
                       | { "-=" | "-=" }
                       | { "*=" | "*=" }
                       | { "/=" | "/=" }
                       | { "%=" | "%=" }
                       | { ">>=" | ">>=" }
                       | { "<<=" | "<<=" }
                       | { "&=" | "&=" }
                       | { "^=" | "^=" }
                       | { "|=" | "|=" } }

CONSTANT-EXP   :      EXP

```

where EXP must evaluate to a constant expression.

Declaration Syntax
=====

```

DECLARATION-LIST      :      DECLARATION
                           [DECLARATION-LIST]

DECLARATION           :      DECL-SPECIFIERS
                           [DECLARATOR-LIST]

DECL-SPECIFIERS       :      { TYPE-SPECIFIER [DECL-SPECIFIERS]
                           | SC-SPECIFIER [DECL-SPECIFIERS]
                           | LANG-SPECIFIER [DECL-SPECIFIERS] }

SC-SPECIFIER          :      { auto
                           | static
                           | extern
                           | register
                           | typedef
                           | $builtin }

TYPE-SPECIFIER        :      { INT-TYPE-SPECIFIER
                           | DOUBLE-TYPE-SPECIFIER
                           | STRUCTURE-SPECIFIER
                           | UNION-SPECIFIER
                           | ENUM-SPECIFIER
                           | TYPEDEF-NAME
                           | void }

INT-TYPE-SPECIFIER    :      { [unsigned] char
                           | int
                           | short [int]
                           | long [int]
                           | [{ short | long }] unsigned [int]
                           | unsigned [{ short | long }] [int] }

DOUBLE-TYPE-SPECIFIER :      [{ short | long }] { double | float }

STRUCTURE-SPECIFIER   :      struct { <identifier>
                           | <identifier> "{"STRUCT-DECL-LIST"}"
                           | "{"STRUCT-DECL-LIST"}" }

STRUCT-DECL-LIST      :      STRUCT-DECLARATION [STRUCT-DECL-LIST]

STRUCT-DECLARATION    :      TYPE-SPECIFIER STRUCT-DECLARATOR-LIST";"

STRUCT-DECLARATOR-LIST :      STRUCT-DECLARATOR ["," STRUCT-DECLARATOR-LIST]

STRUCT-DECLARATOR     :      { DECLARATOR
                           | DECLARATOR ":" CONSTANT-EXP
                           | ":" CONSTANT-EXP }

UNION-SPECIFIER       :      union { <identifier>
                           | <identifier> "{"UNION-DECL-LIST"}"
                           | "{"UNION-DECL-LIST"}" }

```

```

UNION-DECL-LIST      :      UNION-DECLARATION [UNION-DECL-LIST]

UNION-DECLARATION    :      TYPE-SPECIFIER UNION-DECLARATOR-LIST";"

UNION-DECLARATOR-LIST :      UNION-DECLARATOR ["," UNION-DECLARATOR-LIST]

UNION-DECLARATOR     :      DECLARATOR [ ":" CONSTANT-EXP ]

ENUM-SPECIFIER       :      enum      { <identifier>
                                      | <identifier> "{"ENUM-DECL-LIST"}"
                                      | "{"ENUM-DECL-LIST"}" }

ENUM-DECL-LIST       :      [ENUM-DECL-LIST ","] ENUMERATOR

ENUMERATOR           :      <identifier> ["=" CONSTANT-EXP ]

DECLARATOR-LIST      :      INIT-DECLARATOR ["," DECLARATOR-LIST]

INIT-DECLARATOR      :      DECLARATOR [INITIALIZER]

DECLARATOR           :      { <identifier>
                              | "(" DECLARATOR ")"
                              | "*" DECLARATOR
                              | DECLARATOR "(")"
                              | DECLARATOR "[" [ CONSTANT-EXP ] "]" }

INITIALIZER          :      "="      { EXP
                                      | "{" INITIALIZER-LIST "}"
                                      | "{" INITIALIZER-LIST "," "}" }

INITIALIZER-LIST     :      { EXP
                              | INITIALIZER-LIST "," INITIALIZER-LIST
                              | "{" INITIALIZER-LIST "}" }

TYPE-NAME            :      TYPE-SPECIFIER ABSTRACT-DECLARATOR

ABSTRACT-DECLARATOR  :      [{ "(" ABSTRACT-DECLARATOR ")"
                              | "*" ABSTRACT-DECLARATOR
                              | ABSTRACT-DECLARATOR "(")"
                              | ABSTRACT-DECLARATOR [ CONST-EXP ]}]

TYPEDEF-NAME         :      <identifier>

LANG-SPECIFIER       :      { $asm
                              | $fortran
                              | $pascal
                              | $pl1 }

```

Module Syntax

=====

```

PROGRAM                :      EXTERNAL-DEFINITION [PROGRAM]

EXTERNAL-DEFINITION    :      { FUNCTION-DEFINITION
                                | DATA-DEFINITION
                                | ";" }

FUNCTION-DEFINITION    :      [TYPE-SPECIFIER] FUNCTION-DECLARATOR
                                FUNCTION-BODY

FUNCTION-DECLARATOR    :      DECLARATOR "["[PARAMETER-LIST]"

PARAMETER-LIST         :      <identifier> ["," PARAMETER-LIST]

FUNCTION-BODY          :      TYPE-DECL-LIST FUNCTION-STATEMENT

FUNCTION-STATEMENT     :      "{" [DECLARATION-LIST]
                                [STATEMENT-LIST] "}"

DATA-DEFINITION       :      [DECL-SPECIFIERS] [DECLARATOR-LIST]";"

```

Preprocessor Syntax

=====

```

#define <identifier>    <token-string><NL>
#define <identifier> "("<identifier-list>)"<token-string><NL>
#undef <identifier><NL>
#include "filename"<NL>
/* C quotation marks around filename this line only */
#include <filename><NL>
/* C angle brackets around filename this line only */
#if CONSTANT-EXP<NL>
#ifdef <identifier><NL>
#ifndef <identifier><NL>
#else<NL>
#endif<NL>
#undef <identifier><NL>
#[line] <constant> <identifier><NL>
/* line is optional */
#list<NL>
#nolist<NL>
#nohref<NL>
#xref<NL>
#assert CONSTANT-EXP<NL>

```

End of Appendix

APPENDIX C

IMPLEMENTATION CHARACTERISTICS

This appendix summarizes differences and extensions in functionality that distinguish Data General's implementation of the C language. It assumes that the de facto standard for C is the language described in

Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language (Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978)

as a point of departure.

Hardware Representations

This section notes details of hardware implementation.

Date Sizes and Machine Dependencies

Data General's implementation of the C language supports the additional data types short unsigned integers, long unsigned integers, unsigned integers, and unsigned characters.

Integer Types

short int is a 16-bit quantity (-32,768 to +32,767).

long int is a 32-bit quantity (-2,147,483,648 to +2,147,483,647).

int is also a 32-bit quantity (-2,147,483,648 to +2,147,483,647).

unsigned int is a 32-bit quantity (0 to +4,294,967,295).

short unsigned is a 16-bit quantity (range 0 to 65,535).

long unsigned is a 32-bit quantity (0 to +4,294,967,295).

Character Types

The char type in Data General's C language is an 8-bit quantity (0 to +255), as opposed to some other implementations, which might use it as a signed 8-bit quantity (-128 to +127). The unsigned char type is functionally the same as char. If you use

the /SIGNEDCHAR switch on the compilation, normal characters will hold integers in the range -128 to 127. Unsigned characters continue to have the range 0 to 255.

Data General's C language uses the full ASCII character set.

Real Types

float is a 32-bit quantity (range $\sim 1E-79$ to $\sim 1E75$).

double is a 64-bit quantity (range $\sim 1E-79$ to $\sim 1E75$).

Character Constants

You may specify up to four characters in a character constant. If you specify more than one character, the last character specified is in the lowest (rightmost) byte.

If you use the /SIGNEDCHAR switch and the first character of a character constant ≥ 200 (octal), the character is sign-extended.

Pointers

Pointers to characters and pointers to other objects do not have the same hardware representations. Programs that use pointers as integers and pass them around might therefore not work under Data General's implementation of the C language. For example, in

```
int    i, *ip;
char    *cp;

i = ip;
cp = i;
```

or

```
int    i, *ip;
char    *cp;

i = cp;
ip = i;
```

or

```
int    *ip;
char    *cp;

cp = (int) ip;
```


the three examples store the wrong value into cp and ip.

This also means that taking the address of a char aligned on an odd-byte boundary and converting it to a word pointer will point to the wrong byte if converted back to a char pointer.

The struct Type

Bit fields within structs are allocated left-to-right as opposed to some other compilers, which allocate them right-to-left. Bit fields can straddle two 16-bit words, but they are aligned to the next word boundary to prevent straddling three 16-bit words.

Implementation Differences

=====

This section sketches known differences between the Data General implementation of the C language and that described in Kernighan and Ritchie (1978).

Line Length

The maximum line length is 136 characters, including the NEW LINE character. A TAB counts as a single character.

Identifiers

This section describes differences and extensions of identifiers.

Lengths

Identifiers may have a significant maximum length ranging from 8 to 32 characters, depending on the setting of the /UNIQUE switch, described in Chapter 10. The default maximum length is 32 characters.

Number of Arguments

The maximum number of arguments to a function or preprocessor macro is 255.

Legal Characters

The characters \$ and @ are also legal identifier characters. You may want to avoid external identifiers beginning with a dollar sign (\$) or underscore (_). Various modules in the libraries use identifiers beginning with these characters for internal entry points and data. If any keywords specific to the Data General implementation are added in the future, the identifier will be preceded by a \$.

Externals

The LINK program supports both upper- and lowercase externals up to 32 characters. Unlike some other C compilers, Data General's C compiler does not prefix an underscore (_) before external names.

Keywords

Beyond the usual set, Data General's C language has additional keywords.

Additional Keywords

Data General's implementation of the C language uses the additional keywords

- enum
- void
- \$asm
- \$fortran
- \$pascal
- \$pl1

Reserved Keywords

Data General has reserved certain identifiers for future implementation as keywords. Do not attempt to use as your own identifier:

- entry

String Constants

The maximum length of a string constant is 255 characters plus a trailing null byte.

Initializations

You can initialize the addresses of externals, unions, and auto structs and auto unions.

unions

The AOS/VS C compiler permits you to initialize a union, unlike some other C compilers. The interpretation is to initialize the first element of the union. That is,

```
static union foo {
    int a;
    double b;
} u = { 1 };
```

initializes u.a to 1.

Storage Type auto structs and unions

The AOS/VS C compiler also allows you to initialize auto structs and auto unions. All of the elements of either must be constant. When the block is entered, the initialized values are copied into the struct or union.

```
auto struct foo { int a; double b; } s = { 1, 2.0 };
```

initializes s.a to 1 and s.b to 2.0 each time you enter the block.

Expressions

Data General's C language relaxes the order of components of assignment operators, but otherwise adheres to the usual conventions on expressions.

Operators

The compiler recognizes the suffixed operators (`=+`, `=-`, `=*`, `=/`, `=%`, `=<<`, `=>>`, `=&`, `=|`, `=^`) as equivalent, respectively, to the prefixed operators (`+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `|=`, `^=`), but issues a level 1 warning on each use.

The compiler recognizes the relational operator `=>` as synonymous with `>=`, and `=<` as synonymous with `<=`.

Labels

You cannot use a label in an expression.

Additional Features

=====

This section lists enhancements to the usual C functionality.

Identifiers

You may use the commercial at character (`@`) and dollar sign (`$`) in identifiers in addition to the normal alphanumeric and underscore (`_`) characters. The SWAT debugger does not recognize the commercial at character (`@`) as a valid identifier character, however. It reserves the character for its own use.

structs

You can move structs, reuse a tag field, and do some forward referencing of structs and unions.

Moving structs

You may move a struct. That is, you may assign one struct to another as in

```
    struct1 = struct2;
```

You may pass structs to functions, which copy by value. Functions can return structs.

Reusing the Tag Field

You may reuse the same tag field in different structs. They need not have the same offset or be of the same data type, as in other C implementations. The Data General C compiler will use the name as appropriate for the struct. If you use the `->` operator, and either the lefthand side is not a pointer to a struct or union, or the identifier on the righthand side is not a member of that struct or union, the compiler attempts to use the earlier definition of structs, searches through all member names with the same name as the righthand side identifier, and allows the operation if the name is used the same way in all declarations. Similarly, if the righthand side of the `.` operator is not a structure, or the identifier on the lefthand side is not a member of a particular structure, the earlier C definition is used.

Forward Referencing

The Data General C compiler allows very limited forward referencing of structs and unions. You can declare a pointer to a struct or union; if the struct or union is not already defined within the current scope, the compiler will wait until the end of the declaration block or the start of a function before reporting the error. At that time, it again checks to see whether that struct has been declared. If so, it allows the declaration. This is useful if you have two structs that have pointers to each other.

Conversions

The `/SHORTARITHMETIC` switch on the compilation command line allows you to use narrow (16-bit integer) arithmetic for types `short` and `short unsigned`, and single-precision (32-bit floating) arithmetic for the type `float`. This switch produces shorter or faster code for the given types, but accuracy suffers.

The `/NOSHORTARITHMETIC` switch allows you to perform implicit type conversions before carrying out arithmetic operations: the compiler automatically converts `short` and `char` to `int`, `short unsigned` to `unsigned`, and `float` to `double`. `/NOSHORTARITHMETIC` is the default value for this pair of switches.

Additional Data Types

Data General's implementation of the C language allows for an unsigned integer type, a complex enumeration type, and the void type.

The unsigned Types

You may declare variables of the types short unsigned and long unsigned as in

```
short unsigned [int] sua;
```

```
long unsigned [int] lua;
```

You may also declare variables of the type unsigned char, which is functionally the same as the regular char type. This is provided to be compatible with other C compilers.

Enumerations

Enumerations are unique complex types with named constants. They serve in part to replace the use of constants introduced by #define in the C language. They offer the advantage of scoped constant names. The syntax for the type enum is

```
ENUM-SPECIFIER :      enum      { <identifier>
                                | <identifier> "{" ENUM-DECL-LIST "}"
                                | "{" ENUM-DECL-LIST "}" }
```

```
ENUM-DECL-LIST :      [ENUM-DECL-LIST ","] ENUMERATOR
```

```
ENUMERATOR      :      <identifier> [ "=" CONSTANT-EXP ]
```

For example,

```
enum rainbow { red, orange, yellow, green, blue, violet };
enum rainbow hue = orange;
```

declares the enumerated type rainbow and a variable called hue with an initial value of orange. The name rainbow is in the same pool as names for struct and union types. The names for the items in the enumeration (orange, yellow, ...) come from the normal identifier pool.

The void Type

The type void allows you to declare a function as returning nothing. This use prevents meaningless values in the register as a result return. You may use void

- * as the return value of a function, signaling that the function does not return anything

```
void foo();
```

- * as an explicit type change, to ignore the result of the current expression

```
(void) foo2( ... );
```

The expressions

```
void foo3;  
void *foo4();
```

are illegal, since only functions can be void, and you cannot specify that a pointer be void.

Initializations

Data General's C language has an extended set of initialization strategies.

Functions with extern Variables

You may initialize extern variables within functions.

Constants in auto Arrays and Structures

You may initialize auto arrays and structs. For example, you can initialize an array with

```
auto fib[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89};
```

or a structure with

```
auto struct {  
    char    name[7];  
    int     address;  
} dial = { "Martin", 3026 };
```

Like the initialization of static/extern arrays and structs, all of the elements within an auto initialization of a struct or array must be constant.

Characters

=====

Characters in Data General's C language have the additions listed here.

Vertical Tab

You may use the vertical tab ('\013') wherever you can use a space. You can also use '\v' to represent this character within strings and character constants.

Uppercase Escapes

You may use the uppercase equivalent of the escaped special characters within strings and character constants ('\f' is the same as '\F').

Escape in Arguments

A backslash preceding string and character constants within the compiler hides any special meaning given to the next character in processing macros. For example, you could pass the comma operator to a macro with

```
macro-name( \, );
```

Format Padding

The Data General implementation of C now accepts the format options %<n>t and %@t for the fprintf, printf, and sprintf functions. Specify blank padding to column n with %<n>t. (You always generate at least one blank.) Specify the value of the next argument for the column number with %@t.

Constant Folding

=====

The Data General C compiler folds all constant floating point expressions at compile time.

Calling Procedures in Other Languages from C

=====

You may call procedures written in other Data General 32-bit languages by using the \$pl1, \$pascal, or \$fortran attributes in declaring the function name. The C compiler will generate the correct call, but it will not pass dope vectors to PL/1 or FORTRAN procedures. To preserve the call-by-value calling sequence of the C language, the arguments are copied into temporaries and the address of the temporary pushed unless an address was actually passed, in which case the actual address is pushed.

To call assembly language routines, you might want to use the \$asm attribute in declaring a function name. You can pass at most three arguments, which must be either integer or pointer type. The return value must be an integer (0 if the routine skipped, 1 if it did not skip). The arguments are passed in accumulators AC0, AC1, and AC2.

Preprocessor Statements

The Data General implementation of the C language provides additional preprocessor statements not found in other implementations.

You can check conditions for compilation with

```
#assert CONSTANT-EXP<NL>
```

which returns an error if the constant-exp is zero. For example,

```
#assert sizeof(foo) > sizeof(bar)<NL>
```

returns an error unless foo is larger than bar.

For error diagnostics, a statement of the form

```
#      <constant> <identifier><NL>
```

is equivalent to a statement of the form

```
#line      <constant> <identifier><NL>
```

The compiler takes the constant as the line number of the

next source line, and the identifier as the name of the current input file. The current input file does not change if you omit the identifier.

You can turn a listing on with

```
#list<NL>
```

and turn it off with

```
#nolist<NL>
```

You can turn off cross referencing at the next executable statement with

```
#noxref<NL>
```

and reverse the effects of a previous #noxref at the next executable statement with

```
#xref<NL>
```

If you did not specify the /XREF switch at compilation time, this command has no effect.

Shared Page I/O

The Data General implementation of C now accepts the fopen option of "s" to open a file as a shared page file, which is implemented via shared memory. The user must reserve one or more pages with the /RESERVE- switch on the CCL link macro.

AOS/RT32

To run on AOS/RT32, the program must conform to the subset of AOS/VS system calls that AOS/RT32 supports. In particular, you should open disk files with the "b" option for block I/O (?GOPEN, ?RDB, ?WDB, ?GCLOSE system calls), rather than with other options which use record I/O. Block I/O also runs under AOS/VS, but you do not have to use it for disk files.

End of Appendix

APPENDIX D

ASCII CHARACTER SET

To find the *octal* value of a character, locate the character, and combine the first two digits at the top of the character's column with the third digit in the far left column.

LEGEND:

Character code in decimal
EBCDIC equivalent hexadecimal code
Character

		10_
0	64	@
	7C	

OCTAL	00_	01_	02_	03_	04_	05_	06_	07_
0	0 00 NUL	8 BS (BACK-SPACE)	16 DLE P	24 CAN X	32 SPACE	40 (	48 0	56 8
1	1 01 SOH A	9 HT (TAB)	17 DC1 Q	25 EM Y	33 !	41)	49 1	57 9
2	2 02 STX B	10 NL (NEW LINE)	18 DC2 R	26 SUB Z	34 " (QUOTE)	42 .	50 2	58 :
3	3 03 ETX C	11 VT (VERT. TAB)	19 DC3 S	27 ESC (ESCAPE)	35 #	43 +	51 3	59 ;
4	4 04 EOT D	12 FF (FORM FEED)	20 DC4 T	28 FS 	36 \$	44 , (COMMA)	52 4	60 <
5	5 05 ENQ E	13 RT (RETURN)	21 NAK U	29 GS 	37 %	45 -	53 5	61 =
6	6 06 ACK F	14 SO N	22 SYN V	30 RS 	38 &	46 (PERIOD)	54 6	62 >
7	7 07 BEL G	15 SI O	23 ETB W	31 US 	39 ' (APOS)	47 /	55 7	63 ?

OCTAL	10_	11_	12_	13_	14_	15_	16_	17_
0	64 7C @	72 C8 H	80 D7 P	88 E7 X	96 79 ' (GRAVE)	104 88 h	112 97 p	120 A7 x
1	65 C1 A	73 C9 I	81 D8 Q	89 E8 Y	97 81 a	105 89 i	113 98 q	121 A8 y
2	66 C2 B	74 D1 J	82 D9 R	90 E9 Z	98 82 b	106 91 j	114 99 r	122 A9 z
3	67 C3 C	75 D2 K	83 E2 S	91 8D [	99 83 c	107 92 k	115 A2 s	123 C0 }
4	68 C4 D	76 D3 L	84 E3 T	92 E0 \]	100 84 d	108 93 l	116 A3 t	124 4F
5	69 C5 E	77 D4 M	85 E4 U	93 9D ^	101 85 e	109 94 m	117 A4 u	125 D0 }
6	70 C6 F	78 D5 N	86 E5 V	94 5F _ or ~	102 86 f	110 95 n	118 A5 v	126 A1 ~ (TILDE)
7	71 C7 G	79 D6 O	87 E6 W	95 6D - or -	103 87 g	111 96 o	119 A6 w	127 07 DEL (RUBOUT)

SD-00217 Character code in octal at top and left of charts.

| means CONTROL

End of Appendix

APPENDIX E

OCTAL AND HEXADECIMAL CONVERSION

	8 ¹	8 ²	8 ³	8 ⁴	8 ⁵	8 ⁶
0	0	0	0	0	0	0
1	32,768	4,096	512	64	8	8
2	65,536	8,192	1,024	128	16	16
3	98,304	12,228	1,536	192	24	24
4	131,072	16,384	2,048	256	32	32
5	163,840	20,480	2,560	320	40	40
6	196,608	24,576	3,072	384	48	48
7	229,376	28,672	3,584	448	56	56

SD-00973

	16 ⁵	16 ⁴	16 ³	16 ²	16 ¹	16 ⁰
0	0	0	0	0	0	0
1	1,048,576	65,536	4,096	256	16	1
2	2,097,152	131,072	8,192	512	32	2
3	3,145,728	196,608	12,288	768	48	3
4	4,194,304	327,680	16,384	1,024	64	4
5	5,242,880	393,216	20,480	1,280	80	5
6	6,291,456	458,752	24,576	1,536	96	6
7	7,340,032	524,288	28,672	1,792	112	7
8	8,388,608	589,824	32,768	2,048	128	8
9	9,437,184	655,360	36,864	2,304	144	9
A	10,485,760	720,896	40,960	2,560	160	10
B	11,534,336	786,432	45,056	2,816	176	11
C	12,582,912	851,968	49,152	3,072	192	12
D	13,631,488	917,504	53,248	3,328	208	13
E	14,680,064	983,040	57,344	3,584	224	14
F	15,728,640		61,440	3,840	240	15

SD-00974

End of Appendix



INDEX

Within this index, the letter "f" following a page entry means "and the following page"; "ff" means "and the following pages".

in column 1 9-2
! (logical NOT) 8-25
!= (inequality operator) 8-21
(number sign) 2-2
(preprocessor delimiter) 2-11
#assert 9-1, 9-11
#define (preprocessor statement) 2-3, 9-1, 9-2ff, 11-3
#else 9-1, 9-6, 11-2
#endif 9-1, 9-7, 11-2
#if 9-1, 9-6, 11-2
#ifdef 9-1, 9-6, 11-2
#ifndef 9-1, 9-6, 11-2
#include (preprocessor statement) 2-11, 9-1, 9-5, 11-4
#line 9-1, 9-7
#list (preprocessor statement) 2-11, 9-1, 9-7
#nolist (preprocessor statement) 2-19, 9-1, 9-7
#noxref 9-1, 9-11
#undef (preprocessor statement) 2-11, 9-1, 9-4, 11-3
#xref 9-1, 9-11
\$ (dollar sign) 2-1
\$asm 6-10ff
\$builtin 7-2, 3-19, 3-22
\$builtin functions 10-6
\$fortran 6-10ff
\$pascal 6-10ff
\$pll 6-10ff
% (modulus operator) 8-27
& (address operator) 8-6
& (bitwise AND operator) 8-9
&& (logical AND operator) 8-24
(and) (parentheses) 4-4f
* (indirection operator) 8-19
* (multiplication operator) 8-28
+ (addition operator) 8-5
++ (increment operator) 8-18
, (comma) as punctuation 4-11, 2-1
, (comma operator) 8-38
- (subtraction operator) 8-35
- (unary minus operator) 8-37
-- (decrement operator) 8-13
-> (structure pointer operator) 8-34
. (structure member operator) 8-33
/ (division operator) 8-14
/* and */ (comment delimiters) 2-12
/BACKSPACE= switch 11-7
/CHECKOFFSET switch 11-7
/CODE switch 11-7
/CONDITIONAL switch 11-7

/DEBUG switch 11-7
/DEFINE switch 11-15
/DISALLOWANCIENT switch 11-7
/E= switch 11-7
/ERRORCOUNT= switch 11-7
/EXTERNMAP switch 11-8
/EXTL switch 11-8, 7-18, 3-21
/FIXEDOVERFLOW switch 11-9
/FLAGDEFINES switch 11-9
/FRAMESIZE switch 11-9
/INCDIRECTORY= switch 11-9
/INCLUDE switch 11-15
/INCLUDES switch 11-9
/L= switch 11-9
/LINEID switch 11-9
/LINK switch 11-9
/LOWERCASE switch 11-10
/LS switch 11-10
/MAP switch 11-10
/N switch 11-10
/NOCHECKOFFSET switch 11-10
/NOCODE switch 11-10
/NOCONDITIONAL switch 11-10
/NODEBUG switch 11-10
/NODISALLOWANCIENT switch 11-10
/NOEXTERNMAP switch 11-11
/NOEXTL switch 11-11
/NOFIXEDOVERFLOW switch 11-11
/NOFLAGDEFINES switch 11-12
/NOFRAMESIZE switch 11-12

/NOINCLUDES switch 11-12
/NOLINEID switch 11-12
/NOLOWERCASE switch 11-12
/NOMAP switch 11-12
/NOOPTIMIZE switch 11-12
/NOPOINTERCHECK switch 11-12
/NOPROCID switch 11-12
/NOSEA switch 11-17
/NOSHAREDSTRING switch 11-12
/NOSHORTARITHMETIC switch 11-12
/NOSIGNEDCHAR switch 11-12
/NOSTATISTICS switch 11-12
/NOSUBCHECK switch 11-12
/NOUNDECLARED switch 11-12
/NOWARNINGS switch 11-12
/NOXREF switch 11-12
/O= switch 11-13
/OPTIMIZE= switch 11-13
/PARTCODE= switch 11-13
/PARTSHORT= switch 11-13
/PARTSTATIC= switch 11-13
/POINTERCHECK switch 11-13
/PROCID switch 11-14
/RESERVE switch 10-101
/REVISION= switch 11-14
/RT32 switch 11-17
/SEARCH switch 11-15
/SHAREDIO switch 11-17
/SHAREDSTRING switch 11-14
/SHORTARITHMETIC switch 11-14
/SHORTSTATIC= switch 11-14
/SIGNEDCHAR switch 11-14, 7-7, 3-18
/STATISTICS switch 11-14

/SUBCHECK switch 11-14
 /TEMPDIR= switch 11-14
 /UNDECLARED switch 11-14
 /UNIQUE= switch 11-15, 2-4
 /WARNINGS switch 11-15
 /XREF switch 11-15
 : (colon) in labels 5-10
 ; (semicolon) with statements 2-12
 < (less-than operator) 8-22
 << (shift left operator) 8-30
 <= (less-than-or-equal-to operator) 8-23
 = (assignment operator) 8-7
 == (equality operator) 8-15
 > (greater-than operator) 8-16
 >= (greater-than-or-equal-to operator) 8-17
 >> (shift right operator) 8-30
 ?: (conditional operator) 8-12, 2-10
 @ (commercial at sign) in indentifiers 2-1
 [and] (brackets) 4-4
 \ (backslash) 2-2
 ^ (bitwise exclusive OR operator) 8-10
 ~ (ones complement operator) 8-29
 _ (underscore) 2-1
 _NEXPAND macro 10-9
 _NOASSERT macro 10-9
 _NOBIT macro 10-9
 _NOMASK macro 10-9
 _tolower() 10-325
 _toupper() 10-329
 | (bitwise inclusive OR operator) 8-11
 || (logical OR operator) 8-26

A

abort() 10-21
 abs() 10-23
 acos() 10-25
 additional features C-6ff
 additional characters C-10
 additional data types C-8f
 additional initializations C-9f
 constant folding C-11
 conversions C-7
 identifiers C-6
 structs C-6
 addition operator (+) 8-5
 address operator (&) 8-6
 Advanced Operating System/Virtual Storage 1-2
 AGENT 10-101
 aggregate data 7-37, 3-11
 alloc() 7-34, 10-27
 alloca() 10-29
 any() 10-31
 AOS/RT32 C-12, 10-101
 AOS/VS 1-2

- argc argument 6-6
- arguments to programs 11-26
- argv argument 6-6
- arithmetic conversions, the usual 3-16f
- arithmetic expressions 4-6f
- arrays 3-8f
 - declaring size of 3-9
 - first element of 3-8
 - initializations of 3-22f
 - multi-dimensional 3-8
 - name of as pointer 3-8
 - of char 7-7
 - with sizeof operator 7-34
- ASCII 7-7, 3-18
- asin() 10-33
- assembly language, calling sequence between C and 6-16ff
- assert() 10-35
- assertions, testing 9-11
- assignment, data type conversion by 3-17
- assignment expressions 4-9f
- assignment operator (=) 8-7
- at sign, commercial (@) 2-1
- atan() 10-36
- atan2() 10-38
- atof() 10-40
- atoi() 10-42
- atol() 10-44
- atou() 10-46
- auto 7-3, 3-19, 3-22

B

- backslash (\) 2-2
 - for escape sequences 2-8
- binary trees 6-9
- bits in C
 - fields of 3-12
 - specifying with struct 7-38
- bitwise AND operator (&) 8-9
- bitwise inclusive OR operator (|) 8-11
- bitwise exclusive OR operator (^) 8-10
- bitwise expressions 4-8f
- block structure in C 6-1ff, 3-2
 - how to create 6-2f
- blockmove() 10-48
- braces ({ and }) 2-12
 - for compound statements 5-11
- brackets ([and]) 4-4f
 - see also arrays
- break 7-4, 5-9
- builtin functions 10-6

C

- C language
 - fundamentals of 1-2ff
 - introduction to 1-1
 - programming style 1-7
- calling sequences 6-10ff
 - assembly language 6-16ff
 - C 6-14ff
 - VS/ECS 6-11ff
- calling procedures in other languages C-11
- calloc() 10-50
- case 7-5, 5-5f
- casts 3-18
- ceil() 10-52
- channel() 10-54
- char 7-7, 3-6
 - data type conversion of 3-16
 - range of possible values 3-18
 - sign extension for 3-18
 - size of 7-7
- character set, for C 2-1
- characters in C 2-1ff
 - character set, C 2-1
 - commercial at sign (@) 2-1
 - constants 2-7f
 - dollar sign (\$) 2-1
 - number sign (#) 2-2
 - underscore (_) 2-1
 - uppercase and lowercase 2-2
- character-handling macros 10-5
- colon (:)
 - specifying bit fields with 7-38
 - in labels 5-10
- comma (,) 4-11, 2-1
- comma operator (,) 8-38
- compilation 11-1ff
 - CC macro 11-16ff
 - command line 11-1f
 - conditional compilation 11-2f
 - global switches 11-5ff
 - include files 11-4
 - local switches 11-7, 11-15
- compound assignment operators 4-10, 8-8
- compound statements 5-11, 7-40
- conditional compilation 9-5f
 - end a conditional compilation 9-7
 - specify an alternative 9-6
 - test a definition 9-6
 - test an expression, 9-6
- conditional operator (?:) 8-12, 2-10, 4-10
- constants in C 2-6ff
 - character 2-7

- decimal 2-5
- enums producing 3-14
- exponents for 2-8
- hexadecimal 2-6
- long integer 2-8
- octal 2-5
- radix of 2-6
- string 2-8
- symbolic (using #define) 2-9
- continue 7-9, 5-9
- conversion
 - arithmetic, the usual 3-16f
 - data type 3-16ff
- conversion functions 10-2
- cos() 10-55
- cosh() 10-57
- COUNTCHAR4.C 1-7
- createdir() 10-59
- curdir() 10-61

D

- data 3-1ff
 - aggregate 7-37, 3-11
 - passing to and from functions 6-8
 - see also data types
- Data General C, introduction to 1-1
- data structures, dynamic 3-11f
- data types in C 3-1ff
 - comparison of, in C and other languages 6-18
 - conversions 3-16ff
 - by assignment 3-17
 - by casts 3-18
 - by passing arguments to functions 3-18
 - creating synonyms for 7-41
 - see also typedef
 - defaults for 3-23
 - derived 3-8ff, 3-3
 - for other-language functions 6-17
 - fundamental 3-4ff
 - keywords for specifying 3-2ff
- debugging aids 11-26ff
 - assembly language debugger 11-28
 - SWAT native language debugger 11-27
- decimal constants 2-5
- declaration syntax B-4f
- declarations of data types in C 3-1ff
 - syntax of 3-2
 - where can occur 3-2
- decrement operator (--) 8-13, 4-5f
- default 7-10, 7-5
- defaults of data types and storage classes 3-22f

delimiters

- comment (/* and */) 2-12
- preprocessor (#) 2-1
- statement (semicolon (;) and braces ({ and })) 2-12
- dg_ctime() 10-63
- dg_localtime() 10-65
- dir() 10-67
- division operator (/) 8-14
- do 7-11, 5-8
- double 7-12, 3-7
 - comparison with other-language data types 6-18
 - function declared as 6-5
 - rounding and truncating of 7-20, 3-17

E

- E, to show exponent 2-8
- else 7-13, 5-3
- entry 7-15
- enum 7-16, 3-14
 - and #define constants 3-14
- EOF character 3-18
- equal() 10-68
- equality operator (==) 8-15
- error handling 11-28f
- error messages A-1ff
 - code generation errors A-6
 - command line errors A-1ff
 - conversion errors A-13ff
 - global errors A-16
 - initialization errors A-12f
 - parser errors A-7ff
 - preprocessor errors A-3ff
 - token errors A-5f
 - severity levels A-1
- escape sequences 2-8
- evaluation, order of 4-7
 - see also the operator or type of expression
- Example C program 1-7
- executing C programs 11-24ff
 - file redirection 11-25
 - program arguments 11-26
- exit() 10-70
- exp() 10-72
- exponent, to show 2-8
- expression syntax B-2f
- expressions 4-1ff
 - arithmetic 4-6f
 - assignment 4-9f
 - bitwise 4-8f
 - data type conversion in 3-16
 - decrement 4-5f
 - kinds of 4-3

- kinds of 4-3
- logical 4-7f
- order of evaluation of 4-2
- primary 4-3
- punctuation for 4-11
- relational 4-7f
- extern 7-18, 3-14, 3-19, 3-22
- data in recursive functions 6-9

F

- fabs() 10-74
- fchannel() 10-76
- fclose() 10-77
- fdopen() 10-79
- feof() 10-80
- ferror() 10-82
- fflush() 10-84
- fgetc() 10-86
- fgets() 10-88
- file-handling functions 10-2
- fileno() 10-90
- float 7-20, 3-7
 - comparison with other-language data types 6-18
 - rounding of 3-7
 - size limits of 3-7
 - truncating of fraction 3-17
 - zero-padding of fraction 7-20
- floating-point arithmetic in C 3-7
- floor() 10-92
- fmax() 10-94
- fmin() 10-96
- fmod() 10-98
- fopen() 10-100
- for 7-22, 5-7f
 - for loop, null statements with 5-11
- form feed 2-11
- FORTRAN 77
 - INTEGER data type 6-18
 - REAL data type 6-18
- fprintf() 10-104
- fputc() 10-107
- fputs() 10-109
- frame pointer in C calling sequence 6-15
- fread() 10-111
- free() 10-113
- freopen() 10-115
- frexp() 10-117
- fscanf() 10-119
- fseek() 10-122
- ftell() 10-125
- ftoa() 10-127
- functions in C 6-1ff
 - calling from other languages 6-10ff

- calling sequence among 6-14ff
- frame pointer in 6-15
- data type of 6-5f
- how to call 6-7
- how to create 6-3
- how to declare 6-4f
- main() 6-6
- passing data to and from 6-8
 - data type conversion by 3-18
- recursive 3-19, 6-9
- scope of 3-15
- storage class of 6-6f
- functions written in other languages, calling 6-10ff
- fundamentals of C language 1-2ff
- fwrite() 10-129

G

- getc() 10-131
- getchar() 10-133
- getdir() 10-135
- getenv() 10-136
- getfp() 10-137
- getlogin() 10-138
- gets() 10-139
- getsb() 10-141
- getsl() 10-142
- getsp() 10-143
- getw() 10-144
- goto 7-24, 5-10
- greater-than-or-equal-to operator (>=) 8-17
- greater-than operator (>) 8-16
- grouping with operators 4-2

H

- hardware representations C-1ff
 - character constants C-2
 - data sizes and machine dependencies C-1f
 - pointers C-2f
 - the struct type C-3
- hexadecimal constants 2-6
- hypot() 10-146

I

- I/O functions, using sizeof for 7-34
- identifiers, C 2-4
- identifiers, scope of 3-14
- if 7-25, 5-2ff
- implementation differences C-3ff
 - expressions C-5f
 - identifiers C-3

- initializations C-5
- keywords C-4
- line length C-3
- string constants C-5
- implementation characteristics C-1ff
- include files 9-5
- include files, standard 10-7ff
- increment operator (++) 8-18, 4-5f
- indirection operator (*) 8-19
- inequality operator (!=) 8-21
- infos() 10-147
- input/output functions 10-2f
- insert_after() 10-148
- insert_before() 10-149
- insert_tail() 10-151
- insert_head() 10-150
- int 7-27, 3-5f
 - comparison with other language data types 6-18
 - data type conversion of 3-16f
 - examples of 3-5
 - padding of leftmost bits 3-18
 - size limits of 3-5
 - truncation of 3-7
- inter-language calling with C 6-10ff
- initializations of C variables 3-22f
- intso() 10-152
- intss() 10-153
- isalnum() 10-154
- isalpha() 10-156
- isalphanum() 10-158
- isascii() 10-160
- isatty() 10-162
- iscntrl() 10-163
- isdigit() 10-165
- ishex() 10-167
- isline() 10-169
- islower() 10-171
- ispunct() 10-173
- isspace() 10-175
- isupper() 10-177
- isxdigit() 10-179
- itoa() 10-181

K

- keywords, C 7-1ff, 2-4
- list of 7-1

L

- L, to show long integer constant 2-8
- labels 5-10, 7-24

- scope of 3-15
- language type of an object 3-1
- lasterror() 10-183
- ldexp() 10-184
- less-than-or-equal-to operator (<=) 8-23
- less-than operator (<) 8-22
- library, C 2-4
- LINK switches 11-17
- linked lists 3-11, 3-12, 7-37
- linking C programs 11-19ff
 - CCL LINK macro 11-23f
 - combinable options 11-21ff
 - terminology 11-19f
 - use of the switches 11-20f
- listing control 9-7ff
 - cross referencing 9-8ff
 - line control 9-7
 - suspension of listing 9-7f
- log() 10-186
- log10() 10-188
- logical
 - AND operator (&&) 8-24
 - expressions 4-7f
 - negation operator (!) 8-25
 - NOT (!) 8-25
 - OR operator (||) 8-26
- long 7-29, 3-4
 - int, data type conversion of 3-17
 - comparison with other-language data types 6-18
- longjmp() 10-190
- lowercase characters in C 2-2
- lvalues 4-3

M

- macros 9-2ff
 - at compilation time 11-3f
 - invalidating macros 9-4
 - simple macros 9-2
 - with arguments 9-3f
- main() 10-192, 6-6
 - argc argument to 6-6
 - argv argument to 6-6
- malloc() 10-194
- max() 10-196
- memory management functions 10-3
- min() 10-198
- miscellaneous functions 10-3f
- mktemp() 10-200
- modf() 10-201
- module syntax B-6
- modulus operator (%) 8-27
- multiplication operator (*) 8-28

N

nargs() 10-203
NEW LINE 2-11
null statements 5-11
number sign (#) 2-2, 2-11
 as preprocessor delimiter 9-1ff

O

octal constants 2-5
ones complement operator (~) 8-29
operating system for C 1-2
operating procedures 11-1ff
operators 8-1ff, 4-1ff, 2-10
 compound assignment 8-8, 4-10
 conditional (?:) 4-10
 grouping with 4-2
 pointer 3-9
 precedence of 8-2ff, 4-1f
 table of 8-2ff
 structure member 4-4
order of evaluation, of expressions 4-7
 see also: the specific operator or type of expression
other-language functions, how to call 6-10ff
otoa() 10-205

P

packets 10-7f
padding of fractions 3-7
parentheses ((and)) 4-4f
Pascal
 calling between C and 6-10ff
 comparison of data types in C and 6-18
 DOUBLE_REAL data type 6-18
 INTEGER data type 6-18
 LONG_INTEGER data type 6-18
 pointer data type 6-18
 REAL data type 6-18
 RECORD data type 6-18, 3-3, 3-11, 6-13, 7-37
 SHORT_INTEGER data type 6-18
passing data to and from functions 6-8
pathname() 10-207
perror() 10-209
PL/I
 ALIGNED data type 6-18
 calling between C and 6-10ff
 CHARACTER VARYING declaration 6-13
 comparison of data types in C and 6-18
 FIXED_BINARY data type 6-18
 FLOAT_BINARY data type 6-18

- pointer data type 6-18
- STRUCTURE data type 6-18, 3-3, 3-11, 6-13, 7-37
- UNION data type 6-18
- pointer, structure 4-4f
- pointers 3-10
 - frame, in C calling sequence 6-15
 - in C calling sequence 6-14
 - in VS/ECS calling sequence 6-12f
 - operators 3-9
 - to arrays 3-9
- pow() 10-210
- precedence of operators 4-1f
- preprocessor delimiter (#) 2-11
- preprocessor syntax 9-1ff, B-6
- primary expressions 4-3
- printf() 6-7, 7-4, 10-212
- program, C, example of 1-7
- program environment functions 10-4
- programming style in C 1-7
- punctuation, for expressions 4-11
- putc() 10-218
- putchar() 6-7, 10-220
- puts() 10-222
- putw() 10-224

Q

- qsearch() 10-226
- queue-handling functions 10-5

R

- rand() 10-227
- randl() 10-229
- real numbers 2-8
 - see also float and double
- realloc() 10-231
- recursive functions 3-19, 6-9
- register 7-30, 3-19, 3-20, 3-22
- relational expressions 4-7f
- remove_element() 10-233
- remove_head() 10-234
- remove_tail() 10-235
- repeat() 10-236
- reserved words in C 2-4f
 - see also keywords
- return statement 7-31, 6-8f
- rew() 10-238
- runtime routines 10-1ff, 2-4

scale10() 10-240
 scanf() 10-242
 scope
 and storage class 3-19
 of formal parameters of a function 3-14
 of functions 3-15
 of labels 3-15
 of variables 3-14
 semicolon (;) 2-12
 separators, token 2-1f
 sequences, calling
 see calling sequences
 setbuf() 10-248
 setfp() 10-249
 setjmp() 10-250
 setsb() 10-251
 setscrn() 10-252
 setsl() 10-253
 setsp() 10-254
 setxy() 10-255
 shared page I/O C-12
 shift left operator (<<) 8-30
 shift right operator (>>) 8-30
 short 7-33, 3-4
 int, truncation of high-order bits 3-17
 unsigned, comparison with other-language data types 6-18
 side effects in C 4-8
 sign extension, for chars 3-18f
 signcmp() 10-257
 signncmp() 10-259
 sin() 10-261
 sinh() 10-263
 sizeof 7-34, 2-10
 not for use with labels, functions, undeclared structures 7-34
 sleep() 10-265
 space character 2-11
 sprintf() 10-266
 sqrt() 10-268
 srand() 10-270
 sscanf() 10-272
 standard \$builtin functions 10-6
 standard functions and macros, library 10-10ff
 standard include files 10-7ff
 statement syntax B-1f
 statements 5-1ff
 compound 5-11, 7-40
 conditional 5-2ff
 delimiters 2-12
 expression 5-2
 flow control 5-10
 loop 5-6ff
 null 5-11

- return 6-8f
- to get out of loops 5-9f
- static 7-35, 3-19, 3-20, 3-22
 - data in recursive functions 6-9
- stdio.h 7-4
- storage class 3-19ff
 - and scope 3-19
 - defaults for 3-23
 - for other-language functions 6-17
 - suggestions for 3-22
- strcat() 10-275
- strchr() 10-277
- strcmp() 10-279
- strcpy() 10-282
- string-handling functions 10-5f
- strlen() 10-284
- strncat() 10-286
- strncmp() 10-288
- strncpy() 10-291
- strnsave() 10-293
- strpbrk() 10-295
- strrchr() 10-296
- strsave() 10-298
- struct 7-37, 3-11, 8-33f
 - copying 3-12
 - functions returning 6-5
 - member operator (.) 8-33, 4-4f
 - passing to functions 3-12
 - pointer operator (->) 8-34
 - pointers 4-4f
 - self-referential 3-12
 - with sizeof operator 7-34
- subtraction operator (-) 8-35
- swab() 10-300
- switch 7-39f, 5-5f, 7-6
- synonyms for data types 7-41, 3-15
 - see also typedef
- syntax graph B-1ff
- syntax, expressions B-2f
- syntax, declarations B-4f
- syntax, modules B-6
- syntax, preprocessor B-6
- syntax, statements B-1f
- sys() 7-2, 10-302
- system() 10-304
- system, operating 1-2
- sys_~~system-call-mnemonic~~() 10-306

T

- tab 2-11
- tail() 10-314
- tan() 10-316

- tanh() 10-318
- testing assertions 9-11
- tmpfile() 10-320
- tmpnam() 10-321
- toascii() 10-322
- token separators 2-11f
 - form feed 2-11
 - NEW LINE 2-11
 - space 2-11
 - tab 2-11
 - vertical tab 2-11
- tokens, C 2-1ff
 - table of 2-3
- tolower() 10-323
- toupper() 10-327
- traceback() 10-331
- truncation of result
 - see division operator
- ttyname() 10-333
- type, language 3-1
- typedef 7-41, 3-3, 3-15, 3-19, 3-21
- types, data 3-1ff
 - see also data types

U

- unary minus operator (-) 8-37
- underscore character 2-1
- ungetc() 10-334
- union 7-43, 3-13
 - initializing 3-13
- unlink() 10-336
- unsigned 7-44, 3-4
 - char 7-7
 - int, data type conversion of 3-17
- uppercase characters in C 2-2
- usual arithmetic conversions, the 3-16f
- utoa() 10-337

V

- variables
 - initializations of 3-22f
 - scope of 3-14
 - with sizeof operator 7-34
- verify() 10-339
- vertical tab 2-11
- void 7-46
- VS/ECS 6-10ff

W

wdleng() 10-341
while 7-47, 5-6f
white space, routine for skipping 5-8

Z

zero() 10-342
zeropad() 10-343