

MPW IIgs ORCA/C™

*An MPW based C Compiler
for
Apple IIgs Cross Development*

Mike Westerfield

Byte Works®, Inc.
8000 Wagon Mound Dr. N.W.
Albuquerque, NM 87120
(505) 898-8183
MikeW50@AOL.COM

Limited Warranty - Subject to the below stated limitations, Byte Works, Inc. hereby warrants that this program will load and run on the standard manufacturer's configuration for the computer listed for a period of thirty (30) days from date of purchase. Except for such warranty, this product is supplied on an "as is" basis without warranty as to merchantability or its fitness for any particular purpose. The limits of warranty extend only to the original purchaser.

Neither Byte Works, Inc. nor the authors of this program are liable or responsible to the purchaser and/or user for loss or damage caused, or alleged to be caused, directly or indirectly by this software and its attendant documentation, including (but not limited to) interruption of service, loss of business, or anticipatory profits.

Apple is a registered trademark of Apple Computer, Inc.
MPW is a registered trademark of Apple Computer, Inc.
Byte Works is a registered trademark of Byte Works, Inc.
ORCA and ORCA/C are trademarks of Byte Works, Inc.

Copyright 1989, 1991, 1992, 1994, 1995
By The Byte Works, Inc.
All Rights Reserved

Master Set 1.0.0.0

Table of Contents

Chapter 1 – Overview of MPW IIGS ORCA/C	1
What's In This Package	1
About the Manual.....	1
Chapter 2 – Installation	3
Other Required Software	3
Installing MPW IIGS ORCA/C.....	3
Chapter 3 – The Command Line.....	5
Chapter 4 – Writing Assembly Language Subroutines	7
Introduction	7
The Basics.....	7
Returning Function Values From Assembly Language Subroutines.....	8
Passing Parameters to Assembly Language Subroutines	8
Accessing C Variables from Assembly Language.....	9
Calling C Procedures and Functions from Assembly Language.....	10
Chapter 5 – Program Symbols.....	11
Character Set.....	11
Identifiers.....	11
Reserved Words.....	12
Reserved Symbols	12
Continuation Lines	13
Constants.....	13
Decimal Integers.....	13
Octal Integers	14
Hexadecimal Integers	15
Character Constants.....	15
String Constants.....	16
Escape Sequences	17
Real Numbers.....	18
White Space.....	19
Comments.....	19
Chapter 6 – The Preprocessor	21
Syntax	21
Preprocessor Macros.....	21
Including Files.....	24
Precompiled Headers	25
Conditional Compilation.....	26
Line Numbers.....	29
Flagging Errors.....	29
Pragmas	30
cda.....	30
cdev	30
databank	31
debug	31
expand	32
float	33

Table of Contents

ignore	33
keep	34
lint	34
memorymodel	35
nba	36
nda	36
noroot	37
optimize	37
path	39
rtl	39
stacksize	40
toolparms	40
xcmd	41
Chapter 7 – Basic Data Types	43
Integers	43
Reals	43
Pointers	44
Chapter 8 – C Programs	45
The Anatomy of a C Program	45
The Function main	45
Argc and Argv	45
Separate Compilation	46
Interface Files	46
Chapter 9 – Declarations	49
Declarations	49
Storage Classes	49
Type Specifiers	51
Enumerations	53
Arrays	54
Pointers	55
Structures	56
Unions	59
Initializers	59
Constructing Complex Data Types	61
Scope and Visibility	62
Chapter 10 – Functions	65
Declaring a Function	65
Parameters	67
Traditional C Parameters	67
Function Prototypes	68
Variable Length Parameter Lists	69
Common Mistakes With Parameters	69
How Parameters are Passed	70
Returning Values from Functions	72
Pascal Functions	72
Chapter 11 – Expressions	73
Syntax	73
Operator Precedence	74
Terms	74

L-values	74
Constants	75
Simple Variables	75
Arrays	75
Function Calls	76
Component Selection	77
Parenthesized Expressions	78
Postincrement and Postdecrement	78
Math Operators	79
Addition	79
Subtraction	79
Multiplication	80
Division	80
Remainder	80
Unary Subtraction	81
Unary Addition	81
Prefix Increment	81
Prefix Decrement	81
Sizeof Operator	82
Comparison Operations	82
Logical Operations	83
Bit Manipulation	83
Assignment Operators	84
Simple Assignment	84
Compound Assignment	85
Multiple Assignments	86
Pointers and Addresses	86
Sequential Evaluation	86
Conditional Expressions	87
Automatic Type Conversions	87
Assignment Conversions	87
Function Argument Conversions	88
Unary Conversion Rules	88
Binary Conversion Rules	88
Type Casting	89
Converting Integers to Integers	89
Converting Floating-Point Values to Integers	89
Converting Pointers to Integers	89
Converting Floating-Point Values to Other Floating-Point Formats	90
Converting Integers to Floating-Point Values	90
Converting to and from Enumerations	90
Converting Pointers to Pointers	90
Converting Integers to Pointers	90
Converting Arrays to Pointers	91
Converting Functions to Pointers	91
Converting to Void	91
Chapter 12 – Statements	93
Compound Statements	93
Null Statements	94
Expression Statements	94
While Statement	94
Do Statement	95
For Statement	95

Table of Contents

If Statement	96
Goto Statement	96
Switch Statement	97
Break and Continue	98
Return Statement	98
Segment Statement	99
Asm Statement	100
Chapter 13 – Libraries	103
Overview of the Libraries	103
System Functions	103
Standard C Libraries	107
Appendix A – Error Messages	161
Compilation Errors	161
Terminal Compilation Errors	173
Appendix B – ANSI C	175
Appendix C – ORCA/C on the Apple IIGS	177
Index	179

Chapter 1 – Overview of MPW IIGS ORCA/C

This chapter describes what is in this manual and the contents of the MPW IIGS ORCA/C disk.

What's In This Package

This package includes a version of ORCA/C for the Apple IIGS which has been ported to run under MPW on a Macintosh computer. It includes the following items:

MPW IIGS ORCA/C Documentation	This documentation, which describes installing ORCA/C under MPW and gives a complete technical description of the compiler.
Tools:cciigs	The executable file for the compiler.
ORCACLibraries:ORCACLib	The run-time library file used with the compiler. As delivered, these libraries are appropriate for creating stand-alone programs for the Apple IIGS. With minor adjustments, they can be used on any 65816 platform.
ORCACDefs:≈	These are the header files for the Apple IIGS toolbox, ANSI C, and some common UNIX I/O libraries.
Source:ORCALib:≈	Source code and build files for the run-time libraries.
Source:cnv:≈	Binaries and source code for the utility used to preprocess the library files. This program is included for completeness, but is probably not needed. It was used as one step in moving the source code for the run-time libraries from ORCA/M format to MPW IIGS Assembly format. It is designed to insert comment characters and convert expanded macro invocations to comments.

About the Manual

Chapter 2 covers installing ORCA/C and describes other required software.

Chapter 3 describes the command line flags and options supported by MPW IIGS ORCA/C.

Chapter 4 describes the calling conventions used by ORCA/C. This information is useful for understanding the subroutine libraries and for writing assembly language functions that are called from C, access C variables, or call C functions.

The remaining chapters are a complete description of the C language as implemented in ORCA/C. With a few exceptions, ORCA/C is an implementation of ANSI C; however, even the ANSI C standard leaves some room for differences in compiler implementations. For this reason, the complete C language description is included to resolve any ambiguity regarding how a particular feature is implemented.

Appendix B summarizes the differences between ORCA/C and ANSI C.

Appendix C summarizes the differences between this implementation of ORCA/C and the native Apple IIGS implementation.

Chapter 2 – Installation

This chapter describes the supporting software needed to use ORCA/C, and how to install ORCA/C.

Other Required Software

MPW IIGS ORCA/C was designed for, and compiled under, the release version of MPW tools contained in E.T.O. 13. You should start with a complete MPW installation.

ORCA/C generates OMF files that are processed by the MPW IIGS Linker to create Apple IIGS executable files. Apple IIGS executable files can be converted to raw binary files using MakeBin IIGS . Both utilities, as well as several others that may prove useful, are in the APDA product *MPW IIGS Tools Version 1.3*.

The libraries are assembled with *MPW IIGS Assembler Version 1.2*. This assembler may also prove useful for developing large or complex subroutines.

Installing MPW IIGS ORCA/C

MPW should be installed before installing ORCA/C. After installing MPW, follow these steps:

1. Copy the ORCACLibraries and ORCACDefs folders from the MPW IIGS ORCAC 1 disk to the installed MPW folder.
2. Copy the file cciigs from MPW IIGS ORCAC 2:Tools:cciigs to the Tools folder of MPW.

▲ **Warning** Do not copy the entire Tools folder. Doing so will erase the original contents of the MPW Tools folder, and many of those tools are required for use of ORCA/C. ▲

3. Create a folder named Source in any convenient location. Copy the files from the Source folders on MPW IIGS ORCAC 2 MPW IIGS ORCAC 3 into this source folder, then copy the files from MPW IIGS ORCAC 3:Source:ORCALib into the ORCALib folder in the Source folder you created.

▲ **Warning** There is a folder called ORCALib in the source folder on both disks. Be sure to copy the files from the folder, rather than the folder itself, when you copy the files from MPW IIGS ORCAC 3. If you copy the folder, you will erase the files copied from MPW IIGS ORCAC 2. ▲

The various Make files will need to be updated based on the actual location of the Source folder.

Chapter 3 – The Command Line

The command line for MPW IIGS ORCA/C has the following form:

```
cciigs <flags> <source file>
```

The flags can be any of the following, in any order, but all options must appear before the source file name.

-b Turns on generation of inline debug code. This works like

```
#pragma debug -1
```

placed at the beginning of the source file.

-d <name> ['=' <token>] Defines a macro from the command line. If the equal character and token are omitted, this is equivalent to placing

```
#define <name> 1
```

at the start of the source file. If the token is included, this is equivalent to placing

```
#define <name> <token>
```

at the start of the source file. <token> can be an identifier, a numeric constant with an optional leading sign, or a string.

-i <path> Adds an include file to the search paths used to resolve #include directives. This works exactly like

```
#pragma path <path>
```

at the start of the source file.

-l The source file is echoed to standard out.

-o Turns optimization on. This is equivalent to

```
#pragma optimize -1
```

at the start of the source file.

-p Tells the compiler to print copyright, version and progress information to standard out.

-r Forces the compiler to rebuild the .sym symbol file, even if the modification dates of the source file and .sym file indicate it isn't necessary.

-s Tells the compiler to print symbol table and intermediate code to standard out. This flag is used in conjunction with internal

debug code, which but be enabled before compiling the compiler before this flag will have an affect.

-y Tells the compiler to ignore the .sym file. Any .sym file present is not used, and no new .sym file is generated.

-w Tells the compiler to pause after printing an error. Compilation continues when you press a key, and aborts if you press Command-.

Chapter 4 – Writing Assembly Language Subroutines

This chapter describes the subroutine calling convention used by ORCA/C. Using this information, you can write assembly language functions that can be called from ORCA/C, and you can call ORCA/C functions from assembly language.

Introduction

There are two ways to mix assembly language with ORCA/C. One way is to use the built-in mini-assembler that is a part of ORCA/C. The mini-assembler is suitable for tasks where a few lines of assembly are needed in a program that is mostly C. When several dozen lines or several subroutines must be coded in assembly language, you can also choose the MPW IIGS Assembler, which offers more power for larger assembly language tasks. This chapter deals with using the MPW IIGS Assembler to write entire functions or libraries in assembly language. Chapter 12 gives details on using the `asm` statement to embed assembly language code in a C program.

The Basics

Calling an assembly language subroutine from C is actually quite easy. For our first example, we will take the simplest case: a function returning void defined in assembly language that has no parameters and does not use any global variables from C.

We will define a small function to clear the keyboard strobe. This is one of those tasks that is difficult to do from C, yet takes only four lines of assembly language. You might want to call this function from a real program—the effect is to erase any character that the user has typed, but that has not yet been processed by a scan library call.

The C program must declare the function as `extern`. This is how you tell the compiler that the function appears outside of the C part of the program. A program that simply calls the subroutine would look like this:

```
extern void Clear(void);

int main(void)
{
    Clear();
}
```

The assembly language function is:

```

                case on
;
; Clear the keyboard strobe
;
                Export      Clear
Clear          proc
                sep        #$20
                sta        >$C010
                rep        #$20
                rtl
                endp
                end

```

An important point to remember is that C is case sensitive. Like most languages, assembly language is normally case insensitive. The case on directive at the start of the assembly language program makes the assembler case sensitive, like C. If you leave this directive out, you must always use uppercase characters when referring to the function from C.

Returning Function Values From Assembly Language Subroutines

Function values are returned in the registers. This means that within your assembly language subroutine you would load the registers with the values that you want to return to your C program. Char, int and unsigned int values are returned in the accumulator as two-byte quantities. Long integers and pointers are returned in the X and A registers, with the most significant word in X and the least significant word in A. Real numbers, both single and double precision, are returned as pointers to floating-point values which have been stored in SANE's extended format. This format is described in *Apple Numerics Manual*. Structures, unions and arrays are returned as a pointer to the first byte of the object. As with other types of pointers, the most significant word should be placed in X and the least significant word should be stored in A.

Please note that characters only require one byte of storage, but are returned in a two-byte register. Be sure to zero the most significant byte of the value that you return.

For a complete discussion of the internal formats of numbers, see Chapters 5 and 7. Basically, though, they correspond to what you are used to in assembly language.

Passing Parameters to Assembly Language Subroutines

ORCA/C places the parameters which appear in a subroutine call on the stack, in the opposite of the order that they appear in the parameter list. It then issues a JSL to your subroutine.

The value that is on the stack depends on the type of the value being passed. Int, unsigned int, long, unsigned long, characters, enumerations, structures, unions and pointers appear on the stack as actual values. The normal unary conversions are performed before the value is placed on the stack. Basically, that means that character values are expanded to two bytes before they are pushed, while two-byte and four-byte values are pushed on the stack as is. Structures and unions are placed on the stack in the same format that they are stored in memory, while floating-point values are pushed on the stack as extended format SANE numbers. Arrays and string parameters (which are actually just a special form of an array) are passed as an address that points to the first byte of the value.

Consider this C program fragment:


```

...

void doSomething(z, ch, i);

int i;
char ch;
float *z;

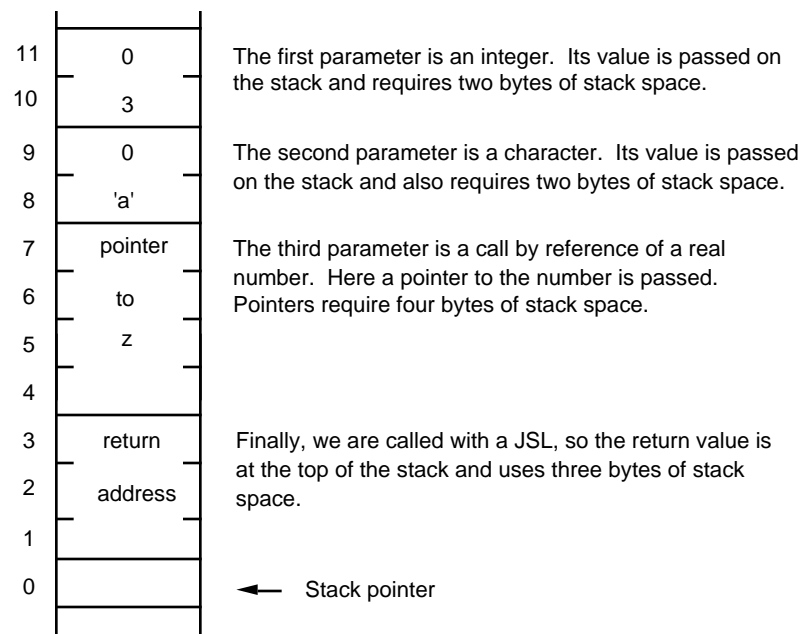
...

i = 3;
ch = 'a';
*z = 5.6;
doSomething (&z, ch, i);

...

```

When doSomething is called, the stack will look like this:



If changed, the direct page register and data bank register must be restored before returning to the calling function. The function is always called with long registers, and you must return control to the caller with long registers.

The assembly language function is responsible for removing parameters from the stack before returning to the calling function.

Accessing C Variables from Assembly Language

All global variables which can be accessed from outside of a C source file by another C function are available from assembly language. If you are using the small memory model, all C variables are accessed using absolute addressing. In the large memory model all variables must be accessed using long addressing.

It is also possible to define variables in assembly language that can be accessed from C. To do that, define the variable as external in the C program, just as you would if the variable was in a separately compiled C function. Then define the variable in assembly language. The assembly language variable must be defined globally; that is, it must be the name of a code segment, or it must be declared as global via the entry directive. If you are using the small memory model, the variable must be placed in the blank segment.

The C language uses case sensitive variable names. The assembler defaults to case insensitive variables, passing all variable names to the linker as uppercase strings. To make the assembler case sensitive, use the assembler's case directive.

Calling C Procedures and Functions from Assembly Language

Calling a C function from assembly language is extremely straightforward. You simply push any required parameters onto the stack and issue a JSL to the function you want to call. The one trick is that you must push the parameters on the stack starting with the last parameter, and working toward the first. Two-byte values are returned to you in the accumulator, four-byte values are returned with the least significant word in the accumulator and the most significant word in the X register, and real and double values are returned as pointers to ten-byte SANE extended format numbers. Note that real numbers should be passed as ten-byte SANE extended format numbers. Structures, arrays and unions are also returned as a pointer to the first byte of the object.

Chapter 5 – Program Symbols

C programs are made up of a series of program symbols called tokens. Tokens are the words used to write a program. They consist of identifiers, symbols, and constants.

Character Set

ORCA/C supports the extended Apple character set. All typable characters are accepted in the source stream, and all can be passed through the standard output channels.

Some of the special characters are also allowed as C source stream tokens. These international alphabetic characters are accepted wherever the 26 uppercase and lowercase ASCII letters would normally be allowed:

Ä	Å	Ç	É	Ñ	Ö	Ü	á	à	ä	ã	å
ç	é	ë	ê	ë	í	ì	î	ï	ñ	ó	ò
ô	ö	õ	ú	ù	û	ü	Æ	Ø	ª	º	æ
ø	À	Ã	Õ	Œ	œ	ÿ	ß				

These characters are treated as distinct alphabetic characters, just as uppercase and lowercase letters are distinct in C. For example, ü and u are not the same character, so `run` and `rün` are not the same identifier.

Six extended characters are also accepted as equivalents to C tokens:

character	C equivalent
÷	/
«	<<
»	>>
≠	!=
≤	<=
≥	>=

Identifiers

Identifiers in C start with an alphabetic character or underscore and are followed by zero or more alphabetic characters, numeric characters, or underscores. C is a case sensitive language, which means that the identifiers `matrix` and `Matrix` are actually two different identifiers.

ORCA/C imposes a limit of 255 characters on the length of any single identifier.

ORCA/C allows the use of characters from Apple's extended character set in identifiers. They are treated as new alphabetic characters, distinct from the original characters. For example, `rün` is a legal identifier, but it is different from the identifier `run`. The extended characters that are treated as alphabetic characters are:

Ä	Å	À	Ã	Ç	É	Ñ	Ö	Õ	Ø	Ü	Æ
Œ	á	à	ã	ç	é	ê	è	ê	ë	í	ì
î	ï	ñ	ó	ô	ö	õ	ø	ø	ú	ù	û
ü	ÿ	ß	ª	º	æ	œ					

Some examples of legal C identifiers are shown below. They each represent a different identifier.

```
main      array      myVar      myvar      _subroutine my_var
x1        x348
```

Reserved Words

Reserved words are identifiers that have special meaning in C. Unless a reserved word is redefined as a preprocessor macro, it can only be used for the meaning that C assigns to it, except that reserved words can appear in comments or string constants. The reserved words in C are shown below. The reserved words that are underlined are reserved in ORCA/C, but not in ANSI C.

auto	<u>asm</u>	break	case	char	<u>comp</u>	const
continue	default	do	double	else	enum	<u>extended</u>
extern	float	for	goto	if	<u>inline</u>	int
long	<u>pascal</u>	register	return	<u>segment</u>	short	signed
sizeof	static	struct	switch	typedef	union	unsigned
void	volatile	while				

Reserved Symbols

Reserved symbols are the punctuation of the C language. Reserved symbols are used as mathematical operators, for forming array subscripts and parameter lists, for separating statements, and so forth. With some restrictions, reserved symbols can also be used in comments, string constants, and characters constants. See the sections below for details.

The reserved symbols in C are:

!	%	^	&	*	-	+	=	~	
.	<	>	/	?	->	++	--	<<	>>
<=	>=	==	!=	&&		+=	-=	*=	/=
%=	<<=	>>=	&=	^=	=	()	[]
{	}	,	;	:					

In some older versions of C, the assignment operators (like +=) could be specified with the equal sign first, as in =+. This caused some semantic problems in the language; most modern compilers, including ORCA/C, do not permit the equal sign to come first. Some compilers also permit white space to appear between the characters, as in + =. ORCA/C does not allow this, either.

Six tokens can also be represented using a single character from the extended character set. They are:

character	C equivalent
÷	/
«	<<
»	>>
≠	!=
≤	<=
≥	>=

Some older computers do not support complete character sets on their terminals or printers. For that reason, the C language includes trigraphs. A trigraph is a sequence of three characters, two

question marks and a third character, which can replace another character. While there is no reason to use trigraphs on the Apple IIGS, you should be aware of their existence, since they can occasionally cause problems with string constants and character constants. The trigraphs, along with the character they represent, are shown below.

trigraph	character
??([
??)]
??<	{
??>	}
??/	\
??!	
??'	^
??-	~
??=	#

For example, the following two lines are completely equivalent in C.

```
while (~one | theOther) printf("Huh|\n");
while (??- one ??! theOther) printf("Huh??!\n");
```

Note the trigraph that appears in the string. This is one place where trigraphs can cause problems. The idea in the example shown was probably to write two question marks and an exclamation point, but this was translated into a single vertical bar. To avoid this, you can use \? in the string constant to represent one of the question marks, as in

```
while (~one | theOther) printf("Huh?\?!\\n");
```

Continuation Lines

If a back slash character (\) appears as the last character on a line, the line is continued from the first character of the next line. Lines may be continued in the preprocessor, in string constants, between any two tokens, or even in the middle of a token.

See the section discussing string constants for details on continuing strings. See Chapter 6 for details on using continuation lines in the preprocessor.

Constants

Constants are used to place numbers and strings into the source code of the program. Each kind of constant has its own unique format, so they are discussed separately.

Decimal Integers

Decimal integers come in two sizes and two kinds. The two sizes are referred to as integer and long integer, while the two kinds are signed and unsigned.

Signed integers consist of one to five digits. The number represented must range from 0 to 32767, and with the exception of the number 0, no number can start with a leading zero. (See octal integers, below, for the reason for this restriction.) You may use a leading - character to form a negative number, although the - character and the number are technically two separate tokens. In practice, this technical distinction is rarely important.

If the number exceeds 32767, or if it is followed by an l or L, the number becomes a long integer. Long integers can range from 0L to 2147483647L. Please note that the l or L character must follow immediately after the last digit, with no intervening white space characters.

Unsigned integers are integers which are followed by a u or U. Unsigned integers can range from 0U to 65535U. Unsigned integers have the same effect on the type of an expression as an unsigned variable; see Chapter 11 for details.

Unsigned long integers include any unsigned integer larger than 65535U, as well as any unsigned integer with an l or L appearing either before or after the u or U. Unsigned long integers can range from 0UL to 4294967295UL. As with unsigned integers, using an unsigned long integer constant will affect the type of the expression.

The table below shows some examples of legal decimal constants.

integer integer	unsigned integer		long integer	unsigned long
0	0U	0L	0LU	
35	35u	35l	35ul	
32767	65535u	100000	100000u	
600	600U	2147483647l	4294967295ul	

Octal Integers

Octal numbers are integers represented in base eight, rather than the more familiar base ten. Octal numbers are made up of the digits 0 to 7. In C, octal numbers are distinguished from decimal numbers by a leading zero. Any number whose first character is zero is interpreted as an octal number in C, and may not contain the digits 8 or 9. This can lead to unexpected results if you are not aware of this convention. For example, the statement

```
printf("%d\n", 010);
```

will print 8 to the screen, since 010 is 8 in base 8.

As with decimal integers, octal integers can be long or short, signed or unsigned. The range of integers allowed in each representation are show below.

int	0	077777
unsigned int	0	0177777u
long	0	017777777777L
unsigned long	0	037777777777uL

One point of confusion this often leads to is the base of the number 0. Technically, the number 0 is an octal number, but in practice it doesn't make any difference. You may use it when a decimal number is required because the difference in base only matters when the number is converted from the string you type in when you enter a program to the internal number used in calculations. In practice, then, you can use 0 as either a base 8, base 10 or base 16 number.

The table below shows some examples of legal octal constants, with their decimal equivalents.

octal integer	decimal integer
0	0
07	7
010	8
0100	64
077777	32767
0177777u	65535u
017777777777L	2147483647L
037777777777uL	4294967295uL

Hexadecimal Integers

Hexadecimal numbers are integers represented in base sixteen, rather than the more familiar base ten. Hexadecimal numbers are made up of the digits 0 to 9 and the letters a to f or A to F. In C, hexadecimal numbers are distinguished from decimal numbers and octal numbers by a leading zero, followed by an x character or X character.

As with decimal and octal integers, hexadecimal integers can be long or short, signed or unsigned. The range of integers allowed in each representation are show below.

int	0	0x7FFF
unsigned int	0	0xFFFFu
long	0	0x7FFFFFFFFL
unsigned long	0	0xFFFFFFFFuL

The table below shows some examples of legal hexadecimal constants, with their decimal equivalents.

hexadecimal integer	decimal integer
0x0	0
0X7	7
0x10	16
0x100	256
0xa	10
0xF	15
0x7FFF	32767
0xFFFFu	65535u
0x7FFFFFFFFL	2147483647L
0xFFFFFFFFuL	4294967295uL

Character Constants

Character constants are formed by enclosing one to four characters in quote marks, as in

```
'a'
```

Character constants containing a single character are treated exactly as if an integer constant equivalent to the ordinal value of the character was used instead of the character constant. For example, in any program that contains the character constant 'a', it would be legal, and have no effect on the executable program, to replace the character constant with 97.

That certainly doesn't mean there is no need for character constants. If you are trying to write a portable program that is comparing against the character a, you should use a character constant, since not all computers use the integer value 97 for a lowercase a.

Multi-character character constants, such as 'AB', are also accepted by ORCA/C. Multi-character constants are stored in memory in string order. Because of the way strings and numbers are stored on the 65816 microprocessor, this means that the character constant 'AB' is equivalent to the integer 0x4241, not 0x4142, as you might suspect. The integer equivalent of 'ABC' is 0x00434241, while the integer equivalent of 'ABCD' is 0x44434241.

Note ANSI C requires all character constants to be of type int, but ORCA/C allows 3 and 4 character constants, returning them as long int constants. Use #pragma ignore to restore strict conformance to the ANSI standard.

ORCA/C uses the ASCII character set to determine the ordinal values of the characters.

Some characters in the ASCII character set cannot be represented using a key that can be typed from the keyboard. These characters are represented as escape sequences, and are discussed later in this chapter.

The C language does not specify if integers are signed or unsigned; that detail is left up to the implementor of the compiler. In ORCA/C, integer constants are signed.

Examples:

```
'a'      'A'      '\''      '\''      '\040'      '\40'      '\x10'
```

String Constants

String constants consist of any sequence of characters except new line or the quote character, enclosed in quote characters. (Both the new line character and the quote character can, however, be represented as an escape sequence.) As with character constants, escape sequences are used to represent non-printing characters; they are described in the next section.

Internally, strings are represented as a sequence of bytes, one for each character in the string, followed by a terminating null byte. (A null byte has a value of zero.) The value of each byte is the ordinal value for the character, as specified by the ASCII character set and the Apple extended character set.

Unlike other constants, using a string constant in an expression does not cause the compiler to load the value of the string. Instead, as with arrays, the address of the first character is loaded. In practice, this means that you can use a string constant wherever a pointer to a string is required, as in the famous Hello, World program, shown below.

```
int main(void)
{
    printf("Hello, world.\n");
}
```

String constants can be spread across more than one line. There are two ways to do this. The first method has been a part of the C language for quite some time. It involves using the \ character as the last character on the line. In that case, the string constant continues with the first character of the next line. The second method is new to ANSI C. In the second method, you simply code two string constants with no intervening tokens, and the compiler plugs them together. The Hello, World program is shown below using these two methods.


```

int main(void)

{
printf("Hello, \
world.\n");
}

int main(void)

{
printf("Hello, "
      "world.\n");
}

```

Note that the second method allows you to put spaces or tabs in the program to improve readability, while the older method of using a continuation line requires that the continuation of the string start in column 1.

ORCA/C imposes two limits on strings. First, no single string constant may have more than 4000 characters. Second, no single function can have string constants whose total length exceeds 8000 characters.

Examples:

```

"Hello, world.\n"
"This is a string constant"      " that has been broken into two parts."
"He said, \"I have arrived.\" "
"What?\?! "

```

Escape Sequences

Escape sequences are used in string constants and character constants to represent characters that would otherwise be difficult to type from the keyboard, or that interfere with the construction of the constant itself. Escape sequences consist of the `\` character followed by a single character, an octal constant, or a hexadecimal constant. (In escape sequences, hexadecimal constants must begin with a lowercase `x`.) The table below shows the escape sequences that consist of a single character, along with the equivalent integral value and the standard use for the escape sequence in C.

escape sequence	integral value	meaning
<code>a</code>	7	alarm (bell)
<code>b</code>	8	back space
<code>f</code>	12	form feed
<code>n</code>	10	new line
<code>p</code>	(special)	p-string
<code>r</code>	13	carriage return
<code>t</code>	9	horizontal tab
<code>v</code>	11	vertical tab
<code>\</code>	92	<code>\</code> character
<code>'</code>	96	<code>'</code> character
<code>"</code>	34	<code>"</code> character
<code>?</code>	63	<code>?</code> character

Four of these escape sequences are used because of the syntax of character constants and string constants. Since the back slash character is used to start an escape sequence, you can also follow it with a second back slash character to place a single back slash character in a constant. The quote mark (") ends a string constant, and the single quote mark (') ends a character constant, so both have an escape sequence to allow you to put these characters in string and character constants. The ? character is used when a ? in a string or character constant might be confused with a trigraph. All of the remaining escape sequences except \p are used to represent control characters that have special meaning to the console driver. While the value used for each of these control characters will vary from computer to computer, they always perform the same action when the string or character constant is used with the standard output libraries.

The escape sequence \p is not used in standard C. It is used in ORCA/C to allow the formation of a p-string. Normally, strings end with a terminating null character. Unfortunately, many of the tool calls in the Apple IIGS toolbox require a p-string, which uses a length byte to indicate the length of a string. When you need to specify a p-string as a constant, use the escape sequence \p right after the opening quote mark. The final string will start with a byte that indicates how many characters are in the string, and will be followed by a normal C string. The terminating null character is not counted as one of the characters when determining the value of the length byte. The \p escape sequence is treated as the character p in a character constant, as well as in a string constant if the escape sequence is not the first character in the string.

When you need to specify a specific numeric value in a character or string constant, you can follow the back slash character by one to three digits, which are then interpreted as an octal number. For example, since ORCA/C uses the ASCII character set, which specifies an ordinal value of 041 octal (33 decimal) for the character !, the character constants '!' and '\41' represent the same value. You can also use a lowercase x followed by one, two or three digits to represent the value in hexadecimal notation. Thus, '\x21' is also a valid way to represent '!'. You should be careful when using values in strings. While '\41' represents the ! character in a character constant, '\410' is not the same as '!0', as you might intend. Instead, the finished string has a single character with an ordinal value of eight. (The most significant two bits do not fit into a byte, and are discarded.)

The C language does not specify what the compiler will do if you follow the \ character by a character other than one of the ones discussed. In ORCA/C, use of a non-escape character after the \ character will place that character in the constant. For example, the character constants '\g' and 'g' are exactly the same.

For examples of strings and character constants that include escape sequences, see the two preceding sections.

Real Numbers

Floating-point constants are used to represent numbers that do not have integral value, or that cannot be represented using an integer because they are too large or too small. The general format is a sequence of digits, followed by a decimal point, followed by another sequence of digits, and an exponent, as in

```
3.14159e-14
```

The exponent can start with either an uppercase E, or a lowercase e, as shown.

The format for floating-point constants can vary quite a bit from this general form. You can leave out the digit sequence before or after the decimal point, as in 1.e10 or .1e10. In fact, you can leave off the exponent, too, as in 1. or .1. You must have either an exponent or a decimal point, but if you specify an exponent, you can omit the decimal point, as in 12e40.

The exponent, if specified, starts with either a lowercase or uppercase e. This is followed by an optional plus or minus sign, and then by a sequence of one or more digits. The number is

represented internally as an extended SANE format number, giving an accuracy of a little over 19 decimal digits, and an exponent range of -4932 to 4932. Keep in mind that if the number is stored in a float or double variable, the accuracy will be reduced to match the accuracy of the variable.

C allows a floating point constant to be followed by `f` or `F` to indicate a float value, or `l` or `L` to indicate long double. While these characters are allowed, they have no effect on a floating-point constant in ORCA/C.

White Space

White space characters may be used to separate the tokens in a program. White space characters include the space, line feed, carriage return, vertical tab, horizontal tab, back space, and form feed characters. The line feed, carriage return, form feed and vertical tab characters all end the current line, which has special meaning in the preprocessor and when continuing lines. With that exception, replacing any sequence of white space characters outside a string or character constant with a single space has no effect on the finished program. Generally, white space characters are used to format the program, making it easier to read.

Comments

A comment starts with the characters `/*`, and ends with the first occurrence of the characters `*/`. Comments are used solely to document the source code; replacing a comment with a single space will have no effect on the finished program.

Note that comments can extend over more than one line, and that they can be used in preprocessor commands.

C does not allow recursive comments (sometimes called nested comments); that is,

```
/*    /*    */    */
```

is not a legal comment in C. You can use the preprocessor's `#if`, `#endif` commands to cause the compiler to ignore large sections of code which may include comments.

Comments can also start with the characters `//`. These comments extent to the end of the current line.

△ **Important** `//` comments are an extension to ANSI C. There are a few pathological statements that are technically legal in ANSI C that will not compile properly in ORCA/C because of its support for `//` comments. If you are compiling a program that should not use `//` comments, see `#pragma ignore` for a way to disable them. △

Chapter 6 – The Preprocessor

The C preprocessor is a series of commands that tell the compiler to take certain actions. Using the preprocessor, you can tell the compiler to skip certain sections of source code, to replace some source code by other source code, and so forth. While the preprocessor is built right into ORCA/C, logically, preprocessing occurs before the program is compiled.

Syntax

Commands to the preprocessor all start with the # character, which must appear before any other non-white space character in the line. This is followed by the preprocessor command, which may require other parameters. You can separate the preprocessor command from the # character with white space characters. The preprocessor command ends with the first new line, vertical tab, carriage return, or form feed character. The # character can appear on a line by itself, in which case it is ignored.

The ability to place white space characters before the # character and between the # character and the name of the preprocessor command is a recent addition to the C language. If you are concerned with portability, you may wish to code the preprocessor commands as we do in the examples, without white space characters.

There are two ways to extend a preprocessor command over more than one line in the source file. The first is to continue the line by placing a \ character at the end of the line, as in

```
#define sec(x) \
    (1.0 / cos(x))
```

The other way to extend a preprocessor command over more than one line is to start a comment on one line, and finish it on another line, as in

```
#define          /* trig extensions
*/    sec(x)    (1.0 / cos(x))
```

While the preprocessor does not parse the source file, it does break the source file up into tokens to find macro names. The same rules are shared by the preprocessor and the compiler for forming tokens and comments.

Preprocessor Macros

The #define preprocessor command allows you to define macros. When the macro is used later in the source code of the program, it is replaced by the tokens laid out in the #define command. One of the most common uses of preprocessor macros is to define constants using simple textual replacement. For example, you can define the boolean constants true and false, as shown in the simple program below.

```

#define TRUE -1
#define FALSE 0

int main(void)
{
    int bool;

    bool = TRUE;
    if (bool)
        printf("Hello, world.\n");
}

```

While the compiler uses a more efficient mechanism to implement preprocessor macros, the way to think of this program is that the two `#define` statements define two macros, called `TRUE` and `FALSE`. Whenever the preprocessor finds one of those words in the program, it is replaced by the text that follows the words. In our example, then, preprocessing occurs before the compiler starts to compile the program, so the program the compiler actually compiles looks like this:

```

int main(void)
{
    int bool;

    bool = -1;
    if (bool)
        printf("Hello, world.\n");
}

```

Before moving on to more complicated examples, a few points are worth mentioning. First, the text that will replace the name of the macro starts with the first non-white space character after the name of the macro itself. This text is converted into a stream of tokens by the preprocessor, and stored as tokens. This has an important consequence. At first glance, it might seem that a statement like

```
bool = 0FALSE;
```

would be legal, since, after macro replacement, the statement looks like this:

```
bool = 00;
```

In fact, the statement is not legal, because the macro has already been converted into an integer with a value of zero, so the compiler sees two integer constants after the equal sign. The correct equivalent, then, is

```
bool = 0 0;
```

One thing you will notice in all of the examples is that all of the macro names are capitalized. You are not required to capitalize the names of macros, but this is a common convention used by many C programmers, and we will follow that convention in our examples. It is worth pointing out, though, that the names of macros, like all identifiers in C, are case sensitive.

Macros are not limited to simple textual replacement. For example, you can define a trigonometric function for the secant using a macro.

```
#define sec(x) (1/cos(x))
```

Note that in this case, since we want the macro to blend in with the standard C math library, we have used lowercase letters in the name of the macro.

When you use a macro that has parameters in a program, you can substitute any number of tokens for the parameter. For example, all of the following are legal uses of the macro preprocessor, although not all of the examples result in legal C programs.

```
sec(pi)
sec(0.45)
sec(pi/12.0)
sec("strings are allowed, although this example will not compile")
sec((pi/12.0+.45)*0.01)
```

Macros can have any number of parameters. To create a macro with more than one parameter, list the names of the parameters separated by commas. For example, you could define a macro to find the distance between two points, as in the following small program, which also shows one macro calling another.

```
#include <math.h>
#include <stdio.h>

#define sqr(x) ((x)*(x))
#define distance(p1,p2) (sqrt(sqr(p1.x-p2.x)+sqr(p1.y-p2.y)))

int main(void)
{
    struct point {float x,y;} point1,point2;

    point1.x = 1.0;
    point1.y = 1.0;
    point2.x = 3.0;
    point2.y = 1.0;

    printf("The distance is %f\n", distance(point1,point2));
}
```

There are a few important points to note about the syntax of macros that have parameters. First, the opening parenthesis must appear immediately after the name of the macro in both the definition and the use. This is to prevent confusion between a macro parameter and a simple textual replacement macro whose first character is a left parenthesis. Second, you can use parentheses within the macro call to enclose token streams that include commas, so long as the parentheses are legal in the C program produced by the macro expansion. The names of macro parameters, like macros, follow the same rules as identifiers in C, with the exception that reserved words are not reserved in the preprocessor. In fact, it is legal, although very poor form, to define a reserved word as a macro.

It is not legal to define more than one macro using the same name.

Within a macro, it is possible to merge the string value of a macro parameter with adjacent string constant, or to treat the parameter as a string constant. The # operator, when it appears before the name of a macro parameter, indicates that the parameter is to be treated as a string rather than a token. For example, the following code fragment will print the familiar Hello, world. string to the screen.

```
#define greet(who) "Hello, " #who "."

printf(greet(world));
```

The macro preprocessor can also merge two tokens to form a new token. The `##` operator controls this process. The following code fragment is equivalent to `strPtr = setPtr`.

```
#define ptr(where) where ## Ptr

ptr(str) = ptr(set)
```

Macros can also be removed with `#undef`. `#undef` takes a single parameter, which is the name of a macro to undefine. The macro by the given name is removed from the preprocessor's macro symbol table. It is not an error to undefine a macro that has never been defined. Once the macro has been undefined, it is also not an error to redefine it.

```
#undef greet
```

There are several predefined macros that exist in any C program. These macros can be used in your program, but you cannot change them or remove them via the `#undef` command. Each of the predefined macros places a single token in your program; the token, and what it means, is shown in the table below.

macro	token	use
<code>__DATE__</code>	string	The date as of the start of the compilation, in the form Mmm dd yyyy, e.g., "Jan 12 1989".
<code>__FILE__</code>	string	Name of the current source file.
<code>__LINE__</code>	integer constant	Line number of the line being compiled. Each physical line in the file is counted, even if the line is in a comment or is skipped because of preprocessor commands.
<code>__TIME__</code>	string	Time as of the start of the compilation, in the form hh:mm:ss, e.g., "15:36:12".
<code>__STDC__</code>	integer constant	A non-zero value in any compiler that implements ANSI C. In ORCA/C, the value is -1. In a compiler that does not implement ANSI C, this macro will not be defined.
<code>__ORCAC__</code>	integer constant	ORCA/C will return a -1. In any other C compiler, this macro will not be defined.
<code>__VERSION__</code>	string	The compiler version in the form "2.0.0".

Including Files

The `#include` command is used to deal with situations where a source program is made up of more than one source file. The most common use of the `#include` directive is to include interface files for the standard C libraries. For example, to be completely correct, the common Hello, World program should actually start off with a `#include` command that includes the header for the standard input and output library, like this:

```
#include <stdio.h>

int main(void)
{
    printf("Hello, world.\n");
}
```

The include statement is followed by a file name, which can be enclosed in quote marks or brackets. The compiler compiles all of the lines in the included file, then returns to the file that

contained the `#include` command, and continues compiling with the line immediately after the `#include` command. In the example, the name of the header file for the standard input and output libraries is `stdio.h`. This file can be found with all of the other standard header files, which include not only the standard C libraries, but also the Apple IIGS toolbox interface. These header files are located in the `ORCACDefs` folder. Any time you enclose a file name in brackets, the compiler will look for the file in the `ORCACDefs` folder.

The other common use for the `#include` command is to include source code you have written specifically for a program. This could include custom header files, macro definitions used in more than one source file, or even a single program which has grown too large to edit. In this case, you would enclose the file name in quote marks, as in

```
#include "mymacro"
```

When the file name is enclosed in quote marks, the compiler looks for the file in the current directory. If the file you want to include is located in a folder in your current directory, you can use a partial path name; you can also use a full path name if the file you want to include is located on another disk, or in some other location on your current disk. Examples are shown below.

```
#include "/network/project.x/secret.macros"
#include "macrofolder/macro.file.1"
```

It is possible to use a macro for the file name, provided the file name can be broken down into tokens. In the case of a quoted file name, this will always work, since the file name is a string constant. In the case of a file name enclosed in brackets, you will be able to use any file name, but some file names from other file systems may not work. Note that the ability to use a macro for the file name is a recent addition to the C language, and may not be present in all C compilers.

There is no fixed limit to the nesting level for included files. For example, an included file can include another file, which can also include still another file, and so on. This process can continue for as long as memory is available.

ORCA/C supports another file inclusion mechanism, the `#append` command. This command is used to chain two files together. Like the `#include` command, the `#append` command requires a file name as a parameter, and this file name can be enclosed in brackets or quote marks. Brackets are still used to indicate that the file should come from the `ORCACDefs` folder, and quote marks are used for files in the local directory, or for full path names. You can also use a macro to specify the path name, with the same restrictions that applied to the `#include` command.

There are two principle differences between the `#include` command and the `#append` command. The first is that all lines after the `#append` directive are ignored. Once the compiler starts processing lines in the appended file, it never returns to the original file, as the `#include` directive does. Conceptually, the `#append` command is like a `goto`, while the `#include` command is like a subroutine call. The other difference between the two commands is that, if the `#append` command appears in the top level file, the file that is appended can be a source file for some language other than C. The top level file is the source file you actually compile, that is, a file that has not been included using the `#include` command. This powerful feature means that you can create a program using two or more languages, and then compile the entire program with a single step.

Precompiled Headers

Many C programs, especially those that use the Apple IIGS toolbox, start with a list of header files to include. In practice, compiling the header files can actually take longer than compiling the executable part of the program.

ORCA/C tries to cut compile times by eliminating the need to compile the headers files over and over. As each header file is compiled, ORCA/C writes the new symbol table information to a file

we will call the .sym file. This file is in the same folder as the initial source file. The name of the file is formed by removing the last extension on the source file name, if any, and adding the characters ".sym". For example, the .sym file for a source file named foo.cc would be foo.sym. The next time the file is compiled, ORCA/C can read the symbol table from the .sym file, often cutting compile times more than in half.

There are many ways to implement this concept. Two design considerations drove our choice for the implementation method. First, we wanted a mechanism that was 100% transparent to anyone using the compiler. With the exception that you will see .sym files formed for each compile, ORCA/C precompiled headers are completely automatic. The other factor that drove the design is that macro definitions that precede include statements can, and often do, effect the way a header file is compiled. For example, it is very common to use a macro to override the size of an array in a tool header file. This useful feature of the C language makes it impossible to compile the header file itself, replacing it with a symbol table, without requiring programmers to know when forcing a recompile of the symbol tables is appropriate.

ORCA/C does several things to determine when a .sym file must be rebuilt. First, if the source file changes in any way before the include statements, the symbol file is rebuilt. If the time or date stamp on any include file changes, the symbol file will also be rebuilt. Finally, if an include file is missing or cannot be accessed, the .sym file is rebuilt.

There are also some restrictions that apply to which header files are included in the .sym file. The first code-generating function or initialized variable in the program completes the .sym file, and all subsequent header files are compiled in the normal way. For example, if an include appears after a function definition (*not* declaration) it will not be included in the .sym file. In addition, any header file that contains an initialized variable or function definition will not be included in the .sym file, and will block subsequent header files, too.

Assuming the .sym file must be rebuilt, it actually takes two compiles before the process is complete. On the compile where the compiler determines that the .sym file should be rebuilt, it uses any information from the .sym file up to the point of change, then deletes the old .sym file. The new .sym file is built on the next compile.

There are two flags that control the use of .sym files. On any of the compile commands, the `i` flag tells the compiler to compile the program as if the compiler did not support precompiled headers. Any .sym file that is present is ignored, and a new .sym file will not be built. The `-r` flag forces the compiler to rebuild the .sym file, even if the compiler does not see any reason to do so.

Precompiled headers are not supported by the small memory version of the compiler.

Conditional Compilation

The preprocessor includes a powerful conditional compilation mechanism. Using conditional compilation, it is possible to control the way a program is compiled by making small changes in the source for a program.

The two commands which form the cornerstone for conditional compilation are `#if` and `#endif`. The `#if` command has an expression for an operand. This expression is evaluated. If the result of the expression is zero, all of the lines from the `#if` command to the matching `#endif` command are skipped by the compiler. If the result of the expression is not zero, the lines following the `#if` command are compiled in the normal way, and the matching `#endif` command is ignored.

As an example, we could write a very short program to test to see if a compiler implements ANSI C. All ANSI C compilers define the `__STDC__` macros as a non-zero value. In compilers that do not implement ANSI C, the macro will not be defined. When an undefined macro name is used in an expression in an `#if` command, it is replaced by the integer constant 0. Our program, then, looks like this:

```

int main(void)

{
#ifdef __STDC__
printf("This compiler implements ANSI C.\n");
#endif
}

```

While the lines between the `#if` and `#endif` statements are skipped if the expression in the `#if` statement evaluates to 0, the lines skipped must still be made up of legal C tokens. For example,

```

#ifdef 0
@
#endif

```

is not legal, since the character `@` is not a legal C token.

`#if` statements may be nested. For example, we could expand our previous example to see if the compiler being used is ORCA/C, like this:

```

int main(void)

{
#ifdef __STDC__
printf("This compiler implements ANSI C.\n");
#ifdef __ORCAC__
printf("In fact, this is ORCA/C.\n");
#endif
#endif
}

```

With a few exceptions, the rules for forming an expression in an `#if` statement are the same as the rules for expressions in the C language. The exceptions have to do with the fact that the expression is evaluated at compile time, rather than at run time, and also with features have been added to simplify tasks in the preprocessor. The major difference between expressions used in the `#if` command and expressions used in the program is that the expression that appears after the `#if` command must be a constant expression. For a detailed discussion of constant expressions, see Chapter 11.

Preprocessor expressions can make use of two features which cannot be used in any other expression. The first has already been mentioned: when an undefined macro is encountered, it is given a value of 0. The second feature is the defined operator. The defined operator is followed by the name of a macro, which may be enclosed in parentheses. The result of the operator is 1 if the macro is defined, and 0 if it is not defined. Rewritten to use the defined operator, our example program that determines if a compiler is an ANSI C compiler looks like this:

```

int main(void)

{
#ifdef defined __STDC__
printf("This compiler implements ANSI C.\n");
#endif
}

```

As with the C language, the preprocessor's `#if` statement can have an `else` clause. Any `#if` statement can have a `#else` statement between the `#if` statement and the matching `#endif` statement. We can use this feature to extend our test program so that it prints a message if the compiler is not an ANSI C compiler, too.

```

int main(void)

{
    #if __STDC__
    printf("This compiler implements ANSI C.\n");
    #else
    printf("This compiler does not implement ANSI C.\n");
    #endif
}

```

Complex conditions can be handled using the `#elif` command. The `#elif` command is a combination of the `#if` command and the `#else` command. Like the `#else` command, the `#elif` command is used after a `#if` command, but before the matching `#endif` command. It must also come before the `#else` command if there is one. Like the `#if` command, the `#elif` command is followed by an expression. This expression is only evaluated if the previous `#if` command or `#elif` command evaluated to zero. In that case, the expression is evaluated. If the result is zero, the preprocessor scans forward to the next matching `#elif`, `#else` or `#endif` command, and takes appropriate action. If the result of the expression is not zero, all statements up to the next matching `#elif` command, `#else` command, or `#endif` command are compiled. Any lines up to the matching `#endif` statement and the `#endif` statement itself are then skipped by the compiler. Note that more than one `#elif` command can appear in a single `#if` structure, in which case the preprocessor evaluates the expression in each statement in turn until one of the expressions results in a non-zero value. The lines following that statement are then compiled. If all of the `#if` and `#elif` statements have expressions that evaluates to zero, the lines following the `#else` statement are compiled.

As an example, we can expand the program that determines what compiler we are using to detect ORCA/C or APW C.

```

int main(void)

{
    #if __ORCAC__
    printf("This compiler is ORCA/C, which implements ANSI C.\n");
    #elif __STDC__
    printf("This compiler implements ANSI C.\n");
    #elif __APWC__
    printf("This compiler is APW C, which does not implement ANSI C.\n");
    #else
    printf("This compiler does not implement ANSI C.\n");
    #endif
}

```

There are two special forms of the `#if` statement which are used to test for the existence of a macro. They are `#ifdef` and `#ifndef`. Both statements require an operand that is a single identifier. If the identifier has been defined as a preprocessor macro, then `#ifdef` works exactly like `#if 1`, and `#ifndef` works like `#if 0`. If the identifier is not a preprocessor macro, the results are switched. Recoding our program to determine if a compiler is an ANSI C compiler to use these commands, we have the following:

```

int main(void)

{
#ifdef __STDC__
printf("This compiler implements ANSI C.\n");
#endif
#ifndef __STDC__
printf("This compiler does not implement ANSI C.\n");
#endif
}

```

Line Numbers

Historically, the C language has been associated with preprocessors. The preprocessor described in this chapter was originally a separate program from the C compiler. It took a source file as input, and produced another source file as output; the output file was then compiled. The C++ language is a more recent example of a preprocessor used with the C language. There are also numerous examples of cross-compilers that work by taking a program written in one language, say FORTRAN, and producing a program written in C for compilation by the C compiler.

This presents a problem when the compiler needs to flag an error. Meaningful error messages are associated with information about the source file in which the error occurs, and the line where the error occurs. It is very discouraging, for example, when a compiler is only able to tell you that a divide by zero error occurred somewhere in your 10,000 line program; it is far more useful if the compiler identifies which line of which source file the error occurred in.

The `#line` command is used by preprocessors for this purpose. It takes one or two parameters, either of which may be produced by expanding a macro. The first, which is required, is an integer constant. The value is used by the compiler as the line number of the next line that is compiled. The second parameter, which is optional, is the name of the source file, enclosed in quote marks. This name is used until it is overridden by another `#line` command. The line number and file name specified this way are used by the compiler for compile-time errors, reported by the program when a run-time error occurs and the `#pragma debug` directive has been set to an appropriate value, and used by the source-level debugger to show where you are in the original source file.

Flagging Errors

The `#error` statement is used to produce a compile-time error message. It uses a single parameter, which must be a string constant. This string is printed as an error message at compile time. For example, the program

```

int main(void)

{
#ifdef __GARBONZOC__
#error "This program requires Garbonzo C."
#endif
}

```

would generate the compile-time error message

```
#error: This program requires Garbonzo C.
```

just before the `#error` command in the output listing.

Pragmas

The `#pragma` command is used for preprocessor commands that are specific to a particular compiler. Each `#pragma` command that ORCA/C recognizes is followed by an identifier that specifies the type of the statement. Any `#pragma` statement that does not start with one of the identifiers described below is skipped.

cda

`#pragma cda name start shutdown`

The `cda` command is used to create a classic desk accessory. Classic desk accessories are not executed like normal C programs; instead, they are copied into the `DESK.ACCS` folder in the `SYSTEM` folder, where they can be used from any program that follows standard rules for the Apple IIGS.

Normal C programs must contain a function called `main`. This function is the one executed when the program starts. Classic desk accessories do not have to have a function called `main`. Instead, you specify the name of the function that will be called when the classic desk accessory is executed as the `start` parameter to the `cda` command.

When the operating system is shutting down, it will call each of the classic desk accessories to give it a chance to do any shut-down processing that it may require. Each classic desk accessory must have a shut-down function. The `shutdown` parameter is the name of your shut-down function. If you do not need to do any shut-down processing, you must still have a shut-down function, but it can be an empty function.

Each classic desk accessory has a name. This name appears in the CDA menu, which is used to select the classic desk accessory to execute. The `name` parameter is used to specify the name of your classic desk accessory. It must be a string constant.

The `cda` command must be used before the start of the first function in the program. If it is used, you cannot use another `cda` command or an `nda` command.

cdev

`#pragma cdev start`

The `cdev` command is used to create a Control Panel Device (CDev) for the Control Panel from System 6.0. Classic desk accessories are not executed like normal C programs; instead, they are copied into the `CDEV` folder in the `SYSTEM` folder, where they can be used from within Apple's Control Panel.

Normal C programs must contain a function called `main`. This function is the one executed when the program starts. CDevs do not have to have a function called `main`. Instead, you specify the name of the function that will be called when the classic desk accessory is executed as the `start` parameter to the `cdev` command. The function itself takes two long integer parameters and an integer parameter, returning a long integer.

For a complete description of Control Panel Devices, as well as details about the parameters passed to this function and the value returned by the function, see Apple II File Type Notes for file type \$C7 (CDV).

databank

#pragma databank *parm*

There are several instances that arise when using the toolbox where you need to create a function that a tool will call. When the C compiler creates the code for a function, it assumes that the data bank register is pointing to the bank that contains the global scalars. When a function is called by a tool, there is no guarantee that this is true. To solve this problem, you must use the databank directive before any function that will be called from a tool. This directive tells the compiler to save the original data bank register, and then set the data bank register to point to the bank containing the global scalars. Before returning from the subroutine, the original data bank register is restored.

```
#pragma databank 1
```

Immediately after the function, switch back to the normal calling conventions using the command

```
#pragma databank 0
```

Functions that have data bank restoration on can still be called from other C functions, but they will be slightly less efficient than functions that do not have data bank restoration enabled.

See also the toolparms directive.

debug

#pragma debug *parm*

The debug command is used to control the types of debugging the compiler will do. The various debugging features can be very useful when you are developing a program, but each debugger feature requires run-time code which takes up space and time in your finished program.

This command is only used in rare cases. If you are using the desktop development environment, you will normally use the debug check box in the compile dialog to control debugging. When that box is checked, all of the debugging features described here are enabled. When that box is not checked, all of the debugging features are disabled.

The debug command requires a single integer parameter. This parameter is actually a series of flags. For each flag, a value of 1 turns the debugging feature on, and a value of 0 turns it off. Even if new debugger features are added in the future, then, the command

```
#pragma debug -1
```

will enable all debugging features, while the command

```
#pragma debug 0
```

will turn all debugger features off.

Setting bit 0 (a value of 1) turns range checking on. The only checking this enables in the C compiler is a check for stack overflows. This check is made at the start of each function. The check ensures that there is enough room left on the run-time stack to declare all of the local variables needed by the function. This reduces the chance of crashes due to stack overflows. It is still possible to overflow the stack with a very complicated expression or by calling a tool.

Setting bit 1 (a value of 2) tells the compiler to generate debug code for the source-level debugger. You should not set this bit unless the program will be executed from the desktop

development environment. If you try to execute a program that has source-level debug code in it from the text shell or the Finder, the program will crash.

Setting bit 2 (a value of 4) tells the compiler to generate profile code. Profile code is used by the desktop development environment's profile command to tell you where the hot spots are in your program. You must set bit 1 if you set bit 2.

Setting bit 3 (a value of 8) tells the compiler to generate trace back code. Without trace back code, a run-time error will simply report which error occurred. If you have enabled trace back code, you will get more information about the error. This information starts with the name of the function where the error occurred, along with the line number in the source file. This is followed by a table showing the function that called the one where the error occurred, and the line number the call was made from, then the function that called the previous one, and so forth, back to the function main. This information is written to error out, which defaults to the text screen in the text environment, and the shell window in the desktop environment.

Setting bit 4 (a value of 16) tells the compiler to check for stack errors. If this check is enabled, the compiler generates code for each function call that keeps track of the stack as the function is called, making sure the function removes exactly the right number of bytes from the stack. For example, if you call a function and pass an integer parameter, two bytes are pushed onto the stack. If the function expected a long integer, though, it will remove four bytes from the stack, and this error check would catch the error. When using this error flag, we suggest

```
#pragma debug 25
```

which also turns on stack overflow checking and tracebacks. With tracebacks enabled, the compiler doesn't just tell you that an error occurred, it also tells you where the error occurred, making the error a lot easier to track down and correct.

expand

#pragma expand int

The expand pragma allows you to see the tokens the compiler is actually compiling, essentially showing you the output from the preprocessor, with all of the preprocessor macros expanded. If INT is a non-zero value, the preprocessor prints the token stream sent to the compiler, after all tokens have been expanded. If INT is zero (the default), this information is not printed. This feature is useful for debugging macros and examining the effects of character constants and escape codes. The exact expansions produced are:

1. All integers and character constants are expanded to base 10 values.
2. Escape sequences in strings are printed as hex escape sequences.
3. Macros are expanded.
4. Floating point constants are converted to exponential form.
5. Trigraphs are converted to their equivalent characters.
6. Preprocessor directives are removed from the source stream.
7. Any input skipped due to conditional compilation directives is removed from the source stream.
8. Comments are removed.

float

#pragma float card slot

By default, ORCA/C generates calls to the Standard Apple Numerics Environment (SANE) to perform floating-point calculations. If you have installed an Innovative Systems FPE card, however, ORCA/C can create a program that calls the card directly. To create a program that calls the FPE card, place the float directive before the first function in your program. The card parameter should be coded as 1. The slot parameter is no longer used, although a value must be coded. If your program does not do any floating-point calculations, this directive will make no difference in the code or execution time. On some floating-point intense programs, however, the FPE card can speed up a program by a factor of 120.

The actual effect of this directive is to tell the compiler not to generate direct calls to SANE, forcing it to use library calls for all floating-point calculations.

In addition to using this directive, you should replace the SysFloat library in the Libraries folder with the library by the same name at the path :MoreExtras:FPE:SysFloat. This library generates calls to the FPE card, rather than calling SANE. Because of this arrangement, you will get some benefit from the FPE card by simply replacing the library, even if you forget to use this pragma.

If a program is compiled with the FPE libraries, an FPE card must be installed or the program will crash or give incorrect answers. If you must create a program that can work with or without an FPE card, write the program to make SANE calls, and use the SANE patches that come with the FPE card.

Card numbers other than 0 or 1 are not currently in use. They are reserved in case other floating-point cards are produced for the Apple IIGS.

ignore

#pragma ignore flags

ORCA/C is an ANSI C compiler, adhering to the language specification defined by the American National Standards Institute (ANSI). ANSI C is actually the latest major standard in a long line of languages that have used the general name C; there are, in fact, five major dialects of C and countless minor variations.

Judging from the changes made in ANSI C, one of the concerns of the standards committee was to make C a safer language, catching many errors in the compiler than might have resulted in incorrect programs in earlier compilers. In general, this is a very good goal, but some older C programs no longer compile under ANSI C compilers. In addition, the language is under pressure to change in the direction of C++. The ignore statement tells ORCA/C to ignore certain kinds of checks that are required of ANSI C compilers so you can port older C programs a little easier.

The flags parameter is a series of bits, each controlling a specific error. If the bit is set, the error is *not* reported. In MPW IIGS ORCA/C, the default is always set to relax the requirements of the ANSI C standard. The flags that are currently available are:

bit	use
0	If this bit is set, the compiler does not report illegal characters in the source stream when the characters occur in code that is not processed by the compiler. For example, it is fairly common for programs to use the \$ character in file names, as in

```
#ifdef VAX
#include <sys$stdio.h>
#endif
```

This code is illegal under any properly implemented ANSI C compiler, whether or not VAX is defined. If bit 0 of the ignore pragma's flag word is set, and VAX is not defined, ORCA/C will not flag an error.

- 1 If this bit is set, the compiler allows multi-character char constants to be 3 or 4 bytes long, converting them to a longint value. If this bit is clear, only 1 and 2 character char constants are allowed.
- 2 If this bit is set, the compiler allows spurious tokens to appear after the #endif directive. The tokens are treated as a comment, and have no effect on the program. If this bit is clear, tokens after #endif are treated as an error.
- 3 If this bit is set, the compiler treats // as the start of a comment that extends to the end of the current source line. If this bit is clear, these characters are scanned as defined by the ANSI C standard.

keep

#pragma keep name

The keep command sets the name of the output file. A name is provided automatically if you are using the desktop environment, although you can use this directive to choose your own name. The name you choose cannot be the name of an existing file unless the file is an executable file or object module.

The single parameter is the name of the output file. As with the parameter for the #include command, this name is enclosed in quote marks or brackets, indicating if the file should be placed in the current directory or the ORCACDefs directory. In general, you would not want to place the file in the ORCACDefs directory.

The keep command must be used before the start of the first function in the program, and only one keep command can be used in a program.

lint

#pragma lint int

The lint pragma forces stricter checking of programs than is required by the ANSI C standard. The checks are individually enabled and disabled by setting and clearing bits in *int*. To enable more than one check, add the values for the bits shown in the table below to form a single integer. To enable all lint checks, use a value of -1.

- 1 Flag the use of a function before the function is defined as an error. This is always bad form, but this flag is also very useful in checking to insure that all header files that should have been included have, in fact, been included. If you missed including the header file for string.h, for example, but used strcat in your program, the compiler will flag an error for using strcat.

Error message: "lint: undefined function"

- 2 Flag functions with no types as errors.

Error message: "lint: missing function type"

- 4 Flag functions with no prototyped parameter lists as errors. If you are using tool header files that do not have prototyped headers, but you still want to use this check for the rest of your program, put the lint pragma after the #include statements that include the tool header files.

Error message: "lint: missing parameter list"

- 8 Flag pragmas that are not recognized by ORCA/C as errors. Normally, ANSI C compilers ignore pragmas they do not recognize, assuming the source file is being used with more than one compiler. Because of this, spelling errors in a pragma can go unnoticed by the compiler; this bit helps find this sort of problem.

Error message: "lint: unknown pragma"

memorymodel

#pragma memorymodel parm

The parameter is an integer constant. If the value is zero, the compiler will generate code for the small memory model; a non-zero value will generate code for the large memory model. You should use a value of 1 for the large memory model to allow for future expansion of the number of memory models.

The large memory model is the most flexible. When you use the large memory model, your program can have up to 64K bytes of global variables other than arrays, structures and unions. Arrays, structures and unions can be as large as memory will allow, and you can have as many of them as will fit in memory. In particular, arrays are not limited to 64K bytes, nor is the total space used by arrays limited to 64K bytes. Dynamically allocated memory can also exceed 64K for a single chunk of memory.

When you use the small memory model, all global variables, including arrays, structures and unions, are limited to a single 64K byte area of memory. This area of memory is shared with any functions you use from the standard C library, and if you do not use the segment statement, with your program's code. In addition, the compiler assumes that you will not allocate any single array, structure or union that is larger than 64K bytes using the Memory Manager or malloc. This restriction applies to any single structure, not to the total amount of space in use; you can allocate arrays, structures and unions whose total space exceeds 64K. These restrictions allow the compiler to generate code that is much smaller and faster than the code generated when the large memory model is used.

It is possible to create programs that are much larger than 64K bytes in length without using the large memory model. The segment statement can be used to split a program into more than one executable code segment. The large memory model is only needed if the total global variable space exceeds 64K bytes, or if you will be creating and manipulating dynamically allocated structures, unions or arrays where a single structure, union or array exceeds 64K bytes. When you can use the segment directive instead of the large memory model, it is best to do that, since the compiler generates smaller, more efficient code in the small memory model.

Regardless of the memory model used, you can use the segment statement to place the code for the program in various static or dynamic segments. In addition, both memory models limit the run-time stack size. For information about the run-time stack, see the stacksize command, below.

nba**#pragma nba start**

The `nba` command is used to create a HyperStudio New Button Action (NBA).

Normal C programs must contain a function called `main`. This function is the one executed when the program starts. NBAs do not have to have a function called `main`. Instead, you specify the name of the function that will be called when the NBA is executed as the `start` parameter to the `nba` command.

The function itself takes a single parameter of type `HSPParamPtr` and returns `void`. When the NBA is called, HyperStudio passes a pointer to a structure containing a variety of information. Information is also returned in this structure.

HyperStudio supports a variety of callbacks; these are calls from the NBA back to HyperStudio to perform some action. From ORCA/C, callbacks are made using the `__NBACALLBACK` function, defined in `HyperStudio.h`. (Note: The name starts with two `_` characters, not one.) This function requires two parameters: The callback number and a pointer to a structure like the one passed to the NBA. In most cases, you will actually pass back the same pointer passed to the NBA, but it is possible to make multiple copies of the structure. If you do make copies of the structure, though, be sure to initialize the copies carefully, generally by copying the original structure in its entirety into the copy. There are several fields in the structure that must be initialized properly for a callback to work.

For a definition of the `HSPParams` structure, along with a number of other useful declarations, see `HyperStudio.h`. For complete details on how to write HyperStudio NBAs, contact Roger Wagner Publishing.

nda**#pragma nda open close action init period eventmask menuLine**

The `nda` command is used to create a new desk accessory. New desk accessories are not executed like normal C programs; instead, they are copied into the `DESK.ACCS` folder in the `SYSTEM` folder, where they can be used from any desktop program that follows standard rules for the Apple IIGS.

A new desk accessory does not require a function called `main`, like standard C programs. Instead, there are four standard functions which must be defined, each of which is called by the Desk Manager to carry out some predefined task. The names of these functions are listed as the first four parameters to the `nda` command. The next two parameters are integer constants that specify how often the desk accessory is called when it is active (the `period` parameter), and what kinds of events it handles (the `eventmask` parameter). The last parameter is the name of the new desk accessory; the name appears in the Apple menu of any desktop program that can call the desk accessory. A more detailed description of each parameter is given below.

<code>open</code>	This parameter is an identifier that specifies the name of the function that is called when someone selects your desk accessory from the Apple Menu. It must return a pointer to the window that it opens.
<code>close</code>	This parameter is an identifier that specifies the name of the function to call when the user wants to close your desk accessory. It must be possible to call this function even if <code>open</code> has not been called. The function does not return a value.

action	The action parameter is the name of a function that is called whenever the desk accessory must perform some action. It must declare a single integer parameter, which defines the action that the function should take. See page 5-7 of the <u>Apple IIGS Toolbox Reference Manual</u> for a list of the actions that will result in a call to this function. The function does not return a value.
init	The init parameter is the name of a function that is called at start up and shut down time. This gives your desk accessory a chance to do time consuming start up tasks or to shut down any tools it initialized. This function must define a single integer parameter. The function will be zero for a shut down call, and non-zero for a start up call. The function does not return a value.
period	This parameter tells the Desk Manager how often it should call your desk accessory for routine updates, such as changing the time on a clock desk accessory. A value of -1 tells the Desk Manager to call you only if there is a reason, like a mouse down in your window; 0 indicates that you should be called as often as possible; any other value tells how many 60ths of a second to wait between calls.
eventMask	This value tells the Desk Manager which events your program can handle. The Desk Manager will only call your program with the events you specify in this mask.
menuLine	The last parameter is a string. It tells the Desk Manager the name of your desk accessory. The name must be preceded by two spaces. After the name, you should always include the characters \H**.

The `nda` command must be used before the start of the first function in the program. If it is used, you cannot use another `nda` command or a `cda` command.

For a complete discussion of new desk accessories, see page 58.

noroot

#pragma noroot

When ORCA/C creates a program, it generates two object files for each source file it compiles. The first object file has a suffix of `.root`; this file contains initialization code and a call to `main`. The second file has a suffix of `.a`, and contains the various variables and functions declared within the source file.

In a program consisting of multiple source files, the only source file that actually needs a `.root` segment is the one that actually contains `main`. This pragma can be used in all of the other source files to tell ORCA/C not to create a `.root` file, which will make the finished program slightly smaller.

optimize

#pragma optimize parm

The `optimize` command is used to control the level of optimization. The various features of the optimizer can make your program smaller and faster, but optimization takes time. During the

development cycle, when you are repeatedly compiling and testing your program, the time required by the optimizer is significant enough that you will probably not want to optimize your program

The optimize command requires a single integer parameter. This parameter is actually a series of flags. For each flag, a value of 1 turns the optimization on, and a value of 0 turns it off. Even if new optimizations are added in the future, then, the command

```
#pragma optimize -1
```

will enable all optimizations, while the command

```
#pragma optimize 0
```

will turn all optimizations off. If you do not use the optimize command, the compiler does not optimize your program.

Bit 0 (a value of 1) controls optimization of the intermediate code. If enabled, the intermediate code is scanned for dead code, which is removed; peephole optimization is performed; and constant expressions are evaluated at compile time.

Setting bit 1 (a value of 2) enables the native code peephole optimizer. This optimization scans the 65816 instructions produced by the compiler, replacing some sequences of instructions with equivalent, but more efficient, instructions.

Setting bit 2 (a value of 4) enables register optimizations. Since the 65816 only has three user registers, all of which have special, preassigned purposes, the compiler does not perform the kinds of register optimizations described in many compiler books. Instead, it scans the 65816 instructions for certain operations that can be avoided. For example, if a

```
ldy #2
```

instruction is detected, but the compiler knows that the value in the y register is already 2, the instruction is removed. While this optimization is technically another form of peephole optimization, the way the checks are performed is substantially different from the other peephole optimizations, so a separate optimizer is used.

Setting bit 3 (a value of 8) turns off the stack repair code normally generated by the compiler. This cuts the size of the code and makes it considerably faster, but if your program has any parameter passing errors, this optimization may cause the program to crash. For help in finding such bugs, see the debug pragma. For a description of legal parameter passing rules, see Chapter 10.

Setting bit 4 (a value of 16) enables common subexpression elimination. This optimization checks for repeated expressions, such as the array calculation for i+1 in

```
a[i+1] = b[i+1] + c[i+1];
```

these common subexpressions are calculated once and saved, generally making the program smaller and faster.

Setting bit 5 (a value of 32) enables loop invariant removal. This optimization checks any code sequence that loops back on itself for expressions which don't change inside the loop, removing these calculations from the body of the loop.

The optimize command is not supported by the small memory version of the compiler. The pragma is accepted, but has no effect on the finished program.

path

#pragma path name

The path pragma adds a folder to the list of folders the compiler searches when looking for a file.

By default, an include or append statement looks in the current folder when a file name is enclosed in quote marks, and in the library folder ({MPW}ORCACDefs:) when a file name is enclosed in <> characters. If the compiler does not find the file, it looks in the other of these two folders.

The path pragma adds a new pathname to the list of folders that are searched. The path name itself is the *name* parameter, which is a string. This string can be a partial pathname, in which case it is expanded to a full path name.

Paths added with the path directive are searched immediately after the current prefix, and are searched in the order that they are encountered by the compiler. For an example, assume a program includes these path pragmas:

```
#pragma path "myheaders"
#pragma path "network:projectx:headers:"
```

Then for the include statement

```
#include <stdio.h>
```

the compiler would look for the file using the following paths, stopping as soon as a file was found:

```
{MPW}ORCACDefs:stdio.h
stdio.h
myheaders:stdio.h
network:projectx:headers:stdio.h
```

For this include:

```
#include "secrets.h"
```

the compiler would look for the file using the following paths, again stopping as soon as a file was found:

```
secrets.h
myheaders:secrets.h
network:projectx:headers:secrets.h
{MPW}ORCACDefs:secrets.h
```

rtl

#pragma rtl

ORCA/C programs normally exit via a Quit call to GS/OS, but some otherwise normal programming environments require a program to exit with an RTL machine language instruction. The rtl command tells ORCA/C to create a program that exits with an RTL rather than a Quit.

The prime example of a program that must exit with a RTL instruction is an initialization program. There are two kinds of initialization programs, permanent initialization programs (file

type \$B6, PIF) and temporary initialization files (file type \$B7, TIF). For information about these kinds of programs, see Apple II File Type Notes for the appropriate file type.

stacksize

#pragma stacksize *parm*

All C programs use a run-time stack for calling functions and allocation of locally defined variables. This run-time stack is allocated from bank zero. There is no fool-proof way to determine how much memory will be available in bank zero when your program runs, although programs produced using ORCA/C will issue an error if there is not enough memory. Allocating too little memory for the stack is also a serious problem that will generally result in your program crashing, and can result in corrupting memory being used by the operating system, tools, or other programs. (See the debug command for one way to detect stack overflows.)

All of this means that you should allocate as small a stack as possible, but that there are serious penalties if the stack is too small.

By default, ORCA/C uses the stack allocated by the program launcher; this is 4K for the System 6.0 Finder, PRIZM 2.0 and the ORCA/M 2.0 shell. This memory is used by the startgraph and startdesk functions to allocate direct page space for tools. It is used in all C programs to allocate direct page space for SANE. It is also used for function calls and local variables defined by your program. The stacksize command lets you change the size of the stack. In the desktop environment, you can generally allocate a stack up to about 16K bytes; in the text environment, or in S16 programs, you can often allocate a stack of 32K bytes. The parameter for the stacksize directive is the size of the stack you wish to allocate, in bytes. If it is not a multiple of 256, the number is increased to the next even multiple of 256.

toolparms

#pragma toolparms *parm*

There are several instances that arise when using the toolbox where you need to create a function that a tool will call. Tools use a different mechanism for passing parameters than ORCA/C does, so you must tell the compiler to use the toolbox's conventions. Immediately before the function, you should tell the compiler to use the toolbox's mechanism for passing parameters by using the command

```
#pragma toolparms 1
```

Immediately after the function, switch back to the normal calling conventions using the command

```
#pragma toolparms 0
```

You do not have to use the toolparms directive if the function returns void and has no parameters.

You cannot call a function defined this way from within the C program, although the function can call other functions written in C.

This directive changes the stack model used to pass parameters from the one normally used by the ORCA languages to the model used by the Apple IIGS toolbox. It does not change the order of the parameters. In ORCA/C (and most other C compilers, for that matter) parameters are pushed onto the stack starting with the rightmost parameter and working to the left, while in the toolbox

(and almost all other high-level languages) the parameters are pushed onto the stack starting with the leftmost parameter and working to the right, just as you would read the parameter list. The `toolparms` pragma does not affect the order in which the parameters are pushed. To change the order of the parameters, use the pascal qualifier.

See also the `databank` command.

xcmd

`#pragma xcmd start`

The `xcmd` command is used to create a HyperCard XCMD or XCFN.

Normal C programs must contain a function called `main`. This function is the one executed when the program starts. XCMDs do not have to have a function called `main`. Instead, you specify the name of the function that will be called when the XCMD is executed as the `start` parameter to the `xcmd` command.

The function itself takes a single parameter of type `XCMDPtr` and returns void. When the XCMD is called, HyperCard passes a pointer to a structure containing a variety of information. Information is also returned in this structure.

HyperCard supports a variety of callbacks; these are calls from the XCMD back to HyperCard to perform some action. From ORCA/C, callbacks work almost exactly like tool calls, and from the programmer's standpoint they can be treated as tool calls. The various callbacks are declared in `HyperXCMD.h`, a header file provided by Apple and included with ORCA/C.

For a definition of `XCMDPtr` and the structure it points to, along with a number of other useful declarations, see `HyperXCMD.h`. For complete details on how to write HyperCard XCMDs, see the HyperCard folder on the System 6.0 CD ROM..

Chapter 7 – Basic Data Types

C has a rich variety of data types. This chapter describes those C data types which are built into the language. The next chapter covers derived and user-defined data types.

Some of the information in this chapter deals with the way information is stored internally in the memory of the computer. This information is provided for very advanced programmers who need to write assembly language subroutines that will deal with C data, or who need to do strange and dangerous tricks with the data to work with the machine at the hardware level. You do not need to understand this information to use ORCA/C for normal C programming. If it does not make sense to you, or if you will not be using the information, simply ignore it.

Integers

The C language supports four different types of integers, each of which can be signed or unsigned. The four basic integer data types are char, short, int, and long. ANSI C specifies the minimum range for each of these integer data types. While older C compilers may not implement the integers using these same sizes, and new compilers are allowed to make the integers larger, the minimum sizes specified by ANSI C also happen to be the most common size for integers on microcomputers, and are the sizes used by ORCA/C.

Integers defined as char are the smallest. Char values require one byte of storage. Signed char values can range from -127 to 127; unsigned char values have a range of 0 to 255. The C language allows the compiler implementor to decide if char values will be signed or unsigned by default; in ORCA/C, char values are unsigned by default. All other integer types are required to be signed integers by default.

Short and int variables are the same size. They require two bytes of storage. Signed values can range from -32767 to 32767, while unsigned values can range from 0 to 65535.

Long integers require four bytes of storage. Signed long integers range from -2147483647 to 2147483647, while unsigned long integers range from 0 to 4294967295.

Internally, all unsigned integers are represented as binary values. Signed integers are represented as two's complement numbers. All of the integers that occupy more than one byte of storage are stored with the least significant byte first, proceeding to the most significant byte.

Reals

ORCA/C supports four storage formats for real numbers. In all cases, calculations are performed using SANE or the Innovative Systems' 68881 floating point card using ten byte intermediate values. With some of the formats, the numbers are stored with less precision, so that you may see different results if you compare one equation that does not save intermediate results with a mathematically equivalent set of equations that store intermediate values in the less precise number formats.

Float and double are required in all C compilers. Float numbers are stored in the IEEE floating point number format, with the least significant byte first. They require four bytes of storage. Float values are accurate to seven decimal digits, and allow exponents with a range of about $1e-38$ to $1e38$.

Double numbers require eight bytes of storage each. They are stored least significant byte first, using the IEEE floating point format. They are accurate to fifteen decimal digits, and allow exponent ranges from $1e-308$ to $1e308$.

Extended numbers are peculiar to implementations of C on Apple computers. They require ten bytes of storage each. The storage format is an extended version of the IEEE format, with the values stored least significant byte first. They are accurate to nineteen decimal digits, with an exponent range of $1e-4932$ to $1e4932$. This is the format used internally by both SANE and the 68881 floating point coprocessor, so there is no loss of precision if intermediate variables are of type extended. In addition, floating point calculations are performed faster if you are using SANE and all numbers are stored in extended format, since a conversion from the original format to extended format is always a first step in doing a calculation with SANE. Exactly the opposite is true if you are using the 68881 floating point card. In that case, conversions are done in hardware. Float variables are faster with the 68881 card, since it takes less time to pass the number to the card and retrieve the result with the shorter formats.

Comp format numbers are signed integers that require eight bytes of storage. They range from $-(2^{63}-1)$ to $2^{63}-1$. While comp numbers are integers, they are treated as a special form of floating-point number.

Pointers

Pointers are represented internally as four-byte unsigned numbers, with the least significant byte stored first. A value of zero is used to represent a null pointer. Using type casting, pointers can be treated as unsigned long integers in mathematical equations with no loss of precision.

Chapter 8 – C Programs

The Anatomy of a C Program

C programs are made up of a series of declarations. These declarations consist of declarations of global variables and declarations of functions. In most C programs, some of the functions and global variables are defined in libraries or separately compiled modules. The compiler is informed of their existence through interface files.

While macro preprocessor commands are used in C programs to define constants and implement some operations, the macro preprocessor is not technically a part of the C language. For a description of the macro preprocessor, see Chapter 6.

The Function main

All C programs except classic desk accessories and new desk accessories must have a single function called main. This function is the first one called; when you return from main, program execution stops.

It is not important where main appears in a program; if the program consists of several modules, main can appear in any of them, regardless of the order in which the modules are compiled or linked. The important point is that a function named main must appear in exactly one of the modules.

You may declare the function main as either a function returning an integer or as a function returning void. Declaring it as a function returning an integer allows you to send a return code back to the caller. If your program is designed to run from script files, this is an important step. It allows you to tell the script file that the program terminated normally by returning zero, or to inform the shell that an error occurred by returning a non-zero value. If you do not return a value from main, the value reported to the shell is unpredictable.

Argc and Argv

The shell passes two arguments to the function main. These arguments can be ignored or used; to ignore them, don't code any parameters to the function. These arguments allow you to look at the command line used to execute your program. The sample program below shows how these arguments are used.

```

/* echo the command line arguments */

#include <stdio.h>

int main(argc, argv)

int argc;
char *argv[];

{
int i;

for (i = 0; i < argc; ++i)
    printf("%2d: %s\n", i, argv[i]);
}

```

In this program, you see `argc` and `argv` defined as parameters to `main`. While you do not have to define `argc` and `argv`, if you do, they must be defined in the order shown. When your program executes, the command line used to execute your program is scanned. If you executed the program from the text environment or from the shell window on ORCA/Desktop, the command line is the line you typed or that appeared in a script file that actually executed your program. If you execute your program using the RUN command from the shell, or using the Compile to Memory or Compile to Disk commands from the desktop, the command line will be empty.

After I/O redirection characters are removed, aliases are expanded, and any shell variables are expanded, the command line is broken into strings. These strings are formed by scanning the command line from left to right. A string consists of any sequence of characters that contain no white space. `Argc` is the number of strings found, while `argv` is an array of pointers to the strings. If any strings are present, the first (`argv[0]`) is always the name of your program.

To see how these strings are built, and to experiment with the way lines are scanned, type in the sample shown above. If you are using the desktop environment, be sure to turn debugging off. Then execute the program from the text shell or shell window, supplying a variety of parameters.

Separate Compilation

C programs can be divided into more than one module, and each module can be compiled separately from the others. The resulting object files can then be linked together, producing a single program.

The object files that make up a program are not limited to object files created by the C compiler. You can mix C code with code written using ORCA/Pascal, the APW assembler or the ORCA/M assembler.

Chapter 9 covers the storage types used to hide variables and functions from other modules, and the storage types used to make functions and variables from one module available to other modules.

Interface Files

Interface files are used to tell the C compiler about functions and data that appear in libraries or separately compiled modules, and to define macros that are used in more than one module. The distinction between an interface file and a file that contains C source code is merely a matter of convention. Interface files are created using the same text editor that you use to create programs.

They can be modified by you. You can even extract declarations from an interface file and embed them directly in your program.

In general, interface files come from two sources. ORCA/C comes with a large number of interface files that help you use the standard C libraries and the Apple IIGS toolbox. These interface files are located in the ORCACDefs folder of your library directory. To include one of these interface files in your program, use a `#include` command, and place the name of the interface file in brackets, like this:

```
#include <stdio.h>
```

When you read the descriptions of the standard C libraries in Chapter 13, each description shows the `#include` command that should be in your program to call the function. With some functions, you can avoid using the interface file, and that was quite common in older C compilers. ORCA/C implements function prototypes, though, which allow the compiler to perform sophisticated checking on the parameters you pass to a function. If you do not use the interface file, the compiler cannot make these checks. Since illegal parameters account for a large number of bugs in C programs, many of which can crash the executing program, it is a good idea to make use of the include files.

The second major source of include files are those that you write yourself. These can be new libraries that you have created, in which case the interface file should be placed in the ORCACDefs folder and accessed like any other library, or they could be interface files used in a program that is made up of separately compiled modules, in which case you would use quote marks around the file name in the `#include` command, instead of brackets. For example, if you need to include an interface file called `commondefs` in your program, and the interface file is in your current working directory, you could use the statement

```
#include "commondefs"
```


Chapter 9 – Declarations

Declarations

A program is made up of one or more declarations, one of which must be a function called `main`. The basic syntax for a program is shown below.

```
program:    {initialized-declaration | function-definition | asm-function-
            declaration}*
initialized-declaration:
            declaration-specifiers initialized-declarator
            {',' initialized-declarator}* ';'
initialized-declarator: declarator {'=' initializer}
function-definition:
            {declaration-specifiers} declarator {declaration-list}
            compound-statement
declaration-list: declaration {',' declaration}
declaration: declaration-specifiers declarator-list ';'
declaration-specifiers: {storage-class-specifier | type-specifier}*
declarator:
            identifier |
            '(' declarator ')' |
            function-declarator |
            array-declarator |
            pointer-declarator
```

Storage Classes

```
storage-class-specifier: [auto | extern | register | static | typedef]
```

Each function or variable has a storage class. This storage class may be assigned explicitly by starting the declaration with one of the storage class names shown above.

For global variables and functions, if no storage class is specified, a storage class of `extern` is assumed. There is a difference, however, between a variable or function defined with the storage class `extern`, and a variable or function defined with no explicit storage class. When the storage class of `extern` is specified explicitly, the compiler creates a reference to the variable or function, but it is not defined. Instead, the compiler expects that the variable or function will be resolved later by the linker. When no storage class is specified, the compiler generates a variable or function that can be accessed from outside of the current module. Only one such occurrence is allowed in any one program.

```
extern int i;           /* an integer declared in another module */
extern foo();           /* a function declared in another module */

int i;                 /* an integer which can be accessed from */
                       /* another module */
```

For variables defined within a function, a storage class of `auto` is assumed. Auto variables are available throughout the function, but cannot be accessed from outside of the function. Auto variables defined within a function lose their value when the function returns to the caller.

```
int i;                /* auto integer */
```

The static storage class has three different meanings, depending on where it is used. Functions definitions of type static are not exported to the linker. They become private to the current module, and cannot be accessed from other modules, whether or not the code is turned into a library. Function declarations (where a function is declared, but the statements that make up the function are not specified) of type static indicate that the function body will appear later in the current source file, and that the function will be of storage class static. Global variables defined with a storage type of static cannot be accessed from outside the current module. Local variables defined with a storage type of static remain intact after leaving the function. That is, any value assigned to the variable during a function call is still available when the function is called the next time.

```
static int i;         /* static integer */
static foo();         /* a function that will be defined later */
```

The register storage class can only be used with variables defined within a function. It has the same meaning as auto, but instructs the compiler to handle the value in the most efficient way possible, placing the value in a register if one is available. The 65816 does not have enough user registers to leave values in a register, so in ORCA/C, this storage class has exactly the same meaning as auto.

Unlike any other storage class, this storage class can be used with a function parameter. Again, though, ORCA/C does not do anything special with the parameter if the storage class is register.

```
register int i;        /* register int */
```

The storage class typedef is used to define types. The name of the type appears where the variable name would have appeared in any other declaration, and the new type has the same type that the variable would have been assigned if typedef were not used. The new type can then be used as a type specifier (see the next section).

```
typedef int *intPtr; /* definition of a type: pointer to integer */
intPtr ip;           /* defining ip as a pointer to an integer */
```

Type Specifiers

```

type-specifier:
    {volatile} {const}
    [ enumeration-type-specifier |
      floating-point-type-specifier |
      integer-type-specifier |
      structure-type-specifier |
      identifier |
      union-type-specifier |
      void ]
    {volatile} {const}
floating-point-type-specifier:
    float |
    {long} double |
    extended |
    comp
integer-type-specifier:
    signed |
    {signed} int |
    {signed | unsigned} short {int} |
    {signed | unsigned} long {int} |
    unsigned {int} |
    {signed | unsigned} char

```

The type specifier specifies the type for a variable or function. The type specifier for a function is optional, whether or not a storage class specifier is used; if omitted, int is assumed. A variable declaration must include either a storage class specifier or a type specifier (and can, of course, include both). If the type specifier is omitted, a type specifier of int is assumed.

Type specifiers for enumerations, structures and unions are covered in separate sections, below.

Floating-point and integer type specifiers are used to declare variables, pointers, and arrays of the various built-in data types. They all have a similar structure. Floating-point numbers come in three sizes, as detailed in Chapter 7. The smallest size requires four bytes of storage; this size is designated by the type specifier float. Double-precision numbers, requiring eight bytes of storage, can be specified as double or long double. SANE extended format numbers, requiring ten bytes of storage each, have a type specifier of extended.

SANE also supports an eight-byte signed integer format called comp. Comp variables are treated as a special case of floating-point numbers, despite the fact that they do not support exponents or non-integer values.

```

float a,b;           /* two floating-point variables */
double sum;          /* sum is double-precision */
long double sum2;     /* sum2 is double-precision */
double *dp;           /* pointer to a double precision variable */
extended a[10];       /* array of 10 extended variables */
extern float sum();    /* external function returning float */
comp big;             /* eight-byte signed integer */

```

Integer type specifiers have a variety of formats, but are easy to understand once you know the systematic way they are defined. There are basically four types of integers: char, short, int and long. Since short and long are considered to be modifications of the basic type of int, you can follow either of these type specifiers with int, but you are not required to do so, and most programmers do not. Any of the four basic integer data types can be signed or unsigned. By default, they are all signed. If you like, you may prefix any of the types with signed, ensuring that

the value is signed even if the source code is ported to another compiler. This is a good idea for char variables, since some compilers use unsigned char by default, but doesn't matter on the other integer data types, which are always signed by default. You may also create unsigned versions of each of the data types by prefixing it with unsigned. When an unsigned integer suits your needs, it is a good idea to use them. On the Apple IIGS, code for comparing unsigned values is much more efficient than code for comparing signed values. The examples below show some typical cases.

```
foo();           /* function returning int */
int i,j,k;       /* three signed integer variables */
signed char *cp; /* a pointer to a signed character */
unsigned long l[10]; /* an array of 10 unsigned long integers */
long int l;      /* a long int */
short sum;       /* a short integer */
```

When you define a type using the typedef storage class, the name of the type (and identifier) can be used as a type specifier in later declarations. For example, let's assume that a pointer to an integer has been defined as a new type:

```
typedef int *intPtr;
```

You can now use this type to define variables and function results.

```
intPtr ip;          /* ip is a pointer to int */
intPtr search();    /* search is a function returning a pointer to int */
```

Void is a special type specifier that declares a function that does not return a value, or that a pointer points to a generic type. There are three places where this type specifier can be used. Functions returning void are functions that do not return a value. In other languages, these are known as subroutines or procedures, and the term function is only used when a result is returned. Older C programs, written before void became a part of the language, defined functions returning int for this purpose, depending on C to dispose of the unneeded value. There are two reasons to use void explicitly on new programs: it avoids confusion, and possibly errors, and ORCA/C generates more efficient code when exiting a function of type void than when exiting a function returning int.

The second place where void is used is to define an untyped pointer. These should be avoided when possible, but are occasionally useful. The most common place to use this type of pointer is in functions that require a pointer to a data structure whose type can change from call to call.

Finally, void is used in type casting to cast a pointer to an untyped pointer, making the pointer type-compatible with any other pointer. In older C programs, a pointer to char was generally used for pointers to an unspecified type.

```
void *untyped_pointer; /* an untyped pointer */
void fn(void);         /* a function that returns nothing */
```

Volatile is a type specifier that modifies a type. If it appears with no other type, a type of int is assumed. Volatile can appear before or after any other type specifier. Volatile is an instruction to the compiler, telling it that the value can be changed in ways that are not under control of the compiler. Examples would be hardware ports like the Apple IIGS keyboard latch at 0x00C000, or variables that will be changed by interrupt handlers. The compiler is not allowed to do optimizations on volatile variables that would delay references to the variable, or change the order in which references occur.

Const is another type specifier which modifies an existing type. If it appears with no other type, a type of int is assumed. Const can appear before or after any other type specifier. Const

variables cannot be changed by the compiler, although they can be initialized. Any attempt to change the variable will result in a compile-time error.

Enumerations

```
enumeration-type-specifier:
    enum {identifier} |
    '{' enumeration-constant {',' enumeration-constant}* '}'
enumeration-constant: identifier {'=' expression}
```

Enumerations provide an easy way to create integer constants. They are used in situations where a series of names make the program easier to read than using integer values. For example, when dealing with the 640 pixel wide screen, you have four colors available. While they can be changed, these colors default to black, purple, green and white, with integer values for the colors of 0 through 3. You could write a program that uses the integers 0 through 3 to represent these colors, or you could use preprocessor macros to create names for the colors. Enumerations provide yet another alternative. The enumeration

```
enum {black, purple, green, white} pencolor;
```

defines the integer variable `pencolor`, and simultaneously defines four integer constants. The values of these constants start at zero and increment by one, so that

```
printf("%d\n", green);
```

would write 2 to standard out.

An enumeration can be used to define a new enumeration type by including an enumeration tag before the list of enumeration constants. For example,

```
enum color {black, purple, green, white} pencolor;
```

In this case, we have still defined the variable `pencolor`, but you can define an enumeration type without creating variables. The enumeration type `color` can now be used to define other variable with the same enumeration type.

```
enum color frameColor, buttonColor, menuColor;
```

By default, enumeration constants are assigned values by setting the first to zero, then incrementing each successive enumeration constant by one. It is possible, however, to set each constant to an explicit integer value. Once this is done, if the next constant does not have an explicit value, a value is assigned to it by incrementing the last constant value by one, as before. To do this, follow the constant by an equal sign and a constant expression, as shown below.

```
enum month {January = 1, February, March, April, May, June, July, August,
            September, October, November, December};
enum ages {George = 28, Matilda = 28, Jimmy = 5, Cindy};
```

In the first case, integer values for months are assigned sequentially, but a starting value of one was needed for January. The second example shows the ages of everyone in a family. Note that it is legal to specify the same value for two different enumeration constants. Cindy's age would be six, since no specific value was specified, and the previous constant has a value of five. While these examples do not show it explicitly, any integer value may be used, including negative values.

Arrays

array-declarator: declarator '[' {constant-expression} ']'

Arrays are indexed lists of data, where all of the data in the array has the same type. In C, you can form arrays of any type except functions (although arrays of pointers to functions are legal), including arrays of structures, unions, or other arrays. To define an array, follow the declarator for the variable with a left bracket, a constant expression, and a right bracket. The constant expression is evaluated, and used to determine how many elements are in the array. The first element is indexed with a value of zero, and remaining indices are formed by adding one to the previous index.

For example, to form an array of ten integers, and then fill the array with the numbers one through ten, you could use the following statements:

```
int a[10],i;

for (i = 0; i < 10; ++i)
    a[i] = i+1;
```

It is important to keep in mind that the lowest index is zero, and the highest is one smaller than the number of subscripts. So, in the example just given, there is no tenth element of the array. That is, `a[10]` does not exist, and storing a value at `a[10]` may cause the program to crash.

Arrays of more complicated elements are formed the same way as arrays of simple variables. For example, to form an array of points, where a point is defined as a structure containing three float numbers, you could use

```
struct point {float x,y,z};
struct point list[100];
```

C does not allow multiply subscripted arrays, but it does allow arrays of arrays, which amount to the same thing. For example, to create a ten by ten matrix of float values, you would use the declaration

```
float a[10][10];
```

The following code shows how to access elements of the array by forming the identity matrix (a matrix with ones along the diagonal, and zeros everywhere else).

```
for (i = 0; i < 10; ++i) {
    for (j = 0; j < 10; ++j)
        a[i][j] = 0.0;
    a[i][i] = 1.0;
}
```

In some cases, it is permissible to define an array without specifying the size. This is true for singly-dimensioned arrays that are external to the current unit, arrays that are passed as a parameter to a function, and arrays that are initialized. In the first two cases, the compiler does not need to know the exact size of the array, since storage for the array is allocated elsewhere. It is up to you to make sure that you do not access values beyond the end of the array. Even in these cases, if the array is an array of arrays, all subscripts but the first must be specified. In the case of an initialized array, the compiler determines the size of the array by counting the initializers. The following example shows legal and illegal declarations of external arrays.

```
extern int a[][10]; /* legal: ? by 10 array of int */
extern int b[10][]; /* NOT LEGAL! */
int a[] = {1,2,3}; /* legal: three elements */
```

The array is stored in memory in such a way that the first subscript indexes the slowest. To visit sequential locations in memory, you would use `a[0][0]`, `a[0][1]`, `a[0][2]`, and so forth.

The size of an array is the number of elements in the array times the size of an array element. In the previous example, each floating-point number requires four bytes of storage, so an array with ten elements requires forty bytes of storage. The array of these ten element arrays also has 10 elements, so the array `a` requires 400 bytes of storage. In a program, you would normally use the `sizeof` operator to determine the size of an array.

Using the small memory model (which is the default), the largest single array that is allowed is 64K bytes. In addition, the total length of all variables, libraries, and code from your program that is not placed in another segment using the segment command is 64K bytes. The large memory model lifts these restrictions at a price of less efficient code. With the large memory model, an array can be as large as the largest free block of memory, so long as there is enough memory to load all global variables and static segments. With a memory card with enough memory, you could manipulate an array that is nearly eight megabytes long.

Pointers

```
pointer-declarator: '*' {type-specifier}* declarator
```

A pointer is a data type that holds the address of another object. Pointers are represented internally as unsigned four-byte values. The value is the address of the object pointed to by the pointer. A pointer is defined by placing an `*` to the left of the variable, as in

```
int *ip /* ip is a pointer to an integer */
```

Two operators are used to handle pointer types. The `*` operator, when used before a pointer, tells the compiler to load the object pointed to by the pointer. The `&` operator, when used before any l-value, tells the compiler to load the address of the object, rather than the actual object. The following code fragment uses the `&` operator to cause `ip` to point to the integer `i`, then sets `i` to four using a reference through the pointer, and finally prints the value of `i` directly and as a reference through the pointer `ip`.

```
ip = &i; /* ip now points to i */
*ip = 4; /* i now has a value of 4 */
printf("%d = %d\n", i, *ip); /* both values printed will be 4 */
```

Pointers can be defined to point to any other data type except a bit field, including other pointers, functions or void.

ANSI C allows a pointer to have a type specifier before the declarator. This is generally used with the type specifiers `volatile` and `const`, but the type specifier is not restricted to just those two types. `Const` and `volatile` can be used in conjunction with other types, so they modify the existing type of the pointer. For example, the declaration

```
int * const ip;
```

declares a pointer to an integer, and states that the pointer cannot be modified. If a type specifier other than `const` or `volatile` is used, it overrides the existing type. For example, the following declaration creates a pointer to a float variable.

```
int * float fp;
```

The best that can be said for such a declaration is that it is legal. Such declarations are confusing, and should not be used in real programs.

There are many other operators that can be used with pointers in C, including the array subscript operator, the increment and decrement operators, addition and subtraction, and comparisons. Chapter 11 discusses these topics.

Structures

```
structure-type-specifier:
    struct identifier |
    struct {identifier} '{' field-list '}'
field-list: component-declaration {' ',' component-declaration}*
component-declaration: type-specifier component {' ',' component}*
component:
    declarator |
    {declarator} ':' expression
```

Structures are collections of variables that do not have to have the same type. With the exception of bit fields, each variable in a structure has a name, just as variables in other parts of the program do. (The name is optional on a bit field.) Variables in a structure are called components, and the names of the variables are called component names. Each of the component names must be distinct from any other component name in the same structure. A structure can contain elements of any type except a function or void. Pointers to functions are allowed, but a function declaration cannot appear as a part of a structure.

It is possible to define a structure type without defining variables, and then define variables later; to define a structure variable without defining a type; and to define both a type and variables at the same time. We will look at the general form by examining a structure that implements a linked list of points, with each point consisting of three float numbers.

```
struct point {struct point *next; float x,y,z;} p, *list;
```

This structure contains four elements: next, x, y and z. X, y and z are the three float variables that record the position of the point. Next is a pointer to another structure of type point. Note that it is permissible to use the definition of the structure point, so long as we are defining a pointer to the structure. Trying to recursively define the structure would be illegal, since the compiler could not compute the size of the structure. Finally, two variables are defined. The first, p, is a structure of type point, while the second is a pointer to a structure of type point.

Further variables of type point may be defined by using the structure name (called the tag) without redefining the form of the structure. For example, we could define three more points like this:

```
struct point p1, p2, p3;
```

The tag field of the structure is optional. If you omit the tag field, however, no other variables of the same type can be created later in the program.

The size of a structure can be determined by summing the size of each component. In the structure point, there are four elements, each of which is four bytes long, so the size of the variable p is sixteen bytes. The values appear in memory in the same order in which they are defined. In our example, next occurs first, followed by x, then y, and finally by z.

To access a component of a structure, the name of the structure is followed by a period and the name of the component name of the variable to be accessed. For example, to initialize the point `p` to (1.0,2.0,3.0), with a null pointer, we would use the assignments

```
p.x = 1.0;
p.y = 2.0;
p.z = 3.0;
p.next = NULL;
```

To set `p1` to the reflection of `p` through the origin, we could use the statements

```
p1.x = -p.x;
p1.y = -p.y;
p1.z = -p.z;
```

To access an element of a structure through a pointer to the structure, the `->` operator is used. For example, to move the point pointed to by `list` two times further away from the origin, you would need to multiply each of the coordinates in the point by two. The following statements will make this change.

```
list->x = list->x*2.0;
list->y = list->y*2.0;
list->z = list->z*2.0;
```

Structures may be assigned to other structures of the same type. For example, using the declarations from our examples, the following assignments are legal.

```
p1 = p2;
*list = p;
p3 = *list;
p1 = *p.next;
```

Functions can be defined which accept structures as parameters and which return structures as results. The following example shows a function which takes two points as parameters, and returns a point midway between the two inputs.

```
struct point midpoint(struct point a, struct point b)
{
    struct point c;
    c.x = (a.x + b.x) / 2.0;
    c.y = (a.y + b.y) / 2.0;
    c.z = (a.z + b.z) / 2.0;
    return c;
}
```

Using this function to find a point midway between `p1` and `p2`, storing the result in `p3`, we would code

```
p3 = midpoint(p1,p2);
```

Bit fields are a data type peculiar to structures. A bit field is an integer data type which is bit-aligned, rather than byte-aligned. They can be used to store integers in a very compact fashion, or to access bits within a byte of memory. To see how this is done, we will define a simple structure using bit fields.

```

struct pack {
    unsigned field1 : 9;
    unsigned       : 2;
    unsigned field2 : 4;
} v;

```

Each of the fields in this structure requires a specific number of bits, not bytes. Field1 requires nine bits; it takes up all of the first byte, and the first (most significant) bit of the next byte. It is an unsigned integer, with a range of 0 to 511. The next bit field has no name; it is only there to reserve a specific amount of space. Field2 requires four bits of space, and has a range of 0 to 31. The variable v, then, requires fifteen bits of space. In all cases where a series of bit fields does not end on a byte boundary, the compiler in effect creates another field to fill out the bits to an even byte boundary. In this example, one bit must be added, so that the variable v uses sixteen bits (two bytes). The two bits between field1 and field2, and the bit that comes after field2 cannot be accessed from the program, and their values are not predictable.

Bit fields may be interspersed with other variables in a structure, as shown in the second example, below.

```

struct data {
    unsigned color640 : 2;
    int i;
    unsigned color320 : 4;
} v;

```

The variables which are not bit fields must start on a byte boundary. Thus, the variable color640 only requires two bits, but since the variable i must start on a byte boundary, the entire structure requires four bytes of storage (one each for color640 and color320, and two for i). A simple rearrangement of the structure reduces the memory requirements by one byte.

```

struct data {
    unsigned color640 : 2;
    unsigned color320 : 4;
    int i;
} v;

```

ORCA/C supports both signed and unsigned bit fields. Unsigned bit fields are stored in binary format. The range that can be represented is 0 to 2^b-1 , when b is the number of bits in the bit field. Signed values are represented in two's complement form, giving a valid range of -2^{b-1} to $2^{b-1}-1$. The maximum size for a bit field is thirty-two bits.

Accessing bit fields is very inefficient compared to accessing integers. If speed is an issue, avoid their use.

Programs that make use of bit fields are difficult to port from machine to machine, so bit fields should be used sparingly, if at all. Not all compilers support signed bit fields. The maximum size for a bit field varies from compiler to compiler, and the way bit fields are stored internally also varies.

Unions

```
union-type-specifier:
    union identifier |
    union {identifier} '{' field-list '}'
```

A union has a syntax similar to that of a structure and, like structures, contains named elements called components. Unlike structures, bit fields are not allowed in unions. Like structures, components of unions can be of any type except function returning or void. Within any one union type, each component name must be unique. No component can be of the same type as the union in which it appears, but components can point to unions of the same type.

Components of unions are accessed the same way as components of a structure. Unions can be assigned to other unions of the same type and returned as the result of a function, just like structures. The major difference between structures and unions is in how storage is allocated for the components. In a structure, memory is assigned to each variable in turn, and the total size of the structure is the sum of the sizes of all of the components. In a union, each of the variables overlaps. The size of the union is the size of the largest component, since only one of the components is stored in the union at any one time. Unions are most useful in situations where two or more types of data will be stored in a location, but the two do not need to be stored at the same time. As an example, let's consider a program that evaluates expressions, and needs to store variable values. We will assume that the name of the variable must be stored, and that it is limited to ten characters. We will also assume that the variables can be integer or float. Since a single variable couldn't be both integer and float at the same time, we will use a union to overlay the int and float variables, saving space.

```
enum kind {integer, real};
struct variable {
    char name[11];
    enum kind vkind;
    union {
        int ival;
        float rval;
    } val;
};
```

This example also shows the use of a tag field to record the type of value stored in the union. While this is not required, it is often useful. Without a tag field, your program must have some other way of figuring out if the value stored in the union is an integer or a floating-point number.

To evaluate the storage requirements for the structure, we first determine the size of the union. Ival is two bytes long, while rval is four bytes long. The union, then, is four bytes long. The name of the variable requires eleven bytes, and the tag field, which is an integer, requires another two bytes. The total size of the structure, then, is seventeen bytes.

Initializers

```
initializer:
    expression |
    '{' initializer-list {' ',''} '}'
initializer-list: initializer {' ','' initializer}*
```

When a variable is defined, the declaration can include an initializer which specifies the initial value for the variable. Variables with a storage class of static and extern can only be initialized with

a constant expression. All static and extern variables that are not explicitly initialized are initialized with a value of zero. Function parameters cannot be initialized.

Integer, enumeration and floating-point variables are initialized by following the variable name with an equal sign and the initial value. The initial value can be enclosed in braces, but the usual practice is to omit the braces. Non-constant expressions can be used to initialize variables that have a storage class other than extern or auto. In that case, the compiler generates the same code that would be generated if the variable was initialized via an assignment statement.

The following examples show some legal initializations.

```
i = 4;
auto j = i*4;
static float x = 1.0, y = 2.0, z = 0.0;
```

An enumeration can be initialized to an enumeration constant or to an integer value. As with integers, the expression must be a constant expression if the storage class of the enumeration variable is extern or static. For example, we can define an enumeration, declare a variable, and assign the variable an initial value all in one step, like this:

```
enum color {black,purple,green,white} pencolor = white;
```

Initialization of pointers follow the same rules as initialization of integers. The following operands are all constants, and can be used in a constant initializer for a pointer:

1. The integer constant 0 (or the preprocessor macro NULL).
2. The name of a static or external function.
3. The name of a static or external array.
4. The & operator when applied to a static or external variable.
5. The & operator when applied to a static or external array with constant subscripts.
6. A non-zero integer constant cast as a pointer type.
7. A string constant.
8. An integer constant added or subtracted to any of the items 3 through 7.

Arrays are initialized by enclosing the initializers in braces, and separating them with commas. Each initializer in an array must be constant expressions. The example below shows the initialization of a ten-element array.

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
```

Multi-dimensioned arrays follow the same pattern:

```
int a[2][2] = {{1,2},{3,4}};
```

In this case the initial values are as follows:

a[0][0]	1
a[0][1]	2
a[1][0]	3
a[1][1]	4

There are four special rules used when dealing with array initializers. First, if an initializer for an array contains fewer values than the size of the array, the remainder of the array elements are initialized to zero. The second rule is that the size of an array does not need to be specified. In that case, the size of the array is derived from the number of initializers. For example, the following array has five elements, initialized to one through five.

```
int a[] = {1,2,3,4,5};
```

Another special case is an array of characters. An array of characters can be initialized using a string, as in

```
char str[15] = "Hello, world.\n";
```

The compiler automatically places a terminating null character at the end of the string constant. Finally, if the initializer list contains the proper number of elements, all embedded braces may be omitted. Repeating our previous example using this rule, we have

```
int a[2][2] = {1,2,3,4};
```

Structures are initialized by enclosing all of the values in braces and separating them with commas. As with arrays, if the number of values supplied is less than the number of variables in the structure, the remaining elements are set to zero. In addition, if a structure is embedded in another structure, and the correct number of initializers are supplied, the braces around the initializers for the embedded structure may be dropped.

```
struct point {float x,y,z;} p1 = {1.0,2.0,3.0};
struct line {point p1,p2} line1 = {1.0,2.0,3.0, 2.0,3.0,4.0};
struct line2 = {{1.0,2.0}, {2.0,3.0}};
```

The first component of a union can be initialized with an expression resulting in the same type as the component. If the storage class for the union is `auto`, the first component must have a type that could be initialized if the component were specified as a separate `auto` variable. For example,

```
union nums {float f; int i;} x = 0.0;
```

For all global or static variables, if no initializer is given, the variable is initialized to zero.

The way initializations are handled by the compiler depends on where the variable is declared. Global and static variables are initialized by setting the memory area reserved for the variable to the correct initial value. Auto and register variables defined in a function are initialized using code that is equivalent to an assignment statement.

If a variable is declared with an explicit storage class specifier of `extern`, it cannot be initialized.

Constructing Complex Data Types

The facilities described in this chapter show how to declare variables in a variety of simple data types. You can also combine many of these attributes, creating complex definitions in a single step. For example,

```
float (*arr[10])();
```

defines a ten-element array of pointers to functions returning `float`. With some restrictions, any combination of storage class specifiers, type specifiers and declarators can be used to form a type. These restrictions are:

1. Void can only be used as the type returned by a function or pointed to by a pointer.
2. Arrays, structures and unions may contain pointers to functions, but may not contain functions.

3. Functions can return structures, unions or pointers to arrays, but they cannot return an array.
4. Functions cannot return another function. They can return a pointer to another function, however.

To read the type of a variable, it is important to understand the precedence of the declarator operators. Basically, declarators that appear to the right of the variable (arrays and functions) have a higher precedence than declarators appearing to the left (pointers). The declarator that is closest to the variable has the highest precedence. Parentheses can be used to override the normal precedence. To see how this is done, we will use an absurdly complex declaration.

```
struct point {float x,y,z;} *(*(*(*x)())[10])());
```

Starting at the variable, we can read off the type. The variable `x` is a pointer to a function returning a pointer to a ten-element array of pointers to functions returning pointers to structures that contain three float variables. In a real program, such a declaration would probably not be needed. If such a declaration was needed, it would be easier to define using typedefs.

Scope and Visibility

All declarations that appear outside of a function are available from the declaration point to the end of the source file, unless some other declaration masks the declaration.

Function parameters are available throughout the function unless some other declaration masks the parameter. Parameters cannot be accessed from outside the function, even from a function called by the one that defined the parameters. (Pointers to variables can, of course, be passed to another function, and the value changed, but the called function cannot access the variable using the original name.) A function parameter can have the same name as a global variable or function, in which case the global variable or function cannot be referenced from the function. For example, the code fragment

```
int count;

void test(float count)
{
    /* statements go here */
}
```

is legal. Any function declared after the global declaration of `count` can use or modify `count`, but that variable cannot be used or modified from within the function `test`. Inside `test`, `count` refers to the float parameter.

Variables defined at the top of the function body are also available anywhere in the function, and are not available from outside of the function. Variables defined at the top of the function body cannot duplicate the names of parameters, but they can duplicate global variables or functions.

```

int count;

void test(void)
{
    double count;
    ...
}

```

Within the function, references to count use the double variable.

Any compound statement within a function can declare variables whose scope is limited to the duration of the compound statement. These variables cannot be accessed from outside the compound statement, although they can be used from compound statements embedded in the one where the variable is declared. Variables defined within a compound statement can reuse the names of global variables or functions, parameters, or variables defined in the program body or other compound statements. For example, the following function will print 1, 2 and 1 to standard out.

```

void print(void)
{
    int i;

    i = 1;
    printf("%d\n", i);
    {
        int i;

        i = 2;
        printf("%d\n", i);
    }
    printf("%d\n", i);
}

```

Preprocessor macros are available from the point they are defined to the end of the source file or to the undef command that removes the macro definition. Note that since macro expansion conceptually occurs before the program is compiled, once a macro is defined, and occurrence of the macro name will be replaced by the macro body, even if that occurrence appears to be defining a new variable. For example,

```

#define name sally
int name;

```

defines a new variable called sally, not a variable called name.

Labels in a function are available throughout the function in which the label appears.

There are several cases in C where a name can be duplicated within the same function, or in the global declaration area. The ability to do this is called overloading. The reason is that C maintains several different tables of symbols. The overloading classes are:

1. Preprocessor macro names
2. Statement labels
3. Structure, union and enumeration tags
4. Components of structures or unions
5. Other names (variables, functions, typedef names, enumeration constants)

What this means is that the same name can be reused within a function, so long as it is not used more than once within the same overloading class. Another important point is that each structure

and union has its own overloading class; thus, it is legal to define two structures or unions, and to use the same component names in the various structures and unions. This is, in fact, one of the few places where it is reasonable to take advantage of the overloading classes. For example, if you are defining more than one kind of linked list, it would be very reasonable to name the pointer to the next structure next in each of the structures. The following example shows each of these rules in effect.

```
void test(void)

{
#define name gorp
int name;
enum name {Fred, Joe};
struct s {int name};
union u {int name};
goto name;
name: ;
}
```


Chapter 10 – Functions

Declaring a Function

```
function-declarator:
    declarator
    '('
    [parameter-type-list {',' ' ' '.' ' ' '.' ' ' } |
    {parameter-list} ]
    ')'
    {inline '(' [long-integer ' ' ] long-integer ')'}
asm-function-declaration: asm identifier '{' {asm-line}* '}'
```

C function declarations can be intermixed with other declarations throughout the program. The only restriction is that a C function cannot be defined within the body of another function.

A function declaration looks very much like the declaration of any variable. The two major differences are that functions do not have to have a type specifier, and the type is always function returning something. When a function is declared without a type specifier, a type specifier of `int` is assumed. For example, the following two declarations are almost identical, but the first defines a function returning a pointer to `int`, while the second defines a pointer to a function returning `int`.

```
int *f();
int (*f)();
```

It is also possible to declare a function automatically. This happens when a function that has not been declared is used in an expression, as in

```
x = test();
```

When a function is declared automatically, it is assumed to be a function returning `int`. Any later declarations or definitions must also be for a function returning `int`. To avoid problems, and to make effective use of function prototypes, it is a good idea to declare all functions before they are used.

Functions can be defined to return any type except a function or array. Functions can be defined to return pointers to other functions or arrays, however.

There is a fine distinction between a function declaration and a function definition. A function declaration tells the compiler that a function exists, either later in the same source file, in a separately compiled or assembled module, or in a library, but does not include the statements that tell the compiler what the function is supposed to do. A function definition includes a function body, which is a compound statement. To declare a function, follow the declaration with a semicolon, as shown above. To define a function, follow the declaration with the function body, like this:

```
void greet(void)

{
    printf("Hello, world.\n");
}
```

Note that there is no semicolon after the closing parenthesis in the first line. It is an error to place one there.

Whenever a function is declared in a program, the compiler assumes that the definition occurs elsewhere. If the definition is not within the current program segment, you must use the extern storage class specifier, as in

```
extern void greet(void);
```

The storage class extern can also be used when the function definition will appear later in the same source file. The storage class static can also be used to declare a function that will be defined later in the same source file. The difference between the two is that, with the storage class static, the function must appear in the same source file, and static functions are not available outside the source file, while functions with a storage class of extern can be called from separately compiled modules.

The default storage class for a function is extern.

ORCA/C supports two special kinds of function definitions. The first is an inline definition, used to create header files for functions at a fixed address, such as the Apple IIGS tools. An inline declaration replaces the function body with the word inline, followed by one or two integer constants enclosed in parentheses and separated by a comma.

The inline directive is used one of two ways. When writing Apple IIGS toolbox header files, the inline directive is always coded with two integers and the pascal qualifier, as in this declaration from QuickDraw.h:

```
extern pascal void LineTo(Integer, Integer) inline(0x3C04, dispatcher);
```

In this case, the first integer in the inline directive is the tool number; it is passed in the X register when the call is made. The second number is the address to call. For normal Apple IIGS tool calls, this will be 0xE10000. The tool header files use the macro dispatcher, which is equated to 0xE10000 in types.h. By varying the second value, which is the call address, you can create header files for user tools, or for any other functions at a fixed address that use toolbox calling conventions.

The inline directive is also used to create headers for functions that use ORCA/C calling conventions, but that are located at a fixed address. In this case, the pascal qualifier is not used. While the first integer can be coded, it is ignored. As with any C function declared using ORCA/C, the X register is undefined upon entry to the function.

For example, given the declaration

```
extern int FixedFunction (int) inline(0x01ABCD);
```

the call

```
j = FixedFunction(5);
```

would call a C function at the fixed address 0x01ABCD.

Another type of function is a function written entirely in assembly language. These functions have a return type and a parameter list, just as other functions do. The compiler uses the function return type and parameter list to check function calls in the rest of the program to make sure parameters are passed correctly, and to check to make sure that the value returned by the function is used in a legal way, but it is up to you to actually write the assembly language statements that use the parameters, remove them from the stack before returning to the caller, and to return any values to the caller.

An example of an assembly language function is shown below.

```
/* See if a key has been pressed; return 128 is so, and 0 otherwise */

asm int keypress ()

{
    lda    >0xC000
    and    #0x0080
    rtl
}
```

For a description of the syntax of assembly language statements, see the description of the asm statement in Chapter 12. For details on how parameters are passed and how function values are returned, see Chapter 4.

Parameters

In addition to returning a result, functions can take inputs in the form of passed parameters. C supports two different ways of handling parameters that have little to do with one another. The historical reason for this is that older C compilers use a very simple mechanism for defining parameters, while modern C compilers, including ANSI C compilers, support a parameter passing mechanism that allows the compiler to do some compile-time checking of function calls. These parameter mechanisms will be described separately.

Traditional C Parameters

*parameter-list: identifier {',' identifier}**

The original parameter passing mechanism is very simple. Function declarations do not have parameter lists, even if the function that will be called allows or requires parameters. In a function definition, the names of the parameters are listed between the parentheses that follow the function name, separated by commas. The parameters are then defined before the start of the function body. Each parameter that appears in the parameter list must be defined, and, while types, structures, unions and enumerations may be defined, the only variables that can appear are those listed in the parameter list. As an example, the following function accepts an integer and a pointer to a string as parameters.

```
void roman(numeral, digit)

char *numeral;
int digit;

{
    /* statements go here */
}
```

For each parameter there must be exactly one variable declaration between the function declaration statement and the body of the function. No other variables can be defined in this area, although type declarations are allowed. Parameters may not be initialized, and the only storage class that can be used is register. Parameters, like variables, can be any type except void or a function.

Because function declarations do not allow parameter declarations, traditional C parameters cannot be checked for correctness by the compiler. For example, if the function shown above is called using a statement like

```
roman(4);
```

there is obviously a problem: the function definition expects two parameters, while the call is only passing one.

Function Prototypes

```
parameter-type-list: parameter-declaration {',' parameter-declaration}*
parameter-declaration: declaration-specifiers
                      [declarator abstract-declarator]
abstract-declarator: {non-empty-abstract-declarator}
non-empty-abstract-declarator: ['(' non-empty-abstract-declarator ')'] |
                      [abstract-declarator '(' ')'] | [abstract-declarator '[' {expression}
                      ']' ] | ['*' abstract-declarator]
```

Function prototypes correct the deficiencies in C described in the last section. They give the compiler the ability to check a function call to ensure that the parameters passed by a function call match the parameter list expected by the corresponding function. With a function prototype, each parameter is specified as a variable declaration, rather than simply a name. The function prototype can be used in both the declaration and definition of a function. Once a function has been declared or defined using a function prototype, the compiler checks subsequent calls to the function, flagging any calls that pass a parameter list that the function cannot handle as an error.

For an example, we will repeat the example from the last section using function prototypes.

```
void roman(char *numeral, int digit)

{
/* statements go here */
}
```

This simple example shows that the major difference in the way a function is defined using function prototypes is that the variable declarations are moved into the parameter list. Once the function has been declared or defined, however, the call

```
roman(4)
```

would cause the compiler to flag an error, rather than silently producing a program that might crash.

In addition to detecting parameter lists that have too few, too many, or incorrect types of parameters, function prototypes allow the compiler to do type conversions. With a traditional parameter list, for example, if you define a function that expects a long integer, and call it with an integer parameter, the result can be as severe as a run-time crash, and will rarely give a correct answer, even if the program does not crash. If the function has been declared using function prototypes, however, the integer parameter is converted to a long integer. All conversions that are performed during assignment using the = operator will also be performed for parameters passed to a function that has been declared using function prototypes.

A special case arises when a function has no parameters. The parameter list should include the single word void, indicating that no parameters are allowed. Any call to the function that tries to pass a parameter will then be flagged as an error.

```
extern void MakeMyDay(void); /* prototype function with no parameters */
```

It is possible to use an abstract declarator in a function prototype. Basically, an abstract declarator defines a type without giving a variable name. This form of declaration is usually restricted to parameter lists for function declarations, rather than function definitions. An abstract declarator would be of no use in a function definition, since the parameter would have no name, and thus could not be referenced in the function body. In a function declaration, however, it allows you to tell the compiler about the parameter list without specifying the names of the parameters.

There is an important restriction that applies when using function prototypes. Functions must be declared before they are used. If a function is used before it is defined, the compiler sets up a default declaration. This declaration assumes that the function is not prototyped. Later, when the function is defined with a prototyped parameter list, a conflict arises, and the compiler flags an error.

Variable Length Parameter Lists

Many C functions, most notably those in the standard input and output library, use variable length parameter lists. For example, when you call `printf`, you always supply a format string, but you can also supply additional parameters. ANSI C introduced a mechanism to handle variable length parameter lists entirely from C, although the method requires you to use a function prototype.

Basically, the parameter list is split into a fixed part and a variable part. The fixed part is required: at least one fixed variable must be present. The variable part is represented in the function prototype by three periods, and must appear at the end of the parameter list. For example, to declare a function that will add one or more integers, returning the sum as a result, we would use

```
int sum(int first,...)
```

In this case, `first` is the fixed parameter.

The functions `va_start`, `va_arg` and `va_end`, from the `stdarg.h` library, provide a way of using the variable length parameter list from C. For examples of their use, see the description of `va_arg` in Chapter 13.

Variable argument lists will not work if stack repair code is enabled; stack repair code is enabled by default. For an explanation of stack repair code, see the next section. To turn off stack repair code, see the `optimize` pragma.

Common Mistakes With Parameters

Probably the most pervasive programming error in C programs is misuse of parameter lists. This isn't helped by the fact that many compilers seem to work with some kinds of parameter errors, and as a result, some programmers have written supposedly portable C programs that are incorrect. In all C standards, from the original Kerninghan and Ritchie specification right through to the most recent ANSI C standard, passing the wrong number or wrong type of parameters to a function gives, as the standards put it, undefined results. Basically, that means that a C compiler can support that practice if it chooses, but programs written that way are not portable to other C compilers; it is perfectly legal for a C compiler to simply ignore the possibility that the error can occur, resulting in a program that could actually crash. Passing a different number of parameters than were expected by the function being called was one early way to handle variable argument lists, which helped encourage the practice, but it is not supported by all C compilers.

To make this work, C compilers that support the practice of allowing you to call a function with the wrong number or type of parameters have the caller remove parameters from the stack after control returns from the function. In effect, this patches the stack, removing parameters whether or not the function that was called knew they were on the stack. On the 65816 CPU used in the Apple IIGS, it is a little more efficient for the function to remove its own parameters, and that is how ORCA/C works. As a result, if you call a function with the wrong number or type of parameters, you risk crashing the computer.

On the one hand, ORCA/C is a perfectly correct implementation of C, since passing incorrect parameters is not a feature that C compilers must support, but on the other hand, there are a lot of programs that assume this is legal C, and parameter errors are unfortunately easy to make in C. For that reason, by default, ORCA/C installs stack repair code that makes sure extra parameters are removed from the stack. This stack repair code takes up a lot of room and slows execution speed considerably, though, so ORCA/C also has a way to turn off the stack repair code. For more information on turning off the stack repair code, see the optimize pragma. Finally, to help you track down this sort of error, ORCA/C also has a debug option that will tell you when the parameter list is the wrong size; you can find a description of this feature under the debug pragma.

How Parameters are Passed

Integers, floating-point variables, and pointers are always passed by value. C does not have a mechanism for passing one of these types by reference like Pascal or Ada. This means that a function cannot change the original value of a variable passed as a parameter. For example, the program

```
void change(int i)
{
    ++i;
    printf("%d\n", i);
}

int main(void)
{
    int i;

    i = 1;
    printf("%d\n", i);
    change(i);
    printf("%d\n", i);
}
```

does not change the value of *i* in main. The program will print 1, 2 and 1 to the screen, not 1, 2 and 2.

This does not mean that there is no way to create a function that can modify a variable in another function, just that there is no way to do it directly. When a function is supposed to change the original value of a variable, a pointer to the variable is passed, rather than the variable itself. Using this idea in the example just presented, we can create a program that will print 1, 2 and 2:

```

void change(int *i)
{
    *i += 1;
    printf("%d\n", *i);
}

int main(void)
{
    int i;

    i = 1;
    printf("%d\n", i);
    change(&i);
    printf("%d\n", i);
}

```

Arrays, structures and unions, on the other hand, are always passed by reference. In all three cases, the address of the first byte of the structure is passed, not the actual bytes that make up the structure. If the function that is called makes any change to the array, structure or union, the change affects the copy passed to the function. The following example illustrates this by using a function to clear an array.

```

void clear(float matrix[10][10]);

{
    int i,j;

    for (i = 0; i < 10; ++i)
        for (j = 0; j < 10; ++j)
            matrix[i][j] = 0.0;
}

int main(void)
{
    float a[10][10];

    clear(a);
    /* a is now all zeros */
}

```

ORCA/C passes parameters starting with the rightmost parameter, and working to the left. The expressions are also evaluated in that order. While there is no strict requirement that parameters be processed this way in C, most C compilers follow the same practice. Simple variables of type char, short, int, long, float, double and extended are all passed by value, placing the actual value on the stack. The most significant byte is always placed on the stack first; since the 65816 stack builds from the top of memory towards the bottom, this means that the values appear least significant byte first in memory. Pointers are passed as unsigned long values, and will always have a range of 0 to 0x00FFFFFF. Arrays, structures and unions are passed by placing the address of the first byte of the array, structure or union on the stack. The compiler expects that all parameters are removed from the stack by the function called.

Returning Values from Functions

Functions can return any type except an array or function, although they can return a pointer to an array of function. The value, if any, is returned as an expression on a return statement. Functions returning void should use the return statement without an expression. If function is declared as returning void, any expression in the return statement will be flagged as an error.

Pascal Functions

C functions have two major differences from functions and procedures in other languages. First, most C compilers push the parameters to a function onto the stack working from right to left. In all other common languages, parameters are generally passed starting with the leftmost parameter and working to the right. Second, C is a case sensitive language, so that the functions foo and Foo are different. All other common languages are case insensitive, and would flag an error if you attempt to create two functions whose names differ only in the case of the letters used. The pascal qualifier allows you to create C functions that can be called from other languages, or to call procedures and functions defined in languages other than C.

To use the pascal qualifier, place the word pascal immediately after the storage class specifier (if any), and before the function type. For example,

```
extern pascal int sum(int a, int b);
```

could be used to allow C to call a function named sum that requires two integer parameters and returns an integer result. Because the pascal qualifier has been used, the integer a is push on the stack before the integer b, exactly the opposite of the normal order. This function could be written in any language – including C, so long as the function definition also specifies the pascal qualifier.

The C compiler still treats the names of functions using the pascal qualifier as case sensitive, but the names are converted to uppercase characters before passing them on to the linker. This preserves the feel of C, but satisfies the requirements of other languages. It does create one problem, however: it is possible to define two functions in C whose names differ only in the case of the letters used, and end up with a linker error, while the compiler does not see a problem. If this happens, you must choose names that have differences other than the case of the letters.

Chapter 11 – Expressions

Syntax

In other chapters about the compiler, the syntax charts for the language is intermixed with the description of the C language. This doesn't work well for expressions, which can be explained more easily in terms of the operators and operands that make up the expression. This section presents the syntax charts for expressions for completeness.

```
expression: comma-expression
comma-expression: assignment-expression {',' assignment-expression}
assignment-expression:
    conditional-expression |
    [unary-expression
    [
    '=' | '+=' | '-=' | '*=' | '/=' | '%=' | '<=' | '>=' | '^=' | '='
    ]
    assignment-expression]
unary-expression: postfix-expression
    | cast-expression
    | sizeof-expression
    | ['- ' | '+ ' | '! ' | '~ ' | '&' | '*' | '-- ' | '++ '] unary-expression
postfix-expression:
    identifier |
    constant |
    ['(' expression ')']
    {['[ ' expression ']' | ['. ' identifier] | ['->' identifier]
    | ['(' {comma-expression} ')'] | '++ ' | '-- ']*
cast-expression: '(' type-name ')' unary-expression
type-name: type-specifier abstract-declarator
sizeof-expression: sizeof ['(' type-name ')'] | unary-expression
conditional-expression:
    logical-or-expression {'?' expression ':' conditional-expression}
logical-or-expression:
    logical-and-expression {'||' logical-and-expression}*
logical-and-expression:
    bitwise-or-expression {'&&' bitwise-or-expression}*
bitwise-or-expression:
    bitwise-xor-expression {'|' bitwise-xor-expression}*
bitwise-xor-expression:
    bitwise-and-expression {'^' bitwise-and-expression}*
bitwise-and-expression: equality-expression {'&' equality-expression}*
equality-expression:
    relational-expression {'==' | '!=' relational-expression}*
relational-expression:
    shift-expression {'<' | '<=' | '>' | '>=' shift-expression}*
shift-expression:
    additive-expression {'<<' | '>>' additive-expression}*
additive-expression:
    multiplicative-expression {'+' | '-' multiplicative-expression}*
multiplicative-expression:
    unary-expression {'*' | '%' | '/' unary-expression}*

```

Operator Precedence

C has a rich variety of operators, as well as a complicated set of operator precedences. Operator precedence is what forces the value of the expression $1+2*3$ to be 7, rather than 9, which would be the result if the expression were evaluated left to right. The table below shows the precedence of all of the operators. The operators with the highest precedence are shown above the operators with lower precedence.

```

terms
++1  --1
++2  --2  sizeof  (...)3~  !  _4  +4  &5  *6
*  /  %
+  -
<<  >>
<  >  <=  >=
==  !=
&7
^
|
&&
||
?:8
=  +=  -=  *=  /=  %=  <<=  >>=  &=  ^=  |=
,

```

- 1 postfix operators
- 2 prefix operators
- 3 type casts
- 4 unary addition, subtraction
- 5 address operator
- 6 indirection operator
- 7 bitwise and
- 8 conditional evaluation operator (a ternary operator)

Terms

Expressions are made up of terms separated by operators. The terms represent the variables and constants to be worked on, or the locations where these variables will be stored. This includes elements of arrays and components of structures and unions. Terms also include values returned by functions.

L-values

In discussing the various forms that a term can take, some will be referred to as l-values. This is an important concept in any language, but is much more important in C, with its multiple assignment operators and operators with side effects. Conceptually, an l-value is any simple value that can be changed. L-values include arithmetic variables, elements of arrays, structures, unions,

components of structures and unions, enumeration variables and pointers. L-values do not include entire arrays, functions, or constants.

Constants

The simplest type of term is the constant. Constants can appear in base ten, base eight (octal) or base sixteen (hexadecimal). Constants can be coded as integers, real numbers, or strings, with a variety of attributes. Constants are not l-values. Constants are covered fully in Chapter 5.

Simple Variables

Another simple type of term is the variable. The simple variable can have a variety of types, but always appears as a name in the expression. The variable must be defined before it is used. Variables come in all of the same types that constants come in, except for strings: a string variable is actually an array of characters. Variables are l-values. All operators have some restrictions on the type of variables they can be used with.

Arrays

Arrays can appear in expressions in one of two ways, with or without a subscript. When an array appears without a subscript, it is not an l-value. It can appear in several places in this form. When an array appears in the operand of the sizeof operator, the size of the entire array is returned. For example,

```
int a[10];

...

i = sizeof(a);
```

would set *i* to twenty, since the array *a* consists of ten integers, each of which is two bytes long. In other situations, the usual unary or binary conversions are performed. (The usual conversions are discussed in the section "Automatic Type Conversions," later in this chapter.) When the conversions are performed, the array is converted to a pointer to the first element of the array, and the type of the result is a pointer to an element of the array. For example,

```
int a[10], *ip;

...

ip = a;
```

sets *ip* to point to *a*[0].

An array element is accessed by coding the name of the array followed by a left bracket, an expression, and a right bracket. The expression is evaluated. The result type must be integral; it is used to index into the array. The result type of the entire term is the same as the type of one element of the array. The type of the term is an l-value if one term of the array is an l-value, and it is not an l-value if one element of the array is not an l-value. For example, with the definition

```
int a[10][10];
```

`a[1]` is not an l-value, since it refers to a ten-element array of integers. `A[1][3]`, on the other hand, is an l-value: the type of the term is `int`.

The subscript operator can also be applied to a pointer. Just as array names are treated as pointers to the first element of an array, a subscript operator applied to a pointer treats the pointer as the address of the first element of an array whose elements have the same type as the base type of the pointer. For example, the following statements show four ways to fill an array of ten integers with the numbers one through ten. The second method shows indexing of a pointer. Note the variety of other ways made possible by the close relationship between pointers and arrays in C.

```
int a[10], i, *ip;

/* method 1: subscripting the array */
for (i = 0; i < 10; ++i)
    a[i] = i+1;

/* method 2: subscripting the pointer */
ip = a;
for (i = 0; i < 10; ++i)
    ip[i] = i+1;

/* method 3: dereferencing the pointer */
for (i = 0; i < 10; ++i)
    *(ip+i) = i+1;

/* method 4: dereferencing the array */
for (i = 0; i < 10; ++i)
    *(a+i) = i+1;
```

Function Calls

A function call is coded as the name of a function followed by a pair of parentheses. If the function requires (or allows) parameters, the parameters are coded as expressions separated by commas, and appear between the parentheses. The parentheses are required even if the function has no parameters. The result type is the type returned by the function. It is not an l-value. If the function returns void, the result may not be used as an operand for any operator described in this chapter; this is equivalent to the procedure call or subroutine call in other languages.

If a function appears in an expression without the parentheses that indicate a parameter list, it is treated as a pointer to the function. This property of functions is often used when assigning a value to a pointer to a function, or when passing a function as a parameter. To call a function when a pointer to the function is available, code the pointer as if it were a function call. The following program uses these principles to print both the sine and cosine of $\pi/4$.

```

#define pi 3.1415926535

void print(float (*f)(), float val)
{
    printf("%f\n", f(val));
}

int main(void)
{
    print(sin, pi/4.0);
    print(cos, pi/4);
}

```

Actual parameters to functions are evaluated starting with the rightmost parameter and proceeding to the left. The order in which parameters are evaluated varies from compiler to compiler. Since the order of evaluation can be critical to the outcome of functions with side effects, it is best not to use such functions in your programs. The following example illustrates this. It can also be used to test the order of evaluation used by a particular compiler.

```

int val;

int change(void)
{
    return val++;
}

int test(int a, int b)
{
    return a+b;
}

int main(void)
{
    val = 1;

    if (test(2*change(), change()) == 4)
        printf("Arguments are evaluated left-to-right.\n");
    else
        printf("Arguments are evaluated right-to-left.\n");
}

```

Component Selection

The direct selection operator allows selection of a field from within a structure or union. It appears as a period separating an expression whose result is a structure or union (on the left) from the name of a component of the structure or union (on the right). The result has the same type as the field of the structure or union. It is an l-value if the expression to the left of the period is an l-value, and the component is not an array. The expression to the left of the period is not an l-value if the structure or union is returned by a function.

```

struct point {float x,y,z};
struct polygon {point a,b,c,d;} p;

p.a.x = 1.0;    /* set the x-coordinate of point a in polygon p to 1 */

```

The indirect selection operator, `->`, is similar to the direct selection operator in that it is used to access components of structures and unions. The difference is that the left side is a pointer to a structure or union, rather than a structure or union. Once again, the result has the same type as the field of the structure or union. The result is an l-value if the component is not an array.

```

struct polygon *pPtr;

pPtr->a.x = 1.0;    /* set the x-coordinate of point a 1 */

```

The indirect selection operator is completely equivalent to the direct selection operator applied to the pointer after it has been dereferenced. For example, the above example could be restated as

```
(*pPtr).a.x = 1.0;
```

and the result would not change.

Parenthesized Expressions

A parenthesized expression is an expression enclosed in parentheses. The parentheses do not affect the status of the enclosed expression in any way. In particular, the type of the parenthesized expression is the same as the type of the enclosed expression, and it is an l-value if and only if the enclosed expression is an l-value. The sole effect of the parentheses is to modify the precedence of the operators. The expression within the parentheses is evaluated before the surrounding operators are applied.

Postincrement and Postdecrement

The postincrement and postdecrement operators are used to increment or decrement scalar values. The operand must be an l-value. The result of the expression is the value before the operator is applied. For example,

```

i = 1;
j = i++;
printf("%d, %d\n", j, i);

```

prints 1, 2 to standard out, not 2, 2.

The result is not an l-value. The result type is the type of the operand.

If the `++` or `--` operator is applied to a pointer, the updated pointer points to one object beyond (`++`) or before (`--`) the original value. The example below illustrates this by filling an array, then printing the contents backwards.

```

int i, a[10], *ip;

ip = a;
for (i = 1; i < 11; ++i)
    ip++ = i;

ip--;
for (i = 1; i < 11; ++i)
    printf("%d\n", ip--);

```

The result is technically undefined if an overflow or underflow results. In ORCA/C, if unsigned numbers are used, the result wraps around zero. For example, using unsigned integers, incrementing an integer whose value is 65535 would result in zero, and decrementing zero would yield 65535. For the case of signed integers, incrementing 32767 would give -32768, while decrementing -32768 would give 32767.

Math Operators

C supports a total of ten operators for dealing with numeric values. Five of these are binary operators, four are unary operators, and one is a built-in function. The operations, and the symbols used to represent the operations, are shown below.

operation	symbol	type	operands
addition	+	binary	any arithmetic type or pointer
subtraction	-	binary	any arithmetic type or pointer
multiplication	*	binary	any arithmetic type
division	/	binary	any arithmetic type
remainder	%	binary	any integer type
unary addition	+	unary	any arithmetic type
unary subtraction	-	unary	any arithmetic type
increment	++	unary	any arithmetic type or pointer
decrement	--	unary	any arithmetic type or pointer
size	sizeof()	function	any type or unary expression

Integer types include char, short, int, long, and the unsigned forms of all of these. Arithmetic types include the integer types plus float, double, comp and extended.

Addition

The operands are converted to the same type using the usual binary conversion rules. The two operands are then added. The result is the same type as the converted operands, and is not an l-value.

Integer overflow is not detected. If two integers exceed the range of the type of the operands, the extra most-significant bits are lost.

Floating-point errors are not detected automatically. Errors can be detected by direct calls to SANE. If a floating-point underflow occurs, the result is zero. Floating-point overflow gives a result of infinity, which can lead to correct, non-infinite answers in some forms of equations.

The addition operator can also be used to add an integer to a pointer. The result is a pointer of the same type as the input pointer. If, for example, n is added to a pointer, the new pointer points n elements past the original pointer. If ip is a pointer to an integer, then, *(ip+2) load the integer that is two integers past the integer pointed to by ip.

Subtraction

The operands are converted to the same type using the usual binary conversion rules. The second operand is then subtracted from the first. The result is the same type as the converted operands, and is not an l-value.

Integer overflow is not detected. If two integers exceed the range of the type of the operands, the extra most-significant bits are lost.

If the operands are one of the unsigned integer types, and the left operand is smaller than the right operand, the result is a positive unsigned number wrapped through the given base. For example, if the operands are unsigned int, and the left operand has a value of 1, while the right has a value of 2, then the result is 65536-1, or 65535. Another way of thinking about this is to recognize that this result has the bit pattern of the signed result of the operation, but that it is represented as an unsigned number.

Floating-point errors are not detected automatically. Errors can be detected by direct calls to SANE. If a floating-point underflow occurs, the result is zero. Floating-point overflow gives a result of infinity, which can lead to correct, non-infinite answers in some forms of equations.

The subtraction operator can also be used with two pointer operators if the pointers are of the same type. The result is an integer; it is the number of elements between the two pointers. For example, `(&a[4])-(&a[1])` would be 3, regardless of the type of the array.

Multiplication

The operands are converted to the same type using the usual binary conversion rules. The two operands are then multiplied. The result is the same type as the converted operands, and is not an l-value.

Integer overflow is not detected. If two integers exceed the range of the type of the operands, the extra most-significant bits are lost.

Floating-point errors are not detected automatically. Errors can be detected by direct calls to SANE. If a floating-point underflow occurs, the result is zero. Floating-point overflow gives a result of infinity, which can lead to correct, non-infinite answers in some forms of equations.

Division

The operands are converted to the same type using the usual binary conversion rules. The first operand (the numerator) is then divided by the second operand (the denominator). The result is the same type as the converted operands, and is not an l-value.

For integer division, any fractional part of the result is discarded; i.e., $4/3$ gives a result of 1, not 1.333. For positive results, all C compilers return the truncated number; i.e., the largest integer that is less than or equal to the floating-point result. Various compilers truncate negative numbers in different ways, so it is not a good idea to depend on the results of integer division when the results are negative. In ORCA/C, as in most implementations of C, truncation of a negative result returns the integer closest to zero. For example, $(-4)/3$ gives a result of -1, not -2.

Integer division by zero is not detected. The results of dividing by zero are not predictable.

For floating-point division, the result is the floating-point number that is the best representation of the correct answer; here, some differences between the actual result and the correct mathematical result may occur due to round-off error.

Floating-point errors are not detected automatically. Errors can be detected by direct calls to SANE. If a floating-point underflow occurs, the result is zero. Floating-point overflow gives a result of infinity, which can lead to correct, non-infinite answers in some forms of equations.

Remainder

The operands are converted to the same type using the usual binary conversion rules. The first operand is then divided by the second. The result is the remainder from the division. The result is the same type as the converted operands, and is not an l-value.

For non-zero b , the following relation always holds. It defines the action of this operation in a mathematical sense, including the results when one argument is negative and the other is positive.

$$((a/b)*b + a\%b) == a$$

Integer division by zero is not detected. The results of dividing by zero are not predictable.

The remainder function requires integer operands; it is an error to supply a floating-point value as one or both of the operands.

Unary Subtraction

The operand, which can be of any arithmetic type, is converted using the usual unary conversion rules. The result is not an l-value. The operation is completely equivalent to subtracting the value from zero. The same rules that would apply to subtracting the number from zero also apply to unary subtraction when handling overflows, underflows, and dealing with unsigned operands.

Unary Addition

The operand, which can be of any arithmetic type, is converted using the usual unary conversion rules. The result is not an l-value. The operation is completely equivalent to adding the value to zero. The same rules apply for handling overflows, underflows, and dealing with unsigned operands. Other than any type conversions performed, this operation does not actually generate any code, since adding a number to zero does not change the number.

Prefix Increment

The operand, which can be an l-value, is incremented by one. The usual binary conversions are applied to the operand and to the constant one. The result is stored back in the operand after the usual assignment conversions are applied. The result is the new operand, and is not an l-value. The type of the result is the same as the type of the operand.

If the operand is a pointer, the pointer is moved so that it points to the next item of the type pointed to by the pointer. For example, if the pointer is a pointer to an integer, a value of two is added to the ordinal value of the pointer, since integers are two bytes long.

If the argument is an unsigned value, and the value of the argument is the largest unsigned number that can be represented with the given integer size, the result is zero. If the argument is a signed integer and the value is the largest signed integer, the result is technically undefined, but is actually one less than the value of the argument subtracted from zero in ORCA/C. For example, if the value of the signed integer i is 32767 (the largest signed integer), and the $++$ operator is applied to the integer, the result is -32768.

Prefix Decrement

The operand, which can be an scalar l-value, is decremented by one. The usual binary conversions are applied to the operand and to the constant one. The result is stored back in the operand after the usual assignment conversions are applied. The result is the new operand, and is not an l-value. The type of the result is the same as the type of the operand.

If the operand is a pointer, the pointer is moved so that it points to the previous item of the type pointed to by the pointer. For example, if the pointer is a pointer to an integer, a value of two is subtracted from the ordinal value of the pointer, since integers are two bytes long.

If the argument is an unsigned value, and the value of the argument is zero, the result is the largest unsigned number that can be represented with the given integer size. If the argument is a signed integer and the value is the smallest signed integer, the result is technically undefined, but is actually one less than the value of the argument subtracted from zero in ORCA/C. For example, if the value of the signed integer *i* is -32768 (the smallest signed integer that can be represented in two bytes using two's complement notation), and the `--` operator is applied to the integer, the result is 32767.

Sizeof Operator

The `sizeof` operator returns the size of the operand, in bytes. The result is not an l-value. The operand can be a type name or any unary expression. The operand cannot be a function, void, or an array declared without giving explicit values for all dimensions. If the operand is an array, the result is the complete size of the array. For any other unary operand, the result is the size of the result type.

There are two forms of the `sizeof` operator. When the operand is a type name, it must be enclosed in parentheses. When the operand is a unary expression, it does not have to be enclosed in parentheses, although it does no harm to use the parentheses.

When the operand is a unary expression, the expression is not evaluated at run-time. In other words, `++` and `--` operators do not change the value of the variables, function calls are not made, and assignments are not performed. The expression is examined at compile time to determine the result type, but no code is generated.

Taking the size of a bit field returns the size of the underlying type.

While the `sizeof` operator does not perform any type conversions, the expression appearing in the operand can perform type conversions.

In ORCA/C, the result of the `sizeof` operator is of type unsigned long.

The examples below illustrate some of these principles. The declarations which precede the tables are used to compute the sizes of the terms.

```
int *ip, i, a[10];
float f, fa[5][5];
struct point (float x,y,z) p;
```

<code>sizeof (int)</code>	2
<code>sizeof (char)</code>	1
<code>sizeof (unsigned)</code>	2
<code>sizeof (long)</code>	4
<code>sizeof (float)</code>	4
<code>sizeof (double)</code>	8
<code>sizeof (extended)</code>	10
<code>sizeof (comp)</code>	8
<code>sizeof (void *)</code>	4
<code>sizeof (i)</code>	2
<code>sizeof ip</code>	4

```

sizeof (*ip)      2
sizeof a          20
sizeof a[1]       2
sizeof f          4
sizeof fa         100
sizeof (point)    12
sizeof p          12
sizeof (0L)       4
sizeof (1+7)      2
sizeof (sizeof(0)) 4

```

Comparison Operations

There are six comparison operators in C. The operands of any of these operators can work on any arithmetic type, or on two pointers if both pointers are of the same type. The equality operators can also be used on a pointer and the integer constant zero. Two pointers are considered to be equal if they point to the same byte of memory, or if both pointers are NULL. A pointer is equal to the integer constant zero if the value of the pointer is NULL. Pointer *a* is less than pointer *b* if *a* points to an object stored at a smaller address than the object *b* points to. The only time this is generally of concern in a C program is when the pointers point to elements of the same array. In that case, pointer *a* is less than pointer *b* if pointer *a* points to an element with a smaller subscript than the element pointed to by *b*.

The six comparison operators are shown in the table below.

operator	condition
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
!=	not equal

For example, *a < b* is true if *a* is less than *b*, and false if it is not.

If the condition tested by the operator is true, the result is a signed integer value of one. If the condition is false, the result is zero. The result is not an l-value.

Logical Operations

There are two binary logical operators, and one unary logical operator. All of the operations accept any scalar type as an operand. The result is of type integer, and is one for a true result, and zero for false. The result is not an l-value.

The logical and operator is `&&`. The first operand is evaluated. If it is non-zero, it is treated as true, while a zero result is treated as false. If the result is false, the right operand is not evaluated at all, since the result has already been determined, and the result of the `&&` operation is zero. If the left operand is non-zero, the second operand is evaluated. If it is zero, the result is zero (false); if the second operand results in a non-zero value, the result is one (true).

The logical or operator is `||`. The first operand is evaluated. If it is non-zero, it is treated as true, while a zero result is treated as false. If the result is true, the right operand is not evaluated at all, since the result has already been determined, and the result of the `||` operation is one. If the left operand is zero, the second operand is evaluated. If it is zero, the result is zero (false); if the second operand results in a non-zero value, the result is one (true).

The logical negation operator is `!`. The single operand, which can be of any scalar type, is converted using the standard unary conversion rules. If the operand is zero, the result is one; if the operand is non-zero, the result is zero.

Bit Manipulation

C has six powerful bit manipulation operators. The operands for all of the operators can be of any integer type. The result is not an l-value.

The bitwise and operator is `&`. It is a binary operator. The two operands are converted using the standard binary conversion rules. The type of the result matches the type of the converted operands. The result is formed by performing an and of the individual bits in the two operands. For example, `0x1248 & 0x1441 == 0x1040`.

The bitwise or operator is `|`. It is a binary operator. The two operands are converted using the standard binary conversion rules. The type of the result matches the type of the converted operands. The result is formed by performing an or of the individual bits in the two operands. For example, `0x1040 | 0x0208 == 0x1248`.

The bitwise exclusive or operator is `^`. It is a binary operator. The two operands are converted using the standard binary conversion rules. The type of the result matches the type of the converted operands. The result is formed by performing an exclusive or of the individual bits in the two operands. The result of an exclusive or is 1 if one of the bits, but not both, is 1, and 0 if both bits are 0 or both bits are 1. For example, `0x1248 ^ 0x1040 == 0x0208`.

The bitwise negation operator is `~`. It is a unary operator. The single operand, which appears to the right of the operator, is converted using the standard unary conversion rules. The type of the result matches the type of the converted operand. The result is formed by performing an exclusive or of the individual bits in the operand with a second operand composed entirely of ones. The effect is to reverse each bit in the operand. For example, `~0x1248 == 0xEDB7`.

C supports two binary shift operators, `<<` and `>>`. In each case, the operands are converted separately using the standard unary conversion rules, not the binary conversion rules. The result type is the type of the converted left operand. It is formed by shifting the operand a certain number of bits to the left (for the `<<` operator) or right (the `>>` operator).

For unsigned operands, zeros are shifted in from both the left and right to fill the new bit positions. Zeros are also used to fill the bit positions created by the shift left operator (`<<`) if the left operand is signed. If the shift right operator (`>>`) is used with a signed left operand, the sign bit is replicated to fill the unused bit position. The result of these rules is that shifting a number left is mathematically equivalent to multiplying the number by two raised to the power of the second operand, providing that no overflows occur. Shifting an unsigned number to the right is equivalent to dividing the number by two raised to the power of the second operand. The same is true for signed operands unless the result would be zero; in that one case, the result of shifting is always -1.

If the right operand is zero, the result is the value of the left operand. The results are not predictable if the right operand is less than zero. Even if you determine the results by experimentation, we reserve the right to change the compiler in the future in ways that could affect such results.

Assignment Operators

Simple Assignment

C has a variety of assignment operators. The simplest is the assignment operator, `=`. The simple assignment operator evaluates the expression to the right of the operator, then assigns the result to the l-value that appears to the left of the operator. The result of the expression is the value assigned to the l-value, and has the same type as the unconverted l-value.

The simple assignment operator can be used with any arithmetic types. If the type of the expression does not match the type of the l-value, the expression is converted to the l-value's type using the same rules that would apply if the expression were to be cast to the correct type explicitly. See the discussion of type casing for details.

Structures and unions can be assigned so long as the type of the expression matches the type of the l-value exactly. The contents of the structure or union are copied to the destination structure or union. All bytes are copied, even if some involve unused bit fields, bits used to align other fields to a byte boundary, or unused bytes in a union. The number of bytes copied is equal to the number of bytes reported by the `sizeof` operator for the size of the structure or union.

Pointers can be assigned so long as the type of the expression matches the type of the l-value exactly. Some C compilers automatically cast pointer types during assignment; ANSI C does not permit automatic casting of pointer types. It is also legal to assign an integer to a pointer so long as the integer is a constant with a value of zero.

Finally, an array can be assigned to a pointer if the elements of the array are of the same type as the values pointed to by the pointer. The resulting pointer points to the first element of the array.

The assignment operator cannot be used to copy one array into another.

Compound Assignment

The compound assignment operators are a combination of simple assignment and one of the arithmetic or bitwise binary operators. The effect is to perform the operation on the l-value and the expression, assigning the result to the l-value. The result is not an l-value. It has the same type as the l-value that appears to the left of the operation.

The one difference between a compound assignment operator and an equivalent expression formed using the simple assignment operator and the binary operation is that the l-value is only evaluated one time. For example, if a function is used to compute the subscript of an array, as in

```
a[f(x)] += 3;
```

the function `f` is only called one time. The equivalent expression using the simple assignment operator is

```
a[f(x)] = a[f(x)] + 3;
```

In this case, the function `f` is called two times. While it is rare in well-written programs, there are cases where the two expressions would yield different results.

The same restrictions that apply to the operator associated with the compound assignment also apply to the compound assignment operator. Errors are handled the same way, and the same types of operands are allowed. The table below shows the compound assignment operators with the associated binary operator and the types of operands that are allowed.

assignment operator	binary operator	operands
<code>+=</code>	<code>+</code>	any arithmetic type or pointer <code>+= integer</code>
<code>-=</code>	<code>-</code>	any arithmetic type or pointer <code>-= integer</code>
<code>*=</code>	<code>*</code>	any arithmetic type
<code>/=</code>	<code>/</code>	any arithmetic type
<code>%=</code>	<code>%</code>	any integer type
<code><<=</code>	<code><<</code>	any integer type
<code>>>=</code>	<code>>></code>	any integer type
<code>&=</code>	<code>&</code>	any integer type
<code>^=</code>	<code>^</code>	any integer type
<code> =</code>	<code> </code>	any integer type

The original C language allowed the operator to appear after the `=` character. This is no longer allowed in C, and most programs have been converted to avoid this form of the compound assignment operator. A more recent change is to require the compound assignment operators to be treated as single tokens. The effect of this change is that white space may not appear between the operator and the `=` character. This is a more recent change to the language, and some programs ported from other compilers may still have white space after the operator. The compiler will identify this error when it occurs, and the program can be easily corrected.

Multiple Assignments

Unlike the other binary operators in C, all of the assignment operators are right-associative. This means that if more than one assignment operator appears in a single expression, the rightmost operation is performed first. This allows for multiple assignments in one expression. For example, `x`, `y` and `z` can all be initialized to zero at one time, as shown below.

```
x = y = z = 0.0;
```

In most other high-level languages, this same operation would require three assignment statements.

Pointers and Addresses

The address operator `&` is used to obtain a pointer to an l-value. The result is a pointer of type "pointer to value," where value is the type of the l-value. For example, the following code uses the address operator to initialize a pointer to point to an integer.

```
int i, *ip;

ip = &i;
```

The address operator can be used with register variables in ORCA/C. Some C compilers do not permit the address operator to be used with register variables, and others may generate less efficient code if the address operator is used with a register variable.

When the address operator is applied to a function, the result is of type pointer to function. When the address operator is used on an array whose elements are of type `T`, the result is of type pointer to array of `T`, or simply pointer to `T`.

The address operator cannot be used on a bit field. For example, the following use of the address operator is illegal.

```
struct {unsigned i: 8; unsigned j: 8;} bar;
char *cp;

cp = &bar.i;                /* illegal */
```

The indirection operator is used to dereference pointers. It appears as an asterisk to the left of an l-value of type pointer. If the pointer is of type "pointer to T," the result is an l-value of type T.

The pointer must point to a valid variable when it is dereferenced. If it does not, the results are unpredictable. In most cases, a random value will be loaded, but if the pointer is pointing into one of the memory mapped I/O areas, almost anything can happen, including turning disk drive motors on or off, switching displays, and so forth.

Sequential Evaluation

The sequential evaluation operator is a comma. It can be used between two expressions in many places where an expression can be used. The left expression is evaluated first. The result is discarded, and the right expression is evaluated. The result and result type are the same as the result and result type of the right expression. The result is not an l-value.

More than one comma operator can be used in a single expression, in which case the expressions are evaluated left to right, and the result and result type are determined by the last expression.

The comma operator cannot be used in parameter lists to function calls, field length expressions in structure and union declarator lists, enumeration value expressions in enumeration lists, or initializer expressions in declarations and initializers. Parentheses can be used to allow the use of the comma operator in these situations.

Conditional Expressions

C has one ternary operator which can be used to conditionally execute a statement.

```
test() ? doTrue() : doFalse();
```

The first operand can be any scalar type. It is evaluated. If the first operand is non-zero, the second expression is evaluated. If the first operand is zero, the last expression is evaluated. The result is the evaluated second or third operand, and is not an l-value.

The last two expressions can take on several forms. If they are both arithmetic expressions, the usual binary conversion rules are applied, and the result is the common type, regardless of which expression is actually evaluated. They can also be pointers to the same type, in which case the result is a pointer to the same type. They can be structures or unions if the types are the same, in which case the result is the same type as the operand. Both expressions can be of type void, in which case the result is of type void. Finally, one can be the integer constant zero, and the other can be a pointer type. In that case, the result type matches the pointer.

The conditional operator is right-associative, so the expression

```
a ? b : c ? d : e
```

is interpreted as

```
a ? b : (c ? d : e)
```

Automatic Type Conversions

Assignment Conversions

When one value is assigned to another, the expression that appears to the right of the assignment operator must be of the same type as the l-value that appears to the left of the operator.

There are several cases when the compiler can automatically perform type conversions on the right hand side to force the type to match the type of the l-value. These are:

1. The integer 0 is assigned to a pointer. In that case, the integer is converted to a long, unsigned value and cast to the appropriate pointer type before storage.
2. The l-value and expression are of different arithmetic types. (The arithmetic types include all integer and floating-point types, as well as enumerations.) In that case, the expression is converted to the type of the l-value as if the expression were explicitly cast to the correct type. See the section on type casing for details.

There are many other conversions that are performed as a normal part of evaluating an expression. These conversions are discussed in the sections that follow.

Function Argument Conversions

When parameters are passed to a function for which a prototype exists, the expressions are converted using the same rules that would apply if the expression were being assigned to a variable of the type of the parameter. If no prototype exists, or if the parameter appears as an optional parameter in a variable length parameter list, the unary conversion rules are applied to the expression.

Unary Conversion Rules

The unary conversion rules are applied to the operands of the unary operators `!`, `-`, `+`, `~` and `*`. They are also applied to the leftmost operand of the ternary `? :` operator, the operands of the bit shift operations `<<` and `>>`, and to parameters to functions when function prototypes are not used.

Unary conversions are used primarily to reduce the number of forms an operator must handle. Short integers and char variables are converted to type `int`; unsigned short and unsigned char are converted to type `unsigned`. Float and double values are converted to type `extended`. Finally, arrays and functions are converted to pointers of the appropriate type.

Binary Conversion Rules

The binary conversion rules are used to ensure that both operands of a binary operator are of the same type before the operation is performed. The binary conversion rules are best expressed as a series of rules which are applied in turn until one of the conditions specified by a rule is matched. Before these rules are applied, the unary conversion rules are applied to each operand. The binary conversion rules, in the order in which they are applied, are:

1. If either operand is not an arithmetic type, or if the operands are of the same type, no conversion is performed.
2. If one operand is of type extended, then the other operand is converted to extended.
3. If one operand is of type double, then both operands are converted to extended.
4. If one operand is of type float, then both operands are converted to extended.
5. If one operand is of type unsigned long, the other operand is converted to unsigned long.
6. If one operand is of type long, the other operand is converted to long.
7. If one operand is of type unsigned int, the other operand is converted to unsigned int.

ORCA/C will convert all expressions involving a floating-point operand to type extended. While C traditionally requires this type of conversion, and SANE and the 68881 floating-point processor both make it economical, ANSI C does not require all floating-point calculations to be performed in the longest format available.

In the cases where the internal representation of the number remains the same, no actual code is generated by the compiler. In cases where the internal representation does change, the conversions are discussed in detail in the next section, type casting.

Type Casting

A type cast takes the form of a type name in parentheses immediately before an expression. The expression is evaluated, and then the value of the expression is changed to match the type specified.

Converting Integers to Integers

When one integer is converted to another, the new value is the same as the old if it can be represented in the new type.

There are several cases where the value of the original integer cannot be represented in the new type. The first occurs when a signed integer is converted to an unsigned integer of the same size, and the original value is less than zero. In that case, the new value is $2^n + \text{val}$, where n is 8 for unsigned char; 16 for unsigned short and unsigned int; and 32 for unsigned long; and val is the original value. Another way of looking at the conversion is that the bit pattern does not change, and the signed value becomes the unsigned equivalent of the two's complement bit pattern used to represent the number.

When an unsigned integer is converted to a signed integer of the same size, no conversion is performed. If the unsigned value is too large to represent in the signed form, the result is undefined. In fact, the result becomes the negative number whose bit pattern is the same as the original unsigned number.

If a shorter value is converted to a longer one, the only case where the arithmetic value cannot be represented is if the shorter value is a signed number, and the value is less than zero, and the longer number is an unsigned number. In this case, the number is first converted to a signed value of the same size as the unsigned type, and then the above rules apply.

If a long value is converted to a shorter one, and the final value cannot be represented exactly, the extra bits are discarded.

Converting Floating-Point Values to Integers

Floating-point values are converted to integers by discarding any fraction part, then converting the resulting number to an integer. If the number is too large or too small to represent as an integer, the results are unpredictable. Any of the floating-point formats can be converted to any integer format.

Converting Pointers to Integers

In ORCA/C (but not in all implementations of C), pointers are the same size as unsigned long integers. When converting a pointer to an integer, the compiler treats the pointer as if it were an unsigned long integer. If the pointer is converted to a format other than unsigned long integer, the rules for converting an unsigned long integer to the specified format apply.

Pointers can be converted to long and unsigned long with no loss of information. In general, converting a pointer to char, int or short and then converting the result back to a pointer will result in a different, probably incorrect, pointer value.

Converting Floating-Point Values to Other Floating-Point Formats

When converting from float to double or extended, or when converting from double to extended, there is no loss of precision, regardless of the input value.

When converting from extended to double or float, or when converting from double to float, several things can happen. It is possible that the value being converted cannot be represented accurately. For example, 1.000000000001 can be represented reasonably well using double or extended, but if the double or extended number is converted to float, loss of precision will result in a float value of 1.0. It is also possible that the exponent will be too large or too small for the smaller format. If the exponent is too small, underflow results, and the new number is set to zero. For example, converting the double number 1e-300 to a float number would result in zero. If the exponent is too large, overflow occurs, and the result is infinity. For example, converting 1e300 from double to float would cause an overflow.

Converting Integers to Floating-Point Values

All of the integer formats can be converted to any of the floating-point formats. In some cases, there may be a loss of precision. For example, float variables are accurate to about seven decimal digits, so converting the long integer 1000000001 to a float value would result in 1.0e9, not 1.000000001e9.

Converting to and from Enumerations

For the purpose of conversions, enumeration constants are treated as signed decimal integer constants, and enumeration variables are treated as type int.

Converting Pointers to Pointers

Any pointer type can be converted to any other pointer type. There is no loss of precision or change in representation, so converting the pointer back to its original type will always result in the

original value. This is not true in all C compilers, so avoid making use of this principle in programs that will be ported to other machines.

Converting Integers to Pointers

The integer 0 is used to represent the null pointer. It can be converted to a pointer in all implementations of C.

ORCA/C also allows conversion of any other integer value to a pointer. To make sense on the 65816 CPU, the apparent range of the integer should be in the range 0x00000000 to 0x00FFFFFF. If it is not, using the pointer will generally result in the most significant byte being ignored. Some tools, however, may crash or access other areas of memory.

No Apple IIGS actually has all of the memory represented by these values installed. It is up to you to ensure that the integer value represents an area of memory that is safe to access. If the memory does not belong to your program, changing it will cause unpredictable results. These results could include damaging disk files or crashing the computer.

Converting Arrays to Pointers

An array name is automatically converted to a pointer to the first element of the array in all cases except when the array appears as the operand for the sizeof operator.

Converting Functions to Pointers

Except in cases where a function name is used to call a function, the function name is automatically converted to a pointer to the function.

Converting to Void

Any type can be converted to void, although the result cannot be used. The only place where this conversion is generally used is when a function that normally returns a value is called in a context where the return value is not needed. In that case, the conversion serves as a clue to the compiler that the result will not be used. This has no effect in ORCA/C programs, since the code to use a value returned by a function is not generated unless the value is actually used.

Chapter 12 – Statements

Compound Statements

```
compound-statement:  '{' {initialized-declaration ';' }* {statement}* '}'
statement:
    [expression ';' ] |
    labeled-statement |
    compound-statement |
    if-statement |
    while-statement |
    do-statement |
    for-statement |
    switch-statement |
    break-statement |
    continue-statement |
    return-statement |
    goto-statement |
    segment-statement |
    asm-statement |
    null-statement
labeled-statement:
    [identifier | [case expression] | default] ':' statement
```

Compound statements (also called blocks) are used as function bodies and to group statements together. For example, the for loop loops over a single statement. By using a compound statement, more than one statement can be executed each time the body of the loop is executed.

The compound statement has two parts. The first part is a series of declarations. If present, variables declared in the block constitute a new scope. They cannot be used or modified from outside of the compound statement; in fact, they do not exist in memory until the compound statement is entered. These variables can be initialized. If so, the initializations are only carried out if the compound statement is executed from the beginning. If control is passed to a statement in a compound statement by a goto statement or switch statement, the variables are not initialized. Since the variables in a compound statement have a scope limited to the compound statement, names that have been used globally, as parameters, or in the compound statement that the new one is embedded in can be redefined within the compound statement. If this is done, the declaration within the compound statement has precedence until control leaves the compound statement. For example, the following program is legal in C, and will print 1, 2, 1 to the screen, not 1, 2, 2.

```

int main(void)

{
    int i;

    i = 1;
    printf("%d\n", i);

    {
        int i;

        i = 2;
        printf("%d\n", i);
    }

    printf("%d\n", i);
}

```

The second part of the compound statement is a list of zero or more statements, including compound statements. Unless a statement changes the flow of control, these statements are executed one at a time until all of the statements have been executed. If the compound statement is the body of a function, control will return to the caller. If the compound statement is embedded in another compound statement, control will pass to the statement immediately after the compound statement.

It is legal to leave a compound statement using a goto, break, continue or return statement. It is also legal to use a goto or switch statement to enter a compound statement without executing some of the statements, although variables will not be initialized in that case.

Null Statements

```
null-statement:   ';'

```

Anywhere a statement can be used, a semicolon can be used. This is the null statement, which is a statement that takes no action and generates no code. It is generally used in connection with loop statements that have no statement body, such as a loop waiting for an interrupt to take place. A null statement is shown as the body of a while statement in the example below.

```
while (NoBurglar()) ;

```

Expression Statements

An expression followed by a semicolon can be used as a statement. The expression is evaluated, and the result, if any, is discarded.

While Statement

```
while-statement:  while '(' expression ') ' statement

```

The while statement consists of the reserved word while followed by an expression in parentheses and a statement. The expression must be an arithmetic type; in particular, it must be of

a type such that the comparison (expression) $\neq 0$ is legal. If the expression is non-zero, the statement is executed, and the process repeats. If the expression evaluates to zero, control passes to the statement following the while statement.

The while loop can terminate early due to the effects of a return, goto or break statement.

```
/* initialize an array */
i = 0;
while (i < 10)
    a[i++] = 0;
```

Do Statement

do-statement: do statement while '(' expression ')' ';' ;

The do statement consists of the reserved word do, a statement, the reserved word while, and an expression enclosed in parentheses. The statement is executed. Next, the expression is evaluated. The expression must be an arithmetic type; in particular, it must be of a type such that the comparison (expression) $\neq 0$ is legal. If the result is zero, control passes to the statement following the while clause that concludes the do statement. If the result of the expression is non-zero, the process repeats.

The difference between the do statement and the while statement is that the do statement always executes the statement at least one time. The while statement checks the loop condition first; if it is zero, the statement never gets executed.

The do loop can terminate early due to the effects of a return, goto or break statement.

```
do
    printf("Please press a key.\n");
while (! KeyPress());
```

For Statement

for-statement: for '(' {expression} ';' {expression} ';' {expression} ')'
statement

The for statement is designed for use when a statement must be executed for a specified number of times. Because of its design, it can be used in many other situations. It consists of the reserved word for, followed by three expressions enclosed in parentheses and separated by semicolons, and a statement.

Each of the expressions is optional. Execution of the for statement starts by evaluating the first expression, which is normally used to initialize a loop variable. The second expression is then evaluated. The expression must be an arithmetic type; in particular, it must be of a type such that the comparison (expression) $\neq 0$ is legal. If the result is non-zero, the statement is executed. The third expression, which is generally used to increment the loop counter, is then evaluated, and the process repeats starting at the evaluation of the second expression. If the second expression evaluates to zero, execution continues with the statement following the body of the for statement.

If the second statement is not coded, the compiler assumes that it always evaluates to a non-zero value.

```
/* initialize an array */
for (i = 0; i < 100; ++i)
    a[i] = i;
```

Another way to think of the for statement is that it is roughly equivalent to a while statement with the expressions placed in certain locations. The for statement

```
for (exp1; exp2; exp3) statement;
```

is equivalent to the statement

```
{
exp1;
while (exp2) {
    statement;
    exp3;
}
}
```

The one exception is the way the continue statement works. In the for statement, the continue statement jumps to exp3; in the while statement, the continue statement jumps to the end of the block, just past exp3.

The for loop can terminate early due to the effects of a goto, return or break statement in the body of the loop.

If Statement

```
if-statement:  if '(' expression ')' statement {else statement}
```

The if statement is used to conditionally execute a statement. The expression is evaluated. The expression must be an arithmetic type; in particular, it must be of a type such that the comparison (expression) != 0 is legal. If the result is non-zero, the statement following the condition is executed. If the result is zero, the statement is not executed.

The optional else clause is used to provide a second statement that will be executed if the conditional expression evaluates to zero. It is not executed if the expression yields a non-zero result.

```
for (i = 1; i <= 10; ++i)
    if (i & 1)
        printf("%d is odd.\n", i);
    else
        printf("%d is even.\n", i);
```

Goto Statement

```
goto-statement:  goto identifier ';' 
```

The goto statement is used to transfer control to another statement. The identifier is a named label. The label must appear on exactly one statement somewhere in the current function body. The next statement executed is the statement that the label appears on.

It is legal to place a label on a statement and not place a goto statement in the function that refers to that label. It is not legal, however, to try to branch to a label that does not exist, or to place two labels with the same name in the same function, whether or not a goto statement exists that refers to the duplicate label.

Switch Statement

```
switch-statement:  switch '(' expression ')'
```

The switch statement is used to choose from a list of statements. The expression, which must yield an integer value, is evaluated. The statement that follows is generally a compound statement. Conceptually, it is scanned for a case label with a value that matches the value generated by the expression. If such a value is found, control is passed to the statement following the case label. If there is no match, and there is a default label, the statement after the default label is executed. If there is no match, and there is no default label, control passes to the statement after the switch statement.

There are some restrictions on the labels in the switch statement body. The case labels must have constants that are the same type as the expression after the usual unary conversions are applied to both. For example, it is not legal to use a case label with an int constant if the expression results in a long int value. Duplicate labels are not permitted. This means that two case statements cannot have constant expressions that result in the same value, and two default labels cannot appear in the body of the statement. Case labels and the default label can only appear in the body of a switch statement.

After control is passed to one of the statements in the switch statement, execution continues as if a goto had been used to branch to the statement. For example, the following statements would print five lines on the screen. The first would have five asterisks, the second would have four, and so on. The point is that in some languages, execution would be transferred out of the switch statement when the next case label was encountered; this is not true in C.

```
for (i = 1; i <= 5; ++i) {
    switch (i) {
        case 1: printf("");
        case 2: printf("");
        case 3: printf("");
        case 4: printf("");
        case 5: printf("");
    }
    printf("\n");
}
```

As the example shows, execution falls through from the statements marked by one case label to the next group of statements. It is customary in C to use the break statement at the end of each of the groups of statements.

```
for (i = 1; i <= 5; ++i) {
    switch (i) {
        case 1: printf("");
                break;
        case 2: printf("****");
                break;
        case 3: printf("*****");
                break;
        case 4: printf("*****");
                break;
        case 5: printf("*****");
    }
    printf("\n");
}
```

The goto and return statements can also be used to leave the body of the switch statement before the last statement is executed.

Break and Continue

```
break-statement:  break ';'
continue-statement:  continue ';;'
```

The break and continue statements are used to exit a loop early or branch to the end of the loop body.

The break statement can be used in the body of a while, do, for or switch statement. When it is encountered, control is transferred to the statement after the while, do, for or switch statement. For example, break statements can be used to cause the C switch statement to work like the Pascal case statement, as shown in the section above.

The continue statement is used to branch to the end of a while, do or for loop. Conceptually, the continue statement does a goto to the end of the loop body.

The easiest way to understand the break and continue statements completely is to look at the equivalent goto statements. The following four statement models show the statements which use break and continue statements. In all of the statements except switch, there is a label called C. A continue statement in the body of the loop (called body in the examples) is equivalent to a goto to the label C. In all of the statements, there is also a label called B. A break statement in the loop body is completely equivalent to goto B.

```
while (expression) {body; C: ;} B: ;
do {body; C: ;} while (expression); B: ;
for (expression; expression; expression) {body; C: ;} B: ;
switch (expression) {body;} B: ;
```

If a break or continue statement appears within nested statements, it applies to the most recent enclosing statement. In the following code fragment, continue and break statements appear inside a switch statement, which is in turn inside a for loop. The break statement exits the switch statement. Since the continue statement has no meaning in the switch statement, it applies to the for statement.

```
for (i = 0; i < 10; ++i)
    switch (i) {
        1: 2: 3: 5: 7:
            printf("%d is prime.\n", i);
            continue;
        9:
            printf("%d is odd.\n", i);
            break;
        default:
            printf("%d is even.\n", i);
    }
```

Return Statement

```
return-statement:  return {expression} ';;'
```

The return statement returns control to the function that called the current function. If the current function is main, control is returned to the program launcher that was used to execute the program.

If the return statement is followed by an expression, that expression is evaluated. The type of the expression must be compatible with the type of the function in the same sense that an expression must be compatible with the l-value when the assignment operator is used. The value of the expression is returned to the caller as the return value of the function.

If a function has a return type, but control is returned by a return statement that does not have an expression, the returned value is unpredictable.

If the function executes to the end of the compound statement without encountering a return statement, the effect is the same as if a return statement with no expression were encountered.

Segment Statement

```
segment-statement:  segment string-constant {',' dynamic} ';' 
```

The segment statement is used to break a program into two or more load segments. It can be used with either the large or small memory model. The affect is to cause all functions defined after the segment statement to appear in a new load segment. To understand what this does, we will look at the memory models used on the Apple IIGS in detail.

In any program compiled by ORCA/C, memory is allocated for up to five different purposes. Local variables are allocated from a stack frame allocated from bank zero when the program starts to execute. Dynamic memory can also be allocated at run time by using the library functions `calloc` and `malloc`. All functions and libraries are placed in a static code segment; this segment is called the blank segment, and has no name. With the small memory model, global scalars, arrays and structures are also placed in the blank segment. If the large memory model is used, global scalars are placed in a separate segment called `~GLOBALS`, and global arrays and structures are placed in a third segment called `~ARRAYS`. Like the blank segment, `~GLOBALS` and `~ARRAYS` are static segments.

The blank segment and the `~GLOBALS` segment are each limited to 64K bytes in length; only the `~ARRAYS` segment can be larger than 64K bytes. If the program is larger than 64K bytes in length, the small memory model cannot be used with a single segment. When this happens, there are two alternatives. If the program has many large arrays, the `memorymodel` directive, described in Chapter 6, can be used to create the `~GLOBALS` and `~ARRAYS` segments. This causes the compiler to generate larger, slower code to access the arrays, however. If the program does not use a large number of arrays or structures, but has several thousand lines of C code, the problem may be that the code itself exceeds 64K bytes. Using the large memory model will not help in that case. What is needed is a way to cause some of the functions to be placed in a separate static segment. The segment statement does just that: the operand is a string constant that becomes the name of a new segment. All functions defined after the segment statement are placed in the new segment.

```
segment "parser"      /* place subroutines in the parser segment */
```

You can create as many static segments as you like. There are advantages and disadvantages to using a large number of segments. On computers with small amounts of memory, or where other programs have fragmented memory, small segments are more likely to load, since there is a better chance that a piece of memory large enough to hold the segment will be found by the loader. A program made up of several segments, however, creates a large number of inter-segment references in the relocation dictionary. These relocation records take up room on the disk, and slow down the loader.

The functions that make up a particular segment do not have to appear in the same source file, nor do they have to appear next to each other in a source file. During the link process, the linker combines all of the functions that have the same segment name into the same load segment, regardless of the order they appear in the source program.

After using the segment statement, if you wish to place a function in the blank segment, code a segment name with ten spaces.

The segment statement can also be used to create dynamic segments. A dynamic segment is not loaded until a call is made to one of the functions in the segment. Once a call is made to a function in the dynamic segment, the segment remains in memory until an explicit call is made to the loader to unload the segment. If more than one segment statement is used to create a dynamic segment, each of the segment statements must specify that the segment is dynamic. If a single function that is not dynamic appears in the load segment, the entire load segment will become a static load segment.

```
segment "initial", dynamic
```

If you have a choice between using the large memory model or the segment statement (as might be the case in a program with several large arrays and a lot of executable code, it is best to use the segment directive, rather than the large memory model, since the compiler generates smaller, faster code when the small memory model is used.

Asm Statement

```
asm-statement:  asm '{' {asm-line}* '}' ';'
asm-line:      {identifier ':'} op-code {operand} {';' comment}
```

The asm statement allows you to code assembly language statements in the body of a function. The syntax for these statements varies a bit from the syntax used by assemblers; these differences are due to the C language itself.

Each of the assembly language statements consists of an operation code. The operation codes are the standard three-letter 65816 operation code mnemonics found in standard references for the 65816. The operation code can be specified in uppercase, lowercase, or a mix of cases. They will not be described in this manual.

Each operation code may be preceded by a label. Unlike assemblers, this label is formatted like a C label. The name of the label follows the same rules as any C identifier, and is case sensitive. Like labels in C, it must be followed by a colon. The label may be used to identify branch points, data locations, or statements.

Many 65816 operations require an operand. This operand is coded after the operation code. It follows the same syntax as is used by assemblers, with a few minor exceptions. First, spaces are allowed in the operand. Second, while expressions may be used, they are limited in form. Expressions must be constant expressions or global or local variables. Global and local variable names may be followed by a + or - operator and a constant expression. Global labels are treated as absolute addresses if the small memory model is in use, and long addresses if the large memory model is in use, unless the addressing mode is modified. (Modification of addressing modes is covered below.) Local variables are treated as direct page locations. If the size of all of the local variables is larger than 255 bytes, this can cause errors in some cases. The mini-assembler does not detect this error. For that reason, it is up to the programmer to ensure that the size of local variables is kept to a minimum if they will be used as the operands of assembly language statements.

The constant expressions used in operands follow the rules for constant expressions in C. All operators and constant operands that can appear in a C constant expression are also allowed in the operand field of an assembly language statement. In particular, note that integer constants use the C syntax for hexadecimal and octal numbers, not the syntax found in most books on assembly language.

When an expression is used in any of the operands, you can force the value to be one byte (forcing direct page addressing), two bytes (forcing absolute addressing) or three bytes (forcing

long addressing). This will override the default addressing mode. To force a particular addressing mode, precede the expression with one of the characters shown in the table below.

character	addressing mode
<	direct page
	absolute
>	long absolute

The syntax for the various addressing modes is shown in the table below. For descriptions on what the addressing modes do, see any of the reference books on the 65816. For addressing modes that have the same physical format, the actual addressing mode used depends on the value of the expression and the kinds of labels used, as described earlier. For example, an operand value of 255 would trigger direct page addressing, 1000 would trigger absolute addressing, and 100000 would trigger absolute long addressing. When a register is shown, such as the A in the accumulator addressing mode, it is shown in uppercase, but may be coded in uppercase or lowercase in a program.

There is one restriction on relative branch operands that does not apply to the other instructions. Relative branches must be made to a label; you cannot code a constant value, nor can you code a label plus or minus some offset.

addressing mode	format
absolute	expression
absolute indexed by X	expression,X
absolute indexed by Y	expression,Y
absolute indexed indirect	(expression,X)
absolute indirect	(expression)
absolute indirect long	[expression]
absolute long	expression
absolute long indexed by X	expression,X
accumulator	A
block move	expression,expression
direct page	expression
direct page indexed by X	expression,X
direct page indexed by Y	expression,Y
direct page indirect	(expression)
direct page indexed indirect	(expression,X)
direct page indirect long	[expression]
direct page indirect indexed	(expression),Y
direct page indirect long indexed	[expression],Y
immediate	#expression
implied	(no operand)
relative addressing	expression
stack relative	expression,S
stack relative indirect indexed	(expression,S),Y

The operand, or operation code if there is no operand, may be followed by a semicolon. If so, this signals the start of an assembly language comment, and all characters from the semicolon to the end of the line are ignored.

Preprocessor macros are still processed within the assembly language statements, and preprocessor macros can be used in the asm statement. The syntactic rules for using the preprocessor and macros in the assembly statement are exactly the same as they are in any other location in the C program. C style comments can also appear the asm statement.

There are three directives supported by the built-in assembler. They are `dcB`, `dcW`, and `dcL`, which create a byte, word or long word (long int) variable, respectively. The operand for each of these directives is an expression which is evaluated and used to initialize the space.

The short example shown below loads a C integer value `i`, counts the bits that are set, and stores the result in the C integer variable `j`.

```
asm {  
    lda    i  
    ldx    #0  
    ldy    #16  
lb1:    lsr    a  
        bcc    lb2  
        inc  
lb2:    dey  
        bne    lb1  
        stx    j  
}
```

Chapter 13 – Libraries

Overview of the Libraries

ORCA/C comes with a powerful set of library functions. Most of these are standard libraries that are provided in any good C compiler. A few are unique to ORCA/C or to implementations of C on Apple computers. These functions have been added to make it easier to deal with the Apple IIGS toolbox.

The standard C libraries are listed in alphabetical order for easy reference. This layout is not the best for learning to use the libraries, but it makes it easy to look up a library function to see exactly how it is implemented in ORCA/C. If you are new to C, the book that you are using to learn C should give a more tutorial introduction to the libraries.

System Functions

The run-time library for ORCA/C contains a number of functions that can be manipulated directly from ORCA/C. In some cases, these are functions normally called by the C startup code, or for some other internal purpose, that perform some service you may need in an unusual circumstance. In these cases, you can declare the function as extern and call it from within your C program. In other cases, the default action may not be what you want; for example, you may want to intercept run-time errors, displaying them in a dialog or trapping them for internal handling. In these situations, you can define the function in your program, and ORCA/C will use your version rather than the one from the library.

Source code for all of these subroutines is included with ORCA/C.

SysCharErrout

```
extern pascal void SysCharErrout (char);
```

Writes a character to error out.

SysCharOut

```
extern pascal void SysCharOut (char);
```

Writes a character to standard out.

SysIOShutDown

```
extern pascal void SysIOShutDown (void);
```

Closes any files opened by SysIOStartup.

SysIOStartup

```
extern pascal void SysIOStartup (void);
```

Starts the I/O system using the files in prefixes 10, 11 and 12. If the files have already been opened, the existing open file is used, and the file is not closed when the program exits. If prefix 10 is .CONSOLE, DRead calls are used to read lines of text, allowing editing. In this case, you can use RETURN to finish a line, or ctrl-@ or command. to signal an end of file.

SysKeyAvail

```
extern pascal int SysKeyAvail (void);
```

Returns 1 (TRUE) if there is an input character available, and 0 (FALSE) if there is not.

If a character has been put back with a call to SysPutback, and has not been read by a subsequent call to SysKeyin, the result is TRUE. The rest of the discussion assumes there is no character in the putback buffer.

If input is redirected from a file, this function is equivalent to a test for end of file, returning the opposite result.

For input from .CONSOLE, if there is remaining input in the line buffer, TRUE is returned. If not, and if the Event Manager is active, the result is TRUE if there is a keypress or auto key event available, and FALSE if not. If the Event Manager is not active, the result is TRUE if bit 7 of 0x0C0000 is set, and FALSE if not. (0x0C0000 is the hardware keyboard input location.)

Note: If input is from .CONSOLE, the fact that this function returns TRUE is *not* a guarantee that a call to SysKeyin will return immediately, since SysKeyin would wait for an entire line to be typed.

SysKeyin

```
extern pascal char SysKeyin (void);
```

Reads a character from standard in. If an end of file condition occurs, (char) 0 is returned.

If input is from .CONSOLE, an entire line is read on the first call to this subroutine, and remaining characters are returned on subsequent calls until the line is exhausted; another call will then read in a new line.

SysPutback

```
extern pascal void SysPutback (char);
```

Places a character in a one-character putback buffer. This character will be the next character returned by SysKeyin, and SysKeyAvail will return TRUE until the buffer is emptied.

If another call is made to SysPutback before the first character is used, the original character is lost.

SystemEnvironmentInit

```
extern pascal void SystemEnvironmentInit (void);
```

This subroutine initializes global variables used by the compilers and their libraries. It should be called by programs that are not started in the normal way as one step in initializing the run-time environment.

SystemError

```
extern pascal void SystemError (int);
```

When a run-time error occurs, libraries call `SystemError`. By defining your own version of `SystemError`, you can intercept and handle run-time errors within your own program. You can also call `SystemError` from within your own program if your own program needs to report an error.

By default, `SystemError` calls two other library subroutines, `SystemPrintError` and `SystemErrorLocation`, to actually handle the error. If you write your own version of `SystemError`, you may want to call one or both of these subroutines for some or all of the errors.

The table below shows the various error numbers currently reported by the run-time libraries. Some of these errors are used by only one of the ORCA languages, so not all of them are actually possible from within a program written entirely in C.

Error Number	Error
1	Subrange exceeded
2	File not open
3	Read while at end of file
4	I/O error
5	Out of memory
6	EOLN while at end of file
7	Set overflow
8	Jump to undefined case statement label <i>This error cannot be recovered from!</i>
9	Integer math error
10	Real math error
11	Underflow
12	Overflow
13	Divide by zero
14	Inexact
15	Stack overflow
16	Stack error

SystemErrorLocation

```
extern pascal void SystemErrorLocation (void);
```

This subroutine is called by `SystemError` when a run-time error is reported. Normally, this subroutine prints any traceback information recorded due to the debug pragma, then shuts down the program. This subroutine can be called from within a C program to print traceback information during the debug cycle, or replaced with a different subroutine that either handles an error and recovers from it, or shuts down the system in a different way.

SystemMinStack

```
extern pascal void SystemMinStack (void);
```

This subroutine finds the start of the segment containing the return address, setting the variable `~MinStack` to this value. It should be the very first subroutine called by programs that are not started in the normal way, assuming the program owns the stack frame. `~MinStack` must be set before calling `SystemSANEInit` or before using any debug options that check for stack overflows. It can be set manually from assembly language.

SystemMMShutDown

```
extern pascal void SystemMMShutDown (void);
```

This subroutine shuts down the memory manager used by the run time libraries. It should be called just before a program exits for the last time. The memory manager is left in a restartable state after this call.

SystemPrintError

```
extern pascal void SystemPrintError (int);
```

Writes a text error message to standard out. See `SystemError` for a list of the errors that `SystemPrintError` can handle, as well as the strings it will print.

SystemQuitFlags

```
extern pascal void SystemQuitFlags (unsigned);
```

This subroutine sets the quit flags field for the GS/OS Quit call that is made to exit from a normal C program. See [Apple IIGS GS/OS Reference](#) for the allowed values for this parameter.

Note: In restartable programs, be sure to initialize this variable to 0 manually. The libraries do not normally initialize this value.

SystemQuitPath

```
extern pascal void SystemQuitPath (GSString255Ptr);
```

This subroutine sets the quit pathname field for the GS/OS Quit call that is made to exit from a normal C program. See [Apple IIGS GS/OS Reference](#) for the allowed values for the parameter.

Note: In restartable programs, be sure to initialize this variable to NULL manually. The libraries do not normally initialize this value.

SystemSANEInit

```
extern pascal void SystemSANEInit (void);
```

This subroutine is called to start SANE. Replacing it with a dummy subroutine would cause the system to skip starting SANE. Calling this subroutine from a CDA or an NDA is a quick way to start SANE, which is not normally started by ORCA/C for these kinds of programs.

This subroutine keeps track of whether SANE was initially started, starting SANE only if needed. `SystemSANEShutDown` will only shut down SANE if it was started by this subroutine.

SystemSANEShutDown

```
extern pascal void SystemSANEShutDown (void);
```

If SANE was started by an earlier call to `SystemSANEInit`, this subroutine shuts down the tool.

SystemUserID

```
extern pascal void SystemUserID (unsigned, char *);
```

This subroutine should be called right after `SystemMinStack` by programs that are not started in the normal way. The first parameter must be passed; it is the user ID for the program. The second parameter can either be a pointer to a command line string or `NULL`. If the string is a pointer to a command line string, the command line should start with an 8 character identifier naming the launcher, and be followed by the command line as a null terminated string.

Standard C Libraries

abort

See `exit`.

abs labs fabs

```
#include <stdlib.h>
int      abs(int x);
long     labs(long x);

#include <math.h>
extended fabs(extended x);
```

The function `abs` accepts an integer argument and returns the absolute value of the argument. Note that `abs` is in `stdlib.h`, not `math.h`, as with some older C compilers.

The function `labs` accepts a long integer argument and returns the absolute value of the argument. Note that `labs` is in `stdlib.h`, not `math.h`, as with some older C compilers.

The function `fabs` accepts an extended floating-point argument and returns the absolute value of the argument.

```
distance = fabs(x1-x2);
```

acos

```
#include <math.h>
extended acos(extended x);
```

The function `acos` returns the trigonometric arc cosine (inverse cosine) of the argument. The result is in radians, and lies in the range 0 to π . If the argument is less than -1.0 or greater than 1.0, `errno` is set to `EDOM`.

```
angle = acos(arg);
```

asctime

See `ctime`.

assert

```
#include <assert.h>
void assert(int v);
```

Assert is a macro generally used while a program is under development. It takes a single argument, which must be an expression that would be legal as the condition expression in an if statement. If that argument is zero, assert prints "Assertion failed: file *file*, line *number*" to standard out, where *file* is the name of the source file, and *number* is the line number within the source file. The program is then stopped by calling `exit(-1)`. If the `#pragma debug` directive has been used to enable trace backs, ORCA/C will also print the line number and source file name where the assert call was made, and a trace back showing what calls were made to arrive at that point.

The macro `NDEBUG` is used to disable the debug code generated by calls to `assert`. If `NDEBUG` is defined when the `assert.h` header file is read, no code is generated for the `assert` calls.

```
assert(parm != 0.0); /* we should never be passed a value of zero! */
```

asin

```
#include <math.h>
extended asin(extended x);
```

The `asin` function returns the trigonometric arc sine (inverse sine) of the argument. The result is in radians, and lies in the range $-\pi/2$ to $\pi/2$. If the argument is less than -1.0 or greater than 1.0, `errno` is set to `EDOM`.

```
angle = asin(arg);
```

atan

```
#include <math.h>
extended atan(extended x);
```

The `atan` function returns the trigonometric arc tangent (inverse tangent) of the argument. The result is in radians, and lies in the range $-\pi/2$ to $\pi/2$.

In some versions of C, this function is called `arctan`. ORCA/C includes `arctan` as a macro equivalent of `atan` to make it easier to port programs written under these compilers.

```
angle = atan(arg);
```

atan2

```
#include <math.h>
extended atan2(extended y, extended x);
```

The `atan2` function returns the trigonometric arc tangent (inverse tangent) of the arguments. Since the actual coordinates of a point are given, rather than the quotient of the two points (as with `atan`), this function can return results in the range $-\pi$ to π . The result is in radians, and represents the angle between the positive x axis and the point (x, y) in Cartesian coordinates. A domain error occurs if both x and y are zero, in which case `errno` is set to `EDOM`.

```
angle = atan2(y,x);
```

atexit

```
#include <stdlib.h>
int atexit(void (*func)());
```

The `atexit` function registers a function so it will be called when the program is complete, either due to a call to the `exit` function, or because a return is made from the `main` function. More than one function can be registered in this way; if so, they are called in reverse of the order in which `atexit` was called to register the functions. The functions must not require parameters, and should return `void`. The `atexit` function returns a non-zero value if the function is registered successfully, and zero if there is not enough memory to satisfy the request.

If the same function is registered more than once, ORCA/C will call the function once for each time it is registered. Other compilers may handle this situation differently.

```
void hello(void)
{
    printf("Hello, world.\n");
}

int main(void)
{
    atexit(hello);
}
```

atof atoi atol

```
#include <stdlib.h>
double atof(char *str);
int      atoi(char *str);
long     atol(char *str);
```

These functions are simpler versions of the string conversion functions `strtod` and `strtoul`. The definitions below define these functions in terms of their more powerful counterparts. For details on what strings are accepted and how errors are handled, see the description of the `strtod` function.

```
double atof(char *str)
{
    return strtod(str, (char**)NULL);
}

int atoi(char *str)
{
    return (int) strtoul(str, (char**)NULL, 10);
}

long atol(char *str)
{
    return strtoul(str, (char**)NULL, 10);
}
```

bsearch

```
#include <stdlib.h>
void *bsearch(void *key, void *base, size_t count, size_t size,
              int (*compar)(const void *ptr1, const void *str2));
```

The `bsearch` function performs a binary search. It searches the array pointed to by `base`. This array consists of `count` elements, each of which is `size` bytes long. The array must be sorted in ascending order. The parameter `key` points to a value of the same type as the elements of the array; `key` is the element to search for. The function `compar` is supplied by the program; it takes two arguments whose types match the type of `key`, and returns 0 if the arguments match, -1 if the first argument is less than the second, and 1 if the first argument is greater than the second. If the value is found, a pointer to the array element is returned; otherwise, `bsearch` returns `NULL`.

```
int CompareZip(address *addr1, address *addr2)
{
    if (addr1.zip == addr2.zip)
        return 0;
    if (addr1.zip < addr2.zip)
        return -1;
    return 1;
}

...
/* find an address with a zip code of 87114 */
addr.zip = 87114;
aPtr = bsearch(&addr, addressList, listSize, sizeof(address),
              CompareZip);
```

c2pstr p2cstr

```
#include <string.h>
char *c2pstr(char *string);
char *p2cstr(char *string);
```

These functions are used to translate between null-terminated C strings and the so-called Pascal strings, which have a leading length byte. In both cases, a pointer to the resulting string is returned. The original string is not changed in any way; the result string is built in an internal buffer. Because an internal buffer is used, subsequent calls to either of these functions can destroy old values. For that reason, it is important to copy the result to a local string buffer if one of these functions will be called before the need for the converted string has passed.

The function `c2pstr` converts a null-terminated string into a string with a leading length byte. If the null-terminated string is longer than 255 characters, a string with 255 characters is created. The string has a terminating null character following the last character, so standard C string manipulation functions can still be used on the result.

The function `p2cstr` converts a string with a leading length byte into a null-terminated string.

These functions are not standard C functions. They are included in Apple IIGS based C compilers to make it easier to deal with the toolbox, which often requires strings with a length byte.

```
/* use a null-terminated string to set a window title */
SetWTitle(strcpy(title, c2pstr(temp)), window);
```

calloc

See malloc.

ceil **floor**

```
#include <math.h>
extended ceil(extended x);
extended floor(extended x);
```

The function `ceil` accepts a floating-point argument and returns the floating-point representation of the argument, rounded up to the next higher integer. If the argument is an integer, the result is the same value.

The function `floor` is similar, except that the result is rounded down towards negative infinity.

```
x = ceil(x);
```

cfree

See `free`.

chmod

```
#include <fcntl.h>
int chmod(char *path, int mode);
```

Changes the access bits in the file. The following bit flags are supported.

0x0100	Read	enables the file for input
0x0080	Write	enables the file for output
0x1000	Delete	allows the file to be deleted
0x2000	Rename	allows the file to be renamed
0x4000	Backup	indicates that the file should be backed up
0x8000	Invisible	makes the file invisible to the Finder

The flags are added together to make up the mode field. Setting a flag enables the corresponding GS/OS bit, enabling the action described. Clearing the flags disables the action. For example, the call

```
chmod("myfile", 0x6100);
```

sets the access bits so that the file "myfile" can be read or renamed, and indicates that it should be backed up. The file cannot be written to or deleted without changing the access bits. The file is visible to the Finder.

The flags for Delete, Rename, Backup and Invisible are unique to the Apple IIGS. All of the bits in 0x0E7F are used for other purposes under UNIX. They are ignored in ORCA/C.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible. In particular, the function should not be used in programs that will be ported to other computers unless it is imbedded in conditional compilation code so that the function will only be called on the Apple IIGS version of the program.

If the call is successful, the function returns 0; otherwise, a -1 is returned and `errno` is set as indicated in the list below.

If the file does not exist, `errno` is set to `ENOENT`.

clalloc

See malloc.

clearerr

See ferror.

clock

```
#include <time.h>
#define CLK_TCK (60)
typedef unsigned long clock_t;

clock_t clock(void);
```

The clock function is used on multi-tasking systems to see how much time has been used by a program. The result is returned as the number of clock ticks since the program started. The number of clock ticks per second is defined by the macro CLK_TCK; on most systems, this time is returned in microseconds. (One microsecond is 10^{-6} seconds.)

The Apple IIGS operating system is not a multi-tasking operating system, and the Apple IIGS clock is not accurate to the microsecond time scale. ORCA/C uses the tick count returned by the Miscellaneous Tool Set GetTick call for a clock count; CLK_TCK is therefore 60. The tick count is started whenever a heartbeat interrupt handler is installed. The Event Manager installs a heartbeat interrupt handler, so the clock function can always be used from the desktop development environment. If you will be using the clock function from the text environment, you will need to ensure that the heartbeat interrupt handler is active.

```
clicks = clock();
```

close

```
#include <fcntl.h>
int close(int filds);
```

The file with the file ID filds is closed. If the file has been duplicated using the dup call, it is not closed until each of the associated file IDs have been closed.

If the call is successful, the function returns 0; otherwise, a -1 is returned and errno is set to EBADF (filds is not a valid file descriptor).

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

See also fclose.

cos

```
#include <math.h>
extended cos(extended x);
```

The cos function returns the trigonometric cosine of the argument. The argument must be supplied in radians.

```
length = cos(x)*hypotenuse;
```


cosh

```
#include <math.h>
extended cosh(extended x);
```

The cosh function returns the hyperbolic cosine of the argument. If an error occurs, errno is set to ERANGE.

```
n = cosh(x);
```

creat

```
#include <fcntl.h>
int creat(char *path, int mode);
```

Creates a new file or opens an existing one for output. If the file exists, its length is set to 0. The name of the file is path. The mode parameter is identical to the mode parameter for the chmod call. The file created is a binary file. If the file already exists, its file type is not changed.

Please note that in APW C, creat does not have a mode parameter. It does in UNIX based C implementations, so we have maintained that use here.

If the call is successful, the function returns 0; otherwise, a -1 is returned and errno is set as indicated in the list below.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

Possible errors

EACCES The file exists, and is not write enabled.
 EACCES The file does not exist, and could not be opened.
 ENOENT The pathname is null.
 EMFILE OPEN_MAX files are already open.

ctime asctime

```
#include <time.h>
typedef unsigned long time_t;

char *ctime(time_t *timeptr);
char *asctime(struct tm *ts);
```

The ctime function takes a pointer to an encoded time as input and returns a pointer to an ASCII string of the form

```
Www Mmm dd hh:mm:ss 19yy\n\0
```

where the fields are:

Field	Example	Description
Www	Mon	day of week
Mmm	Feb	month
dd	29	date
hh	16	hour (24 hour format)
mm	03	minutes
ss	57	seconds
yy	88	year

The `asctime` function creates a similar time string, but takes a pointer to a calendar time structure created by `localtime` or `gmtime` as input.

The return string is in a static buffer which is reused by each call to `ctime` or `asctime`. If you must keep a copy of the string, and subsequent calls will be made to `ctime`, be sure to save a copy of the string in a local buffer.

See also `time`, `gmtime`, `localtime`.

```
int main (void)

{
    time_t bintime;
    struct tm timestruct;

    bintime = time(NULL);
    printf("The time is %s\n", ctime(&bintime));

    timestruct = *gmtime(&bintime);
    printf("The time is %s\n", asctime(&timestruct));
}
```

difftime

```
#include <time.h>
typedef unsigned long time_t;

double difftime(time_t t1, time_t t0);
```

The `difftime` function returns the difference between `t0` and `t1`, in seconds. The parameters are specified in the format used by the `time` function.

```
printf("%.0f seconds have elapsed.\n", difftime(time(NULL), oldtime));
```

div ldiv

```
#include <stdlib.h>
typedef struct div_t {int quot,rem;};
typedef struct ldiv_t {long quot,rem;};

div_t div(int n, int d);
ldiv_t ldiv(long n, long d);
```

The function `div` computes the result of division and the remainder as a single step. The `quot` (quotient, or result of division) and `rem` (remainder) are returned in a structure. The function `ldiv` does the same thing for long arguments. The results are unpredictable if the denominator is zero.

```
/* print a result as a whole number and fraction */
res = div(numerator, denominator);
printf("%d and %d/%d", res.quot, res.rem, denominator);
```

dup

```
#include <fcntl.h>
int dup(int old);
```

Duplicates a file ID. The original file ID is created by a `creat` or `open` call. This call creates a duplicate of the file ID. The actual file on disk is not closed until each individual file ID is closed.

If the call is successful, the function returns 0; otherwise, a -1 is returned and `errno` is set as indicated in the list below.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

Possible errors

EBADF	Old is not a valid file descriptor
EMFILE	OPEN_MAX files are already open

enddesk

See `startdesk`.

endgraph

See `startgraph`.

EOF

```
#include <stdio.h>
#define EOF (-1)
```

EOF is a value used to see if you have reached the end of a file. If so, the file input routines will report this character as the one read. After seeing this character, always use the `feof` function to ensure that the end of the file has, indeed, been reached, as opposed to the file containing a character 0xFF which has been sign extended to become a -1.

```
/* process the characters in a file */
do {
    done = (ch = fgetc(myFile)) == EOF;
    if (done)
        done = feof(myFile);
    if (!done)
        process(ch);
}
while (!done)
```

errno perror strerror

```
#include <errno.h>
extern int errno;

#include <string.h>
char *strerror(int errnum);

#include <stdio.h>
void perror(char *s);
```

This collection of functions, variables and macros implement the standard run-time error package for C.

The variable `errno` is used by the libraries to report errors. Any time an error is detected by one of the libraries, an error number is stored in `errno`. Note that `errno` is never cleared by the libraries. To make effective use of `errno`, your program should clear `errno` before calling a function, call the function, then check `errno` to see if an error occurred.

The error numbers used by `errno` are defined as macros in two interface files. Mathematical errors can be found in `math.h`, while all other errors are defined in `errno.h`.

The `perror` function is used to print an error message. The error message is printed to error out. It consists of an error message (which is supplied as a parameter to `perror`), a colon and a space, a description of the error currently reported by `errno`, and a new line character.

The `strerror` function returns a pointer to a string. If `errno` is supplied as the argument, the result is identical to the message description printed by `perror`. (This is not true in all implementations of C, however.)

See also `toolerror`.

```
if (errno) {
    perror("Error at line __LINE__ of __FILE__");
    exit(errno);
}
```

exit _exit abort

```
#include <stdlib.h>
void exit(int status);
void _exit(int status);
void abort(void);
```

The `exit` function exits the program, calling all functions registered by the function `atexit`, and then performing the normal clean-up operations of closing open streams, flushing buffers, and deleting files created by calls to `tmpfile`. The `_exit` function also exits, but does not do the clean-up operations. Any open streams will still be closed by the shell, which will also dispose of any memory used by the program. Both functions accept an integer argument, which is returned to the shell as a completion code. A completion code of zero tells the shell that the program finished normally, and any script files will continue to execute. A non-zero value tells the shell that some error occurred. Some shell utilities also use the return code to return a value to the shell.

The `abort` function is the same as `_exit(-1)`. `Abort` is not always implemented the same way. Depending on the implementation, `abort` can exit with some other exit code (so long as it is non-zero), or even exit with a maskable interrupt, so that the program can handle the situation as an error and return to the caller, often returning a value.

```
exit(0);
_exit(errno);
```

exp

```
#include <math.h>
extended exp(extended x);
```

The `exp` function returns the extended floating-point representation of e raised to the x power, where e is the base of the natural logarithm (approximately 2.718281828). If the exponent cannot be represented as an extended number, an overflow results. In that case, infinity is returned and `errno` is set to `ERANGE`.

```
res = exp(x);
```

fabs

See `abs`.

fclose

```
#include <stdio.h>
int fclose(FILE *stream);
```

The function `fclose` takes an open file as input, and closes the file. If an error occurs, `fclose` returns EOF; otherwise, it returns zero.

```
fclose(myfile);
```

fcntl

```
#include <fcntl.h>
int fcntl(int filds, int cmd, int arg);
```

The function `fcntl` provides control over open files. The *filds* parameter is the file ID for a previously opened file. Under UNIX, the *cmd* parameter can take on a number of values, but on the Apple IIGS, the only value that makes sense is `F_DUPFD`, which makes a copy of the file descriptor. The number of the file descriptor returned is always greater than or equal to *arg*, and has exactly the same characters as the old file descriptor.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

If an error occurs, `fcntl` returns -1 and sets `errno` to one of these values; if an `errno` does not occur, `fcntl` returns 0.

Possible errors

EBADF	<i>filds</i> is not a valid file descriptor.
EINVAL	The <i>cmd</i> parameter is not <code>F_DUPFD</code> .
EMFILE	There are no available file descriptors greater than <i>arg</i> -1.
EMFILE	<i>arg</i> is negative.

feof

```
#include <stdio.h>
int feof(FILE *stream);
```

The feof function checks to see if an end of file condition exists for the specified stream. An end of file condition exists if an attempt has been made to read past the end of a file. Note that an end of file condition does not exist if all of the characters in a file have been read, but no attempt has been made to read another character. A value of zero is returned if an end of file condition does not exist; a non-zero value is returned if an end of file has been detected.

```
/* echo a file */
stream = fopen("myfile", "r");
if (stream != NULL)
    do
        putchar(fgetc(stream));
    while (!feof(stream));
fclose(stream);
```

ferror clearerr

```
#include <stdio.h>
int ferror(FILE *stream);
void clearerr(FILE *stream);
```

The ferror function returns a non-zero value if a read or write error has occurred on the specified stream. It returns a zero if no error has occurred.

If an error condition exists, it can be cleared by a call to clearerr or by closing the file.

```
if (ferror(myFile)) {
    clearerr(myFile);
    HandleError();
}
```

fflush

```
#include <stdio.h>
int fflush(FILE *stream);
```

The fflush function takes a stream open for output and flushes any internal buffers, writing them to the destination device. The stream remains open. EOF is returned if there is an error; otherwise, zero is returned.

```
fflush(myfile);
```

fgetc getc getchar

```
#include <stdio.h>
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
```

These functions are used to read characters from a stream. The function fgetc and the macro getc do exactly the same thing: the only difference is that fgetc is implemented as a function, while

`getc` is implemented as a macro. In each case, a character is read from the stream and returned. The current position of the stream is updated after each read. If an error occurs during the read or if a read is attempted after the last character in the file has been read, EOF is returned. In that case, the `feof` should be used to see if the end of file has actually been reached. `Errno` can also be checked to see if an error has occurred.

The function `getchar` works like an `fgetc(stdin)`, reading a character from standard in. If `stdin` is set to the keyboard (the default), control-@ is used to signal the end of a file. The end of a line is signaled with the `return` key or `enter` key. Note that the console input routines buffer the characters in chunks of one line to allow the user to edit the line while it is being typed. The `getchar` function will not return a character until an entire line has been typed.

```
ch = fgetc(myFile);
ch = getchar();
```

fgetpos fsetpos

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

The `fgetpos` function records the current position in a file. The information needed to restore the file to its position at the time of the call is recorded in a value of type `fpos_t`. The `fsetpos` function is used to restore the file to the position at the time of the `fgetpos` call.

If either call is successful, the function returns 0; otherwise it returns a non-zero value and sets `errno`. While the exact values returned vary from compiler to compiler, in ORCA/C, these functions return -1 for a failure, and set `errno` to `_IOERR`.

See also `fseek` and `ftell`.

fgets gets

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```

The function `fgets` reads a string from a file. Up to `n-1` characters are read; they are placed in the character array pointed to by `s`. A pointer to the first character in `s` is returned if the read was successful. Input stops if an end of file condition is encountered, a new line character is encountered, or if `n-1` characters are read. In all cases, a terminating null character is placed after the last character read. The new line character, if encountered, does become part of the string; it is placed just before the terminating null character.

If an end of file is encountered before any characters are read, or if an error condition occurs during the read, `fgets` returns a null pointer and the buffer `s` is left undisturbed.

The function `gets` reads characters into the buffer `s`, taking the characters from standard in. Characters are read until an end of line character is encountered. Unlike `fgets`, `gets` does not include the end of line mark as part of the string.

```
/* process a file */
do {
    fgets(line, 255, myFile);
    process(line);
}
while (!feof(myFile));
```

floor

See ceil.

fmod

```
#include <math.h>
extended fmod(extended x, extended y);
```

The function fmod returns the floating-point remainder of x/y. If the result cannot be represented as an extended floating-point number, the result is undefined. This function can be thought of as performing the division, then removing the whole-number portion of the result. If y is zero, x is returned.

See also modf.

```
x = fmod(x,y);
```

fopen freopen

```
#include <stdio.h>
FILE *fopen(char *filename, char *type);
FILE *freopen(char *filename, char *type, FILE *stream);
```

The function fopen opens a file. The name of the file to open is specified as a null-terminated string. It must be acceptable to the operating system as a file name.

The second string, type, is used to determine the characteristics of the file to be opened. It consists of one of three flags, r, w or a. Each of these flags can be followed by a + character, which indicates that a file is to be opened for both input and output. Even if the file is open for both input and output, a call to fseek, rewind or fflush must appear between input calls and output calls. The meaning of the flags is:

flag	meaning
r	Open an existing file for input.
w	Create a new file, or delete the contents of an existing one, for output.
a	Create a new file, or append to an existing file, for output.
r+	Open an existing file for input and output. The first record read/written will be at the start of the file.
w+	Create a new file or delete the contents of an old file, opening it for input and output.
a+	Create a new file, or append to an existing one, opening the file for input and output. The first value written will appear after all old entries in the file; reading without positioning the file mark will result in an end of file condition.

In addition, any of these flags may be followed by the character b. If the file does not exist, and the b flag is used, fopen will create a file with a BIN file type (a binary file); if the b flag is not used, the file type will be TXT (a text file). The other effect of the b flag is to change the way the \n character is handled. The C language uses the \n character to mark the end of a line, but not all computers use a \n character to mark the end of a line in a file. There are a number of conventions, but the most common are to use the \n character at the end of a line (e.g. UNIX), to use the \r character at the end of a line (e.g. Apple II, Macintosh) or to use both characters (many MS-DOS programs do this). If you open a file without the b flag, the ORCA/C libraries do their best to hide this difference between the way lines are marked by C and the way they are marked in a file. The

ORCA/C libraries automatically strip all `\n` characters, replacing them with `\r` characters on both input and output. If you use the `b` flag in `fopen`, the libraries do not change these characters.

If the call to `fopen` completes without error, `fopen` returns a pointer to a file record. This pointer can then be supplied as the input to calls that require an open stream. If an error occurs, `fopen` returns a null pointer and stores an error code into `errno`.

The function `freopen` is used to change the file associated with an existing stream. It starts by closing the file passed as a parameter, and then opens a new file using the same file buffer. If the open succeeds, `freopen` returns the original file buffer. If the open fails, `freopen` returns a null pointer and sets `errno`.

ORCA/C does not impose any practical restrictions on the number of open files. So long as the operating system does not complain, ORCA/C can handle thousands of open files. For programs that will be ported to other machines, use the macro `SYS_OPEN` to determine the maximum number of files that can be open at one time.

```
/* sample calls to fopen and freopen */
FILE *myfile;

myfile = fopen(fileName, "r"); /* open a file for input */
myfile = fopen("out.cc", "w"); /* prepare to write to out.cc */
myfile = fopen(fileName, "a"); /* place new info at the end */
myfile = fopen(fileName, "wb"); /* open a binary output file */
myfile = fopen(filename, "r+"); /* open a file for input & output */
```

fprintf printf sprintf

```
#include <stdio.h>
int fprintf(FILE *stream, char *format, ...);
int printf(char *format, ...);
int sprintf(char *s, char *format, ...);
```

These functions implement formatted output to a stream. They all process characters the same way; the difference is in where the characters are sent. In the case of `fprintf`, the characters can be sent to any stream open for output. The `printf` function writes characters to standard out. The `sprintf` function sends characters to a string, appending a terminating null character. In all cases, the number of characters written to the output device is returned if no error occurred, and EOF is returned if an error was detected. In the case of `sprintf`, the terminating null character is not included in the character count.

The format string controls the characters that are sent to the output device. In the simplest case the characters in the format string are simply copied to the output device. The string may, however, have embedded conversion specifiers. These take the form of a `%` character followed by other information. In most cases, a conversion specifier requires a value to convert and write. In that case, the values are taken from the variable length parameter list that follows the format string. The number and type of parameters expected by the conversion specifiers must match exactly with the number and type of parameters specified, or a crash is likely.

The general format for a conversion specifier is shown below. Most of the fields are optional; when this is true, it is noted in the text. The various fields are specified in the order given, with no intervening white space. Not all of the optional fields have any effect on a particular format specifier. In those cases, the field is still allowed, but is ignored.

`%` All conversion specifiers start with the `%` character.

`-, 0, +, space, #` Zero or more of these flag characters follow the `%` character, in any order. The flag characters are described in detail below.

unsigned int	An optional unsigned decimal constant specifies the minimum field width. This constant must not start with a zero, since the zero would be confused with the format flag. An asterisk can also be used for the minimum field width, in which case an integer parameter is used for the field width. If the number of characters needed to represent the value is greater than the field width, all of the characters are still printed. If the number of characters needed to represent the value is less than the minimum field width, extra characters are added to fill out the field. The character used can be a space or zero, and the extra characters can appear to the left or right of the value. These factors are under control of the flags.
.unsigned int	The precision of the field is specified by a period and an unsigned decimal constant. In general, this represents the number of digits that will be used to represent a numeric value. Like the field width, the precision can be specified as an asterisk, in which case an integer is removed from the parameter list and used as the precision. See the descriptions of the individual conversion specifiers for the specific use of this value.
h, l or L	This optional field is a size specifier. The h specifier indicates a short operand. It is valid with the d, o, u, x and X conversion specifiers, and is ignored for all other conversion specifiers. The l designator indicates that an integer value is long. It is valid with the d, o, u, x and X conversion specifiers, and is ignored for all other conversion specifiers. L is used with double-precision numbers to indicate double numbers. It is valid with the e, E, f, F, g and G conversion specifiers, and is ignored for all other conversion specifiers. The s and L designators are not needed by the formatter; they are included to make the format specifiers used here more compatible with those used by sscanf.
conversion	The conversion specifier tells what type of variable is being formatted. This field is required. Each of the individual conversion operators is described below.

The flag characters modify the normal operation of the format specifier.

-	If a formatted value is shorter than the minimum field width, it is normally right-justified in the field by adding characters to the left of the formatted value. If the - flag is used, the value is left-justified.
0	If a formatted value is shorter than the minimum field width, it is normally padded with space characters. If the 0 flag is used, the field is padded with zeros instead of spaces. The 0 pad character is only used when the output is right-justified.
+	When a negative number is formatted, it is always preceded by a minus sign. This flag forces positive numbers to be preceded by a plus sign.
space	When a negative number is formatted, it is always preceded by a minus sign. This flag forces positive values to be preceded by a space, lining up positive numbers with negative numbers of equal length.
#	This flag modifies the standard output format for certain numeric conversions. The specific effects are described with the conversion specifiers.

One of the conversion operations shown below must appear as the last character of any conversion specifier.

- d or i** One argument is removed from the parameter list and written as a signed, decimal number. If the `l` flag is used, the argument should be of type `long`; otherwise, the value should be of type `int`.
- If there is no precision specified, the sequence of digits created is as short as possible to represent the value. If a precision is specified, and the digit sequence is less than the precision, it is padded on the left with zeros to reach the specified number of digits. If the precision is zero and the value is zero, no value is printed. The default precision is one.
- The prefix is a `-` character if the argument is negative. For a positive argument, the prefix is `+` if a `+` flag is used, a space if the space flag is used, and there is no prefix if neither flag is used.
- The `#` flag is not used by the `d` conversion specifier.
- The `i` format specifier is identical to the `d` designator. It is included for compatibility with `fscanf`.
- u** One argument is removed from the parameter list and written as an unsigned, decimal number. If the `l` flag is used, the argument should be of type `unsigned long`; otherwise, the value should be of type `unsigned int`.
- If there is no precision specified, the sequence of digits created is as short as possible to represent the value. If a precision is specified, and the digit sequence is less than the precision, it is padded on the left with zeros to reach the specified number of digits. If the precision is zero and the value is zero, no value is printed. The default precision is one.
- The `#`, `+` and space flags are not used by the `u` conversion specifier.
- o** One argument is removed from the parameter list and written as an unsigned, octal number. If the `l` flag is used, the argument should be of type `unsigned long`; otherwise, the value should be of type `unsigned int`.
- If there is no precision specified, the sequence of digits created is as short as possible to represent the value. If a precision is specified, and the digit sequence is less than the precision, it is padded on the left with zeros to reach the specified number of digits. If the precision is zero and the value is zero, no value is printed. The default precision is one.
- The `+` and space flags are not used by the `o` conversion specifier.
- If the `#` flag is used, the number is preceded with a leading zero.
- x or X** One argument is removed from the parameter list and written as an unsigned, hexadecimal number. If the `l` flag is used, the argument should be of type `unsigned long`; otherwise, the value should be of type `unsigned int`.
- If there is no precision specified, the sequence of digits created is as short as possible to represent the value. If a precision is specified, and the digit sequence is less than the precision, it is padded on the left with zeros to reach the specified number of digits. If the precision is zero and the value is zero, no value is printed. The default precision is one.
- The `+` and space flags are not used by the `x` conversion specifier.
- If the `#` flag is used, the number is preceded by a leading `0x` (for the `x` specifier) or `0X` (for the `X` specifier).
- The `x` specifier causes the digits `a` through `f` and the `x` in the leading `0x` (if present) to be written in lowercase, while the `X` specifier writes these as uppercase letters. This is the only difference between the two specifiers.
- p** One pointer argument is removed from the parameter list and written as a pointer. The format used to write a pointer is implementation-defined; in ORCA/C, `%p` is completely equivalent to `%lX`.

- c One argument is removed from the parameter list and written as a character. The argument should be of type unsigned or int.
If the value is not a valid ASCII character, it is still sent to the output device. What effect this has depends on the device. Unless you are familiar with the output device, and are deliberately using a character for some special effect, you should stick to characters with ordinal values from 0 to 127.
The #, + and space flags are not used by the c conversion specifier.
- s One argument is removed from the parameter list and written as a string. The argument should be of type char *, and should point to a null-terminated string.
If no precision is specified, all characters up to, but not including the terminating null character are written to the output stream. If a precision is specified, the number of characters written will be the smaller of the precision and the length of the string.
The #, + and space flags are not used by the s conversion specifier.
- b One argument is removed from the parameter list and written as a string. The argument should be of type char *, and should point to a string with a length byte in the first character position.
If no precision is specified, all characters except the length byte are written to the output stream. Note that this gives one way of writing a null character to the output stream. If a precision is specified, the number of characters written will be the smaller of the precision and the length of the string, as specified by the length byte.
This conversion specifier is not present in ANSI C.
The #, + and space flags are not used by the s conversion specifier.
- n No characters are written to the output device. One argument is used from the parameter list; it must be of type int *. The number of characters that have been written up to this time by this formatting operation is saved to the specified location.
- f One argument is removed from the parameter list and written as a signed, floating-point number. The argument can be of type float, double, comp or extended.
If there is no precision specified, a precision of six is assumed. The number produced consists of at least one leading digit, but no more than are needed to represent the whole part of the number. This is followed by a decimal point and any fraction digits. The precision determines how many fraction digits are present.
If the precision is zero, no fraction digits are written. The decimal point is also not written unless the # specifier is used.
If the value cannot be represented accurately in the precision allowed, it is still written. The value is simply rounded to the closest value to the correct one.
The prefix is a - character if the argument is negative. For a positive argument, the prefix is + if a + flag is used, a space if the space flag is used, and there is no prefix if neither flag is used.
- e or E One argument is removed from the parameter list and written as a signed, floating-point number. The argument can be of type float, double, comp or extended.
The value written is in exponential format. It consists of a leading digit of 1 through 9 (or 0 if the actual value is 0), followed by a decimal point and any remaining digits. This is followed by an exponent, which is an e (for the e

specifier) or E (for the E specifier), followed by a plus or minus sign, and at least two exponent digits. If more than three digits are needed to express the exponent, then as few as possible are printed to represent the value.

The precision specifies how many digits appear after the decimal point. If no precision is given, a precision of six is used. If the precision is zero, the decimal point and digits are omitted unless the # flag is present, in which case the decimal point is printed, but no digits follow the decimal point.

The prefix is a - character if the argument is negative. For a positive argument, the prefix is + if a + flag is used, a space if the space flag is used, and there is no prefix if neither flag is used.

g or G One argument is removed from the parameter list and written as a signed, floating-point number. The argument can be of type float, double, comp or extended.

The g specifier is used to print a value in either exponential or fraction format, depending on which size is more appropriate. If the exponent for the number is less than -3 or greater than or equal to the precision specified, the e format specifier is used. When this happens, the effect is as if the e specifier was used with a # flag and a precision one less than the precision given for the g specifier. If no precision is specified, a precision of six is assumed; a precision less than or equal to zero is converted to a precision of one.

If the exponent value is greater than or equal to -3, and less than or equal to the precision, the number is formed using the rules for the f format specifier, with a precision equal to the specified precision less the value of the exponent, and again assuming that the # flag was used.

If the # flag was specified for the g conversion specifier, the resulting number is the one sent to the output stream. If the # specifier is not present, any trailing fractional zeros are stripped from the number. If all of the fractional digits are zero, the decimal point is also removed.

The prefix is a - character if the argument is negative. For a positive argument, the prefix is + if a + flag is used, a space if the space flag is used, and there is no prefix if neither flag is used.

% The % specifier is used to write a % character to the output stream. No arguments are removed from the stack. Note that the minimum field width, the - justification character and the pad characters are still used with the % specifier. The precision specifier and all other flags have no effect.

```
printf("Hello, world.\n");

for (i = 0; i < 10; ++i)
    printf("%d\n", i);

str = "Hello, world."; /* NOTE: str is defined as char *str */
for (i = 0; i < strlen(*str); ++i)
    printf("%s\n", i, str);

printf("x          sin(x)      cos(x)\n");
printf("-          -        -\n");
for (x = 0.0; x < pi; x += pi/10)
    printf("%10f %10f %10f\n", x, sin(x), cos(x));
```

fputc putc putchar

```
#include <stdio.h>
int fputc(char c, FILE *stream);
int putc(char c, FILE *stream);
int putchar(char c);
```

These functions are used to write characters to an output stream. The function `fputc` and the macro `putc` do exactly the same thing: the only difference is that `fputc` is implemented as a function, while `putc` is implemented as a macro. In each case, the character supplied as a parameter is written to the specified output stream and returned as the value of the function. If an error occurs during the write, EOF is returned instead of the character supplied as a parameter.

The function `putchar` works like an `fputc(c, stdout)`, writing a character to standard out.

```
/* write the printing ASCII characters to standard out */
for (i = 32; i < 128; ++i)
    putchar(i);
```

fputs puts

```
#include <stdio.h>
int fputs(char *s, FILE *stream);
int puts(char *s);
```

The function `fputs` writes the characters in a null-terminated string to an output stream. The string is not followed by a line feed, although any line feed in the string is written. If the write is successful, zero is returned; otherwise, EOF is returned.

The function `puts` is similar, but it writes the string to standard out. In addition, `puts` always writes a line feed character after writing the string.

```
int main(void)
{
    puts("Hello, world.");
}
```

fread

```
#include <stdio.h>
size_t fread(void *ptr, size_t element_size, size_t count,
             FILE *stream);
```

The function `fread` reads characters from a stream, placing them in a memory location pointed to by `ptr`. The array should be at least `count` elements long, with each element `element_size` bytes long. The value returned is the actual number of elements read, which can be less than `count` if an error occurs (in which case zero is returned) or an end of file is found (in which case the number of items read before the end of file was encountered is returned). If either `count` or `element_size` is zero, no characters are read, and zero is returned.

```
numItems = fread(array, sizeof(element), 40, myFile);
```

free

```
#include <stdlib.h>
void free(void *ptr);
void cfree(void *ptr);
```

The function `free` is used to deallocate memory allocated using `malloc`, `calloc` or `realloc`. Once deallocated, the memory can be reused by subsequent calls to `malloc` or `calloc`. Memory allocated by `malloc` is allocated from the toolbox Memory Manager either as a single chunk of memory (for requests over 4K bytes), or as a smaller part of a 4K byte piece of memory. For large chunks of memory, `free` returns the memory to Apple's Memory Manager, allowing other programs and tools to use the memory. For smaller pieces of memory, `free` will only return the memory to Apple's Memory Manager if all of the individual pieces have been freed.

Older C libraries require memory allocated with `calloc` to be freed with a call to `cfree`, rather than `free`. In ANSI C, `cfree` has been deleted, and `free` is used to deallocate all memory. In ORCA/C, `cfree` has been retained as a macro that calls `free` for compatibility with old programs.

See also `malloc`.

```
/* deallocate space for the array pointed to by aPtr */
free(aPtr);
```

freopen

See `fopen`.

frexp ldexp

```
#include <math.h>
extended frexp(extended x, int *nptr);
extended ldexp(extended x, int n);
```

The function `frexp` splits a floating-point number into a mantissa and exponent. The mantissa is returned as the result, and the exponent is saved to the integer pointed to by `nptr`. The mantissa will lie in the range 0.5 inclusive to 1.0 exclusive. The mantissa times 2 raised to the exponent will give the original argument `x`. If `x` is zero, `n` is set to zero and zero is returned.

The function `ldexp` reverses the effect of `frexp`. The argument `x` is multiplied by 2 raised to the power `n`, and the result is returned. If an error occurs, `errno` is set to `ERANGE`.

```
mantissa = frexp(val, &exponent);
val = ldexp(mantissa, exponent);
```

fscanf scanf sscanf

```
#include <stdio.h>
int fscanf(FILE *stream, char *format, ...);
int scanf(char *format, ...);
int sscanf(char *s, char *format, ...);
```

These three functions are used to read formatted input. In each case, a format string controls the number of items read, what type they are, and several other characteristics. The only difference between the three functions is the source of the characters to format. The `fscanf` function reads characters from any stream that is open for input. The `scanf` function is similar, but always takes its characters from standard in. The `sscanf` function reads the characters from a null-terminated string.

Each of the functions returns the number of successful scan operations. For example, if the function call is set up to read four integers, and all four are read successfully, the value four is returned. If an input/output error, end of file condition, or improper input data forces the read to stop early, only the number of successful scans is returned. If an end of file is encountered before any assignments are made, the functions return EOF. For the purpose of `sscanf`, the null terminator at the end of the string is treated as an end of file.

The format string describes how many variables should be read, as well as several other characteristics about the input. It consists of three kinds of characters.

- white space If a white space character is found in the format string, the scanner skips to the next non-white space character. All white space characters are also read from the input stream. The effect, then, is that any white space in the format string causes all white space in the input stream to be skipped.
- conversions Conversion specifiers start with a % character. The syntax for a conversion specifier is quite complex, so its format will be discussed later.
- other characters Any other character must match a character read from the input stream. If the character in the format string does not match the character in the input stream, the scanner stops. The character that caused the mismatch is left in the input stream for the next input call.

There must be exactly one variable in the parameter list for each conversion specifier. (There is an exception to this rule; it will be discussed when the format is explained.) The types of the format specifiers must match the type of the parameter exactly. The parameters are all pointers to a variable location; once a value is read by the scanner, it is stored at the location pointed to by the parameter.

Each conversion specifier has the following fields, in the order listed. Some of the fields are optional; if so, this is mentioned in the text.

- % All conversion specifiers start with the % character.
- * The asterisk is an assignment suppression flag. If present, it tells the scanner that it should read a value from the input stream, but that the value will not be saved. When the assignment suppression flag is used, a pointer is not removed from the variable parameter list.
- unsigned int An optional unsigned decimal constant specifies the maximum field width. If this field is present, no more than the specified number of characters will be read from the input stream. It is possible, however, that fewer than the specified number of characters will be read. The field width must be greater than zero.
- h or l This optional field is a size specifier. When used with an integer variable, h indicates that the variable is short, and l indicates that the variable is long. The l specifier can also be used with the floating-point types, where it indicates a double variable. The h and l specifiers are used with the d, u, o x or X format specifiers. In addition, the l specifier can be used with the f, e, E, g, or G floating-point specifiers. These specifiers are ignored when used with any other format specifier.
- conversion The conversion specifier tells what type of variable is being read. This field is required. Each of the individual conversion operators is described below.

There are a variety of conversion specifiers used to describe the variable to be read. These are shown in the table below.

d	<p>A signed decimal value is read from the stream, converted to the internal format used by the 65816, and saved. The type of the pointer for this operation must be short * if the h size specifier is used, int * if no specifier is used, or long * if the l size specifier is used.</p> <p>The scanner starts by skipping any white space characters in the input stream. These are not counted toward the maximum field width if one is specified. An optional sign is then allowed (+ or -). Next, the scanner expects zero or more decimal digits. Digits are read until a non-digit character is found, and end-of-file is encountered, or the maximum field width is reached. The digits read are then converted into a signed decimal integer and saved. If there are no digits, the resulting value is zero.</p> <p>The results are not predictable if an overflow occurs.</p>
i	<p>This format specifier is identical to the d format specifier, except that octal and hexadecimal numbers can be read. A number is considered to be octal if the first digit is a zero and the next character is not an x or X. A number is considered to be hexadecimal if the first two characters (after a leading sign) are 0x or 0X. Scanning of an octal number stops when a non-octal digit is found. Scanning of a hexadecimal number stops when a non-hexadecimal digit is found. The rules for converting the characters into a number are identical to those used by the compiler, but a trailing type marker is not allowed.</p>
u	<p>An unsigned decimal value is read from the stream, converted to the internal format used by the 65816, and saved. The type of the pointer for this operation must be short * if the h size specifier is used, int * if no specifier is used, or long * if the l size specifier is used. Except for the fact that a leading sign is not allowed and the numbers are unsigned, this conversion specifier is identical to the d specifier used for signed decimal conversion.</p>
o	<p>An unsigned octal value is read from the stream, converted to the internal format used by the 65816, and saved. The type of the pointer for this operation must be short * if the h size specifier is used, int * if no specifier is used, or long * if the l size specifier is used. No leading sign is allowed, and only octal digits are scanned, but with those exceptions, scanning works exactly like scanning of a signed decimal integer with a format specifier of d.</p> <p>While ORCA/C will stop scanning if an 8 or 9 is encountered in the input stream; the way these digits are handled is not consistent across all compilers.</p>
x or X	<p>An unsigned decimal value is read from the stream, converted to the internal format used by the 65816, and saved. The type of the pointer for this operation must be short * if the h size specifier is used, int * if no specifier is used, or long * if the l size specifier is used.</p> <p>The scanner starts by skipping any white space characters in the input stream. These are not counted toward the maximum field width if one is specified. Next, the scanner expects zero or more hexadecimal digits. Digits are read until a non-hexadecimal digit character is found or the maximum field width is reached. The digits read are then converted into an unsigned integer and saved.</p> <p>A leading 0x or 0X is allowed. If these characters appear, they are skipped; they do count toward the maximum field width.</p>

The x and X conversions are identical. Either conversion specifier will accept uppercase or lowercase hexadecimal digits.

- p A pointer is read from the source stream, converted to the internal format used by the 65816, and saved. Since the exact format of a pointer is implementation-defined, this conversion specifier should only be used to read pointers written with the %p conversion specifier with one of the print commands, like printf.
- c The pointer that is used from the variable list should be of type char *. If no field width is specified, exactly one character is read from the input stream and saved at the specified location. If an end of file condition exists, the conversion fails.

If a field width is specified, the pointer should point to an array of characters that can hold as many characters as the maximum field width. Unless an end of file condition occurs, characters are read and stored in the array until the entire field width has been read. No terminating null character is appended to the end of the array. The conversion operation fails, and no characters are saved, if an end of file is found before the entire field width has been processed.

Note that leading white space characters are not skipped by this conversion. While a size specifier can be present, it is ignored.
- s One pointer argument of type char * is used. A string is read from the stream and placed at the specified location.

The scanner starts by skipping any white space characters in the input stream. These are not counted toward the maximum field width if one is specified. Characters are read until an end of file is found, a white space character is found, or the maximum field width is reached (if one is specified). A null character is added to the end of the characters read. If an end of file is encountered before any non-white space characters have been found, the conversion operation fails.

While a size specifier can be present, it is ignored.
- b The b format specifier works exactly like the s format specifier, except that a string with a leading length byte is saved. The string also has a null terminator. If the input string is longer than 255 characters, the resulting string will be n mod 256 characters long, where n is the actual number of characters read.

Note that this conversion specifier is not present in ANSI C. It is included in ORCA/C to make it easier to deal with p-strings, which are used extensively by the toolbox.
- n No input is read from the stream, and any size or field width specifier is ignored. One argument is used from the parameter list; it must be of type int *. The number of successful conversions that have been performed up to this time by the scan operation is saved to the specified location.
- f, e, E, g or G A signed decimal floating-point value is read from the stream, converted to the internal format used by the 65816, and saved. The type of the pointer for this operation must be float * if no specifier is used, or double * if the l size specifier is used.

The scanner starts by skipping any white space characters in the input stream. These are not counted toward the maximum field width if one is specified. The number itself consists of an optional leading sign, an optional digit sequence, an optional decimal point, another optional digit sequence, and

an optional exponent. The exponent, if present, consists of the character `e` or `E`, an optional sign, and an optional digit sequence. If no digits appear before the exponent, the resulting value is zero. If no digits appear after the `e` or `E` character, the exponent value is zero. If the value is too large to represent, the result is an infinity with an appropriate sign. If the value is too small to represent, the result is zero. Note that the results of an overflow or underflow are not consistent across C compilers; other compilers may not return infinity or zero for overflow and underflow.

All of these format specifiers are identical. They will all accept any form of floating-point value, with or without an exponent.

`%` A single `%` character is expected in the input stream. The field width and size specifiers are ignored if present. No pointer is used from the parameter list.

`[...]` A string is read from the input stream. A pointer of type `char *` is used from the parameter list; the characters read are stored at that location.

This specifier allows you to state exactly what characters are allowed in the input stream. Any characters appearing between the `[` and `]` characters are allowed; any other characters are not allowed, and cause the scanner to stop. The characters can be specified in any order, and duplicates are allowed.

If the first character after the `[` character is the `^` character, the meaning of the characters is reversed. In that case, the characters specified are not allowed in the input stream, and any characters not listed are accepted.

Because of the syntax of the conversion specifier, it is not possible to specify a `^` character as the first character of a list of characters, nor is it possible to use the `]` character in the list.

Scanning continues until an end of file condition is reached, a character which is not in the allowed set of characters is found (in which case it remains unread), or, if a maximum field width is specified, the maximum number of characters have been read. The characters are stored at the location specified by the pointer in the parameter list, and a terminating null character is added to the characters.

While a size specifier can be present, it is ignored.

```
/* echo a file of integers */
while (fscanf(myFile, "%d", &i))
    printf("%d\n", i);

/* echo white space delimited words from a file */
while (fscanf(myFile, "%50s", str))
    printf("%s\n", str);

/* read words consisting of characters from a file */
while (fscanf(myFile,
    "%50[abcdefghijklmnopqrstuvwxyz" /* read a word */
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ]"
    "%[^abcdefghijklmnopqrstuvwxyz" /* skip non-word chars */
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ]", str))
    printf("%s\n", str);
```

fseek rewind

```
#include <stdio.h>
int  fseek(FILE *stream, long offset, int wherefrom);
void rewind(FILE *stream);
```

The `fseek` function is used to position the file pointer in a file. Stream is the file to position. Wherefrom is a code telling how to change the position. Three methods of changing the position are currently defined, and are shown in the table below. In ANSI C compilers, names are assigned for each of these codes; older compilers use the numeric values for the codes. Offset is the number of bytes to move by. If an end of file condition exists when this call is made, it is cleared. If an `ungetc` has been performed, its effects are ignored. The `fseek` function returns zero if the call is successful, and non-zero if an error occurs.

code	name	meaning
0	SEEK_SET	Moves to offset bytes from the beginning of the file.
1	SEEK_CUR	Moves offset characters from the current file position. Offset can be a negative quantity, which causes the file mark to move toward the start of the file. If an attempt is made to move before the beginning of the file, the mark is set to the first byte in the file. If an attempt is made to move past the end of the file, random bytes are added to extend the length of the file.
2	SEEK_END	Moves offset bytes relative to the end of the file. As with SEEK_CUR, an attempt to move before the start of the file moves to the start of the file, and an attempt to move past the end of the file extends the file to the necessary length with random bytes.

The `rewind` function is a simple form of the `fseek` function. It is equivalent to a call to `seek` with an offset of zero and wherefrom set to `SEEK_SET`.

See also `ftell`.

```
/* read a record based on an index */
fseek(myFile, index*sizeof(record), SEEK_SET);
fread(record, sizeof(record), 1, myFile);
```

fsetpos

See `fgetpos`.

ftell

```
#include <stdio.h>
long int ftell(FILE *stream);
```

The `ftell` function returns the current file mark for an open stream. In effect, this is the number of bytes between the start of the file and the next byte that will be read or written. It is generally used to record a position, with the result supplied to `fseek` at a later time.

If an error occurs, `ftell` returns `-1L` and sets `errno` to the error number of the error detected.

See also `fseek`.

```
position = ftell(myFile);
```

fwrite

```
#include <stdio.h>
size_t fwrite(void *ptr, size_t element_size, size_t count,
              FILE *stream);
```

The function `fwrite` writes `count` elements of an array with elements `element_size` bytes long to the output stream, taking the values from memory starting at the location indicated by `ptr`. The function returns the actual number of array elements written, which will be less than `count` if an output error has occurred.

```
fwrite(matrix, sizeof(float), 10*10, myFile);
```

getc

See `fgetc`.

getchar

See `fgetc`.

getenv

```
#include <stdlib.h>
char *getenv(const char *name)
```

The `getenv` call returns a pointer to the value of a shell variable. If the shell variable has not been set, or if the program is executing from outside of the ORCA programming environment, where shell variables do not exist, a NULL value is returned.

Note that the meaning of `getenv` varies from environment to environment. On other computers, `getenv` may return a value from a different source.

gets

See `fgets`.

gmtime

See `localtime`.

isalnum

```
#include <ctype.h>
int isalnum(char c);
```

`Isalnum` is a macro that returns a non-zero value if the argument is an alphanumeric character, and zero if the argument is not an alphanumeric character. The argument must lie in the range -1 to 255, or the result is not valid. The alphanumeric characters are shown below.

```

0 1 2 3 4 5 6 7 8 9
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```

```

for (i = 0; i < 128; ++i)
    if (isalnum(i))
        printf("%c is alphanumeric.\n", i);

```

isalpha

```

#include <ctype.h>
int isalpha(char c);

```

Isalpha is a macro that returns a non-zero value if the argument is an alphabetic character, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The alphabetic characters are show below.

```

a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```

```

for (i = 0; i < 128; ++i)
    if (isalpha(i))
        printf("%c is alphabetic.\n", i);

```

isascii

```

#include <ctype.h>
int isascii(char c);

```

Isascii is a macro that returns a non-zero value if the argument's ordinal value is in the range 0 to 127, which is the range if the ASCII character set used on the Apple IIGS. It returns zero if the character is outside of the range of the ASCII character set.

```

for (i = -500; i < 500; ++i)
    if (isascii(i))
        printf("%d is the ordinal value of an ASCII character.\n", i);

```

iscntrl

```

#include <ctype.h>
int iscntrl(char c);

```

Iscntrl is a macro that returns a non-zero value if the argument is a control character, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The control characters are all those characters with an ordinal range of 0 to 31, plus the character whose ordinal value is 127.

```

for (i = 0; i < 128; ++i)
    if (iscntrl(i))
        printf("%d is the ordinal value of a control character.\n", i);

```

iscsym

```
#include <ctype.h>
int iscsym(char c);
```

iscsym is a macro that returns a non-zero value if the argument is a character that can appear in a C identifier, and zero if it cannot. The argument must lie in the range -1 to 255, or the result is not valid. The characters which can appear in a C identifier are shown below.

```

_ 0 1 2 3 4 5 6 7 8 9
  a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

for (i = 0; i < 128; ++i)
    if (iscsym(i))
        printf("%c can appear in a C symbol.\n", i);
```

iscsymf

```
#include <ctype.h>
int iscsymf(char c);
```

Iscsymf is a macro that returns a non-zero value if the argument is a character that can appear as the first character of a C identifier, and zero if it cannot. The argument must lie in the range -1 to 255, or the result is not valid. The characters which can appear as the first character of a C identifier are shown below.

```

_ a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

for (i = 0; i < 128; ++i)
    if (iscsymf(i))
        printf("%c can appear as the first character in a C symbol.\n", i);
```

isdigit

```
#include <ctype.h>
int isdigit(char c);
```

Isdigit is a macro that returns a non-zero value if the argument is one of the ten decimal digits, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The ten decimal digits are shown below.

```

0 1 2 3 4 5 6 7 8 9

for (i = 0; i < 128; ++i)
    if (isdigit(i))
        printf("%c is a digit.\n", i);
```

isgraph

```
#include <ctype.h>
int isgraph(char c);
```

Isgraph is a macro that returns a non-zero value if the argument is one of the printing characters, other than a space. The argument must lie in the range -1 to 255, or the result is not valid. See isprint for a list of the printing characters.

```
for (i = 0; i < 128; ++i)
    if (isgraph(i))
        printf("%c is a graph character.\n", i);
```

islower

```
#include <ctype.h>
int islower(char c);
```

Islower is a macro that returns a non-zero value if the argument is one of the lowercase alphanumeric characters, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The lowercase alphanumeric characters are shown below.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z

for (i = 0; i < 128; ++i)
    if (islower(i))
        printf("%c is a lowercase character.\n", i);
```

isodigit

```
#include <ctype.h>
int isodigit(char c);
```

Isodigit is a macro that returns a non-zero value if the argument is one of the eight octal digits, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The eight octal digits are shown below.

```
0 1 2 3 4 5 6 7

for (i = 0; i < 128; ++i)
    if (isodigit(i))
        printf("%c is an octal digit.\n", i);
```

isprint

```
#include <ctype.h>
int isprint(char c);
```

Isprint is a macro that returns a non-zero value if the argument is one of the printing characters, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The printing characters include the entire ASCII character set with the control characters removed. This includes all characters with an ordinal value of 32 to 126. The printing characters are shown below. Note that the first printing character is a space.


```

! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~

for (i = 0; i < 128; ++i)
    if (ispunct(i))
        printf("%c is a printing character.\n", i);

```

ispunct

```

#include <ctype.h>
int ispunct(char c);

```

The function `ispunct` is a macro that returns a non-zero value if the argument is one of the punctuation characters, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The punctuation characters include the entire ASCII character set with the control characters and alphanumeric characters removed. This includes the space character, plus all of the characters shown below.

```

! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~

for (i = 0; i < 128; ++i)
    if (ispunct(i))
        printf("%c is a punctuation character.\n", i);

```

isspace

```

#include <ctype.h>
int isspace(char c);

```

`Isspace` is a macro that returns a non-zero value if the argument is a white space character, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The white space characters include the space, tab, carriage return, new line, vertical tab and form feed characters. These are the characters with ordinal values of 9 to 13 and 32.

```

for (i = 0; i < 128; ++i)
    if (isspace(i))
        printf("%d is the ordinal value of a white space character.\n", i);

```

isupper

```

#include <ctype.h>
int isupper(char c);

```

`Isupper` is a macro that returns a non-zero value if the argument is one of the uppercase alphanumeric characters, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The uppercase alphanumeric characters are shown below.

```

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```

isxdigit

```
#include <ctype.h>
int isxdigit(char c);
```

Isxdigit is a macro that returns a non-zero value if the argument is one of the sixteen hexadecimal digits, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The sixteen hexadecimal digits are shown below.

```
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F
```

```
for (i = 0; i < 128; ++i)
    if (isxdigit(i))
        printf("%c is a hexadecimal digit.\n", i);
```

labs

See abs.

ldexp

See frexp.

ldiv

See div.

localtime gmtime mktime

```
#include <time.h>
typedef unsigned long time_t;
struct tm {
    int tm_sec; /* seconds; 0 to 59 */
    int tm_min; /* minutes; 0 to 59 */
    int tm_hour; /* hours; 0 to 23 */
    int tm_mday; /* day of the month; 1 to 31 */
    int tm_mon; /* month; 0 to 11 */
    int tm_year; /* year; 0 == 1900 */
    int tm_wday; /* day of week; sun == 0 to sat == 6 */
    int tm_yday; /* day of year; 0 to 365 */
    int tm_isdst; /* daylight savings time?; a boolean value */
}

struct tm *localtime(time_t *t);
struct tm *gmtime(time_t *t);
time_t mktime(struct tm *ttmptr);
```

The localtime function takes a time encoded as an unsigned long integer and returns a structure with the various time fields filled in. See the definition of the tm structure, above, for a description of the fields that are filled in. On the Apple IIGS there is no way of telling if the current time is daylight savings time, so localtime always returns false (zero) for that field.

The `gmtime` function is designed to return the time on the prime meridian, which runs through Greenwich England. This time is known as Greenwich Mean Time, or GMT. Since the Apple IIGS does not have a way of knowing the difference between local time and GMT, the `gmtime` function returns the same values as `localtime` in ORCA/C.

The `mktime` function takes a time structure as input and returns the time in the long integer format returned by the `time` function. As a side effect, it also recomputes the `tm_wday` and `tm_yday` fields based on the values of the other fields. It returns -1 if the call fails.

See also `time`.

```
curtime = time(NULL);
t = *localtime(&curtime);
```

log

```
#include <math.h>
extended log(extended x);
```

The natural logarithm of the argument is returned. If the argument is negative, `errno` is set to `EDOM`. If the argument is zero or close to zero, a range error occurs and `errno` is set to `ERANGE`.

```
res = log(x);
```

log10

```
#include <math.h>
extended log10(extended x);
```

The base-10 logarithm of the argument is returned. If the argument is negative, `errno` is set to `EDOM`. If the argument is zero or close to zero, a range error occurs and `errno` is set to `ERANGE`.

```
res = log10(x);
```

longjmp

See `setjmp`.

lseek

```
#include <fcntl.h>
long lseek(int filds, long offset, int whence);
```

Repositions the file pointer, so that the next read or write occurs from a new position. The position is specified by `offset`. *whence* indicates one of the following:

- 0 The file pointer is set to offset bytes from the start of the file.
- 1 The file pointer is set to the current position plus offset bytes.
- 2 The file pointer is set to offset bytes from the end of the file.

If the call is successful, the function returns 0; otherwise, a -1 is returned and `errno` is set as indicated in the list below.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

See also `fseek`.

Possible errors

EBADF	Old is not a valid file descriptor.
EINVAL	whence is not 0, 1 or 2.

malloc calloc

```
#include <stdlib.h>
void *malloc(size_t size);
void *mllalloc(size_t size);
void *calloc(size_t count, size_t size);
void *cllalloc(size_t count, size_t size);
```

The function malloc allocates size bytes of memory and returns a pointer to the first byte of that memory. If there is not enough free memory to satisfy the request, or if the size is zero, malloc returns NULL.

On the Apple IIGS, memory is allocated through Apple's Memory Manager. If you ask for more than 4096 bytes of memory, the memory will be requested from the Memory Manager in a single chunk. There is no guarantee that the memory will not cross a bank boundary. While special memory is only used if all other memory has already been allocated, there is also no guarantee that the memory will not be special memory. (Special memory consists of memory from banks 0, 1, 0xE0, and 0xE1. This memory is used for special purposes on the Apple IIGS.) If a call to malloc is made asking for less than 4096 bytes of memory, 4096 bytes are still allocated from the Memory Manager. This memory is then subdivided by the compiler's run-time memory manager to satisfy other requests for small amounts of memory, taking a burden off of the toolbox's Memory Manager.

The calloc function accepts two parameters, a count and a size. Enough memory is allocated to hold count elements of size size, using the same mechanism for allocating memory as is used by malloc. All of the allocated memory is then set to zero.

Older C libraries often restrict malloc and calloc to a parameter of size unsigned, which would limit these functions to allocating 64K bytes of memory at a time. ANSI C requires parameters of type unsigned long, and eliminates mllalloc, which is used in older compilers when more than 64K bytes of memory is needed. ORCA/C still includes mllalloc as a macro which calls malloc, but its use should be avoided in new programs. The function cllalloc was provided for similar reasons; again, cllalloc is a macro in ORCA/C; it calls calloc.

See also free, realloc.

```
/* allocate space for the array pointed to by aPtr */
aPtr = malloc(sizeof(*aPtr));
```

memchr

```
#include <string.h>
void *memchr(const void *ptr, int val, size_t len);
```

The function memchr searches for a byte with the value val starting at the location pointed to by ptr for a maximum of len bytes. If a matching byte is found, a pointer to that byte is returned. If a matching byte is not found, a null pointer is returned.

See also strchr. Unlike strchr, memchr does not stop if a zero value is found.

```
/* skip to the next line in a TXT file */
file = ((char *) memchr(file, '\r', 255))+1;
```

memcmp

```
#include <string.h>
int memcmp(void *ptr1, void *ptr2, size_t len);
```

The function `memcmp` compares two areas of memory, one starting at `ptr1`, and the other starting at `ptr2`, for a length of `len` bytes. If all bytes match, `memcmp` returns zero. If a byte is found that does not match, a negative number is returned if the byte pointed to by `ptr1` is less than the byte pointed to by `ptr2`, and a positive value is returned otherwise.

See also `strcmp`. Unlike `strcmp`, `memcmp` does not stop if null characters are found.

```
/* see if two files loaded into memory are equal */
if (memcmp(file1, file2, fileLen) == 0)
    printf("The files are equal.\n");
```

memcpy memmove

```
#include <string.h>
void *memcpy(const void *dest, void *src, size_t len);
void *memmove(void *dest, const void *src, size_t len);
```

The function `memcpy` copies `len` bytes from the location starting at `src` to the location starting at `dest`. It returns the value of `src`. If the memory areas overlap, the results are unpredictable. The function `memmove` is similar, but is guaranteed to work if the memory areas overlap. Regardless of the location of the memory areas, `memmove` will produce a faithful copy of the original memory area at the destination.

```
/* initialize a doubly-subscripted array row-by-row */
for (i = 0; i < maxCol; ++i)
    a[0][i] = 1.0;
for (i = 1; i < maxRow, ++i)
    memcpy(a[i], a[0], sizeof(a[0]));
```

memmove

See `memcpy`.

memset

```
#include <string.h>
void *memset(const void *ptr, int val, size_t len);
```

The function `memset` copies the value `val` into an area `len` bytes long, starting at `ptr`. It returns the value of `ptr`.

```
/* clear a large array */
memset(array, 0, sizeof(array));
```

mktime

See `localtime`.

mlalloc

See malloc.

modf

```
#include <math.h>
extended modf(extended x, int *nptr);
```

The function `modf` returns the fraction part of the argument `x`. As a side effect, the integer part is placed in the location pointed to by `nptr`. The fraction part and integer part will have the same sign.

See also `fmod`.

```
res = modf(x, &i);
```

offsetof

```
#include <stddef.h>
size_t offsetof(type, member);
```

The `offsetof` macro returns the position of a structure member within a structure. The first input is a struct type. It must be defined in such a way that the expression

```
&(type *)
```

is legal; for example, any type defined like this:

```
typedef struct type {...} type;
```

will work. The second parameter is the name of a variable in the structure. The value returned is the number of bytes that occur before the member in the structure.

open

```
#include <fcntl.h>
int open(char *path, int oflag);
```

Creates a new file or opens an existing one for output.

The *oflag* parameter is formed by oring one or more of the following flags.

<code>O_RDONLY</code>	Open for read only.
<code>O_WRONLY</code>	Open for write only.
<code>O_RDWR</code>	Open for both reading and writing.
<code>O_NDELAY</code>	Not used on the Apple IIGS. Included for UNIX compatibility.
<code>O_APPEND</code>	If this flag is set, the file pointer is advanced to the end of the file before every write operation.
<code>O_CREAT</code>	If the file does not exist, one is created.
<code>O_TRUNC</code>	If the file exists, the length is set to 0 after opening the file.
<code>O_EXCL</code>	If <code>O_EXCL</code> and <code>O_CREAT</code> are both set and the file exists, the call fails.
<code>O_BINARY</code>	The file opened is a binary (BIN) file, rather than the default text (TXT) file. For text files, <code>\n</code> characters are translated to <code>\r</code> characters on output, and <code>\r</code>

characters are translated to `\n` characters on input, conforming to Apple II GS conventions. For binary files, no such conversion is performed.

This flag is unique to the Apple II GS.

If the call is successful, the function returns 0; otherwise, a -1 is returned and `errno` is set as indicated in the list below.

Possible errors

EACCES An I/O error or invalid pathname prevented opening of the file.
 EEXIST O_CREAT and O_EXCL are set, and the file exists.
 EMFILE OPEN_MAX files are already open.
 ENOENT O_CREAT is not set and the file does not exist.
 ENOENT The pathname is null.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

See also `fopen`.

p2cstr

See `c2pstr`.

pow

```
#include <math.h>
extended pow(extended x, extended y);
```

The argument `x` is raised to the power of the argument `y`, and the result is returned. If `x` is not zero, but `y` is zero, 1.0 is returned. If `x` is zero and `y` is positive, 0.0 is returned. Domain errors will occur if `x` is negative and `y` is not an integer, or if `x` is zero and `y` is zero or negative. In these cases, `errno` is set to `EDOM`. Range errors can also occur; in these cases, `errno` is set to `ERANGE`.

```
cube = pow(num, 3.0);
```

putc

See `fputc`.

putchar

See `fputc`.

puts

See `fputs`.

qsort

```
#include <stdlib.h>
void qsort(void *base, size_t count, size_t size,
           int (*compar)(const void *ptr1, const void *ptr2));
```

The qsort function sorts an array. The array has count elements, each of which is size bytes long. The base parameter points to the first element of the array. The function compar is supplied by the program; it accepts two pointers as arguments, each of which points to an element of the array, and returns an integer as a result. The value returned is 0 if the first element is the same as the second, -1 if the first element is less than the second, and 1 if the first element is greater than the second. After the call, the array will be sorted in ascending order.

```
int CompareZip(address *addr1, address *addr2)
{
    if (addr1->zip == addr2->zip)
        return 0;
    if (addr1->zip < addr2->zip)
        return -1;
    return 1;
}

...
/* sort the addresses by zip code */
qsort(addressList, listSize, sizeof(address), CompareZip);
```

raise

```
#include <signal.h>
int raise(int sig);
```

The raise function generates a signal that will be handled by the signal handler. See signal for a way to install a signal handler and a description of signals.

The function returns 0 if the sig parameter is valid, and 1 if it is not. If the signal is not valid, errno is also set to ERANGE.

rand srand

```
#include <stdlib.h>
#define RAND_MAX 32767
int rand(void);
void srand(int seed);
```

The function srand is used to initialize a pseudo-random number generator. Subsequent calls to rand will return integers in the range 0 to RAND_MAX.

Choosing a seed value is very important. If the same seed is used each time a program is executed, the program will always return the same pseudo-random number sequence. The seed should be chosen in such a way that it will be fairly random itself. One way of choosing a seed that works in most situations is to use the seconds in the current time of day.


```
time_t t;
struct tm trec;

t = time(NULL);
trec = *localtime(&t);
srand(trec.tm_sec);
for (i = 0; i < 100; ++i)
    printf("%8d", rand());
```

read

```
#include <fcntl.h>
int read(int filds, void *buf, int n);
```

Up to n bytes are read from the file whose file ID is *filds*. The bytes are placed in the buffer pointed to by *buf*.

If an error occurs during reading, the function returns -1 and sets *errno* as indicated in the table below. If the file is already at the end of file mark, a zero is returned, but no error is set. If there are n or more bytes left in the file, n bytes are read, and n is returned; the file pointer is advanced n bytes. If there are less than n bytes left in the file, the number of bytes remaining in the file are read, and this number is returned; the file pointer is then advanced to the end of the file.

If the *O_BINARY* flag was not set when the file was opened, any *\r* characters are converted to *\n* characters as they are read.

Possible errors

EIO	A physical I/O error occurred.
EBADF	<i>filds</i> is not a valid file ID.
EBADF	The file is not open for reading.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

See also *fread*.

realloc

```
#include <stdlib.h>
void *realloc(char *ptr, size_t size);
void *relalloc(char *ptr, size_t size);
```

The function *realloc* takes a pointer to a piece of memory allocated previously by *malloc* or *calloc* and changes the size of the memory. If the memory area shrinks, the deleted bytes are no longer available, although the memory not deleted is not disturbed. If the memory area grows, the old contents are preserved, but the value of the new bytes is unpredictable. A pointer is returned if the request is successful; the memory area may have been moved, so old copies of the pointer are no longer useful. If the request cannot be satisfied, a null pointer is returned, and the old memory area is not disturbed.

If a null pointer is passed to *realloc*, the function behaves as if *malloc* had been called. If the pointer is not null, and the size is zero, the memory is deallocated and *NULL* is returned.

In older C compilers, *realloc* was limited to a parameter of size unsigned, and another function called *relalloc* was used for requests for larger memory. In ANSI C, *relalloc* does not exist, and *realloc* can be used to request any amount of memory. ORCA/C includes *relalloc* as a macro equivalent of *realloc* for compatibility with older programs.

See also *free*, *malloc*.

```

/* Bytes are obtained from InByte, buffered until no    */
/* more memory is available, and processed by DoIt.    */
length = 0;
size = remaining = growSize;
ptr = start = malloc(growSize);
if (ptr == NULL)
    printf("Insufficient memory.\n");
else {
    ptr++ = InByte();
    length++;
    if (--remaining == 0) {
        size += growSize;
        ptr = realloc(start, size);
        if (ptr == NULL) {
            DoIt(start, length);
            free(start);
            length = 0;
            size = remaining = growSize;
            ptr = start = malloc(growSize);
        }
        else {
            start = ptr;
            ptr += length;
        }
    }
}
}

```

remove

```

#include <stdio.h>
int remove(char *filename);

```

The named file is deleted from the disk. The file name can be any file name or path name. A zero is returned if the delete is successful, and a non-zero is returned if the delete is not successful.

```
remove("/hd/languages/apwc");
```

rename

```

#include <stdio.h>
int rename(char *oldname, char *newname);

```

The file with the name represented by the string oldname is renamed. The file names can be any file name or path name. After the rename call, the name of the file is the name given by newname. If the rename is successful, the function returns a zero; otherwise, a non-zero value is returned.

The rename function is implemented by calling the GS/OS rename facility. Because of this, rename is more powerful under ORCA/C than it is under some other C implementations. This call can be used to move a file from one directory to another by specifying the path names of the old and new files.

```

/* change a n file name */
rename("myfile", "hisfile");

/* move a file */
rename("/hd/system/finder", "/hd/utilities/finder");

```

rewind

See `fseek`.

scanf

See `fscanf`.

setbuf setvbuf

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

The function `setvbuf` is used to change the size and characteristics of the default file buffer. By default, when `fopen` opens a file, the file has a `BUFSIZ` (1024) byte buffer used to collect characters to avoid the overhead of calls to the operating system for small read/write requests. The file is fully buffered, which means that, barring intervention with a call to `fflush` or `fclose`, 1024 bytes are collected before data is written to the disk, and 1024 bytes are read from disk to satisfy any read request.

When `setvbuf` is called, the first parameter is a pointer to an open stream. This stream must already be open, but no input or output calls are allowed before the call to `setvbuf`. The `buf` parameter is a pointer to a character buffer to use as the file buffer. If `buf` is a null pointer, `setvbuf` will allocate a buffer from available memory; if `buf` is not null, it should be at least `size` bytes long. `Size` is the number of bytes to use for a buffer, while `type` is the type of buffering to use. There are three buffering types, designated using macros defined in the `stdio.h` interface files. These are:

type	meaning
<code>_IOFBF</code>	Full buffering. A buffer of <code>size</code> bytes is collected before output operations, and a buffer of <code>size</code> bytes is read for any read operation. This is the default buffering mechanism.
<code>_IOLBF</code>	Line buffering. On output, the buffer is flushed any time a new line character is written to the buffer, or when the buffer is full.
<code>_IONBF</code>	No buffering. All output passes directly to the operating system, and all input is satisfied with direct calls to the operating system.

The function `setbuf` is a simplified form of `setvbuf`. If the `buf` is a null pointer, no file buffering is used. If the `buf` pointer is not null, full buffering is used, and a pointer to a buffer of `BUFSIZ` bytes long is required as the input.

```
/* Use line buffering, using line as a buffer 255 characters */
/* long. */
setvbuf(myFile, line, _IOLBF, 255);
```

setjmp longjmp

```
#include <setjmp.h>
typedef int jmp_buf[4];

void longjmp(jmp_buf env, int status);
int setjmp(jmp_buf env);
```

The setjmp and longjmp functions are used to transfer control to a previous point in the program. The setjmp function is used to establish a jump location in the program. A jump buffer is passed; this jump buffer is later used by longjmp. The setjmp function returns a value of zero.

Later, a call is made to longjmp, and the jump buffer and an integer are passed as parameters. The call can be made from the function that called setjmp, or from any function it called or that was called by a function it called (and so on), but longjmp must not be used from a function that called setjmp. Control is returned as if setjmp was called. The call to setjmp is not actually made; instead, a return is simulated. The stack is cleaned up, including disposing of the space used by any local auto variables. The value passed to longjmp as a status, which must be non-zero, is returned as the return value of setjmp. As far as the function that called setjmp in the first place can tell, setjmp has simply returned with a non-zero value.

If longjmp is called with an invalid buffer or the function containing the setjmp call returns before the longjmp call is made, the results are unpredictable.

```
jmp_buf buffer;

void test(void)
{
    printf("In test.\n");
    longjmp(buffer, 1);
    printf("This line will not be executed.\n");
}

int main(void)
{
    if (setjmp(buffer))
        printf("Returned from jump.\n");
    else {
        printf("Setting up the jump.\n");
        test();
    }
}
```

signal

```
#include <signal.h>
void (*signal(int sig, void(*func)(int)))(int);
```

Signals are used to handle asynchronous interrupts. They are primarily used on multi-tasking systems, where another executing program or some hardware facility can cause an interrupt. In that sort of environment, signals can be used to inform the program what has happened. Signals are not used on the Apple IIGS; the signal.h library is included solely to aid in porting programs from multi-tasking environments.

The signal function is used to specify how the various signals should be handled. The first parameter tells the signal handler which of the various signals you are talking about. ORCA/C supports the standard signals required by the ANSI standard. They are:

SIGABRT	Abnormal termination.
SIGFPE	Arithmetic errors.
SIGILL	Invalid function image.
SIGINT	Interactive attention signal.
SIGSEGV	Invalid memory access.
SIGTERM	Termination request sent to the program.

None of these errors apply to the Apple IIGS, and the ANSI standard does not require any of them to be handled. The SIGILL and SIGSEGV errors simply don't exist on the Apple IIGS. Abnormal terminations on the Apple IIGS are severe enough that attempting to execute a function after one occurs would be foolish; it could damage data on a disk, for example. The other signals either have no direct counterpart on the Apple IIGS, or are handled by the Apple IIGS toolbox, which is not always active.

The second parameter to signal is either SIG_DFL, SIG_IGN, or a function. SIG_DFL tells the signal handler to use default handling, while SIG_IGN tells the signal handler to ignore the signal. With ORCA/C, these are the same, since the default handling is to ignore a signal. If you pass a function, the function will be called whenever the raise function is called to raise a signal. The signal number (e.g., SIGABRT) is passed to your function.

The signal function returns the value of the previous signal handler, which you can save, and then restore later. If the *sig* parameter is not one of the signals listed above, signal returns SIG_ERR and sets errno to ERANGE.

See raise for a way to create a signal.

sin

```
#include <math.h>
extended sin(extended x);
```

The sin function returns the trigonometric sine of the argument. The argument must be supplied in radians.

```
height = sin(x)*hypotenuse;
```

sinh

```
#include <math.h>
extended sinh(extended x);
```

The sinh function returns the hyperbolic sine of the argument. If an error occurs, errno is set to ERANGE.

```
n = sinh(x);
```

sqrt

```
#include <math.h>
extended sqrt(extended x);
```

The square root of the argument is returned. If the argument is negative, `errno` is set to `ERANGE`.

```
length = sqrt(x*x + y*y);
```

srand

See `rand`.

sscanf

See `fscanf`.

stderr

See `stdin`.

stdin stdout stderr

```
#include <stdio.h>
extern FILE *stderr;
extern FILE *stdin;
extern FILE *stdout;
```

These three predefined variables are the standard input stream (`stdin`), the standard output stream (`stdout`), and the error output stream (`stderr`). They are already open when any C program that runs under the shell is executing. In C programs that do not run under the shell, your program should open them using `freopen` before using any of these streams explicitly, or before using any of the file input and output facilities that automatically use one of these streams.

When running under the shell, the shell will automatically handle any input or output redirection. If no redirection has been specified, input will come from the keyboard, and both standard out and error out will appear on the text screen (text environment) or the shell window (desktop environment).

```
/* use the GS/OS console driver for standard out */
freopen(".CONSOLE", "w", stdout);
```

stdout

See `stdin`.

strcat strncat

```
#include <string.h>
char *strcat(char *s1, char *s2);
char *strncat(char *s1, char *s2, size_t n);
```

Strcat is a function that appends the contents of the string s2 to the contents of the string s1. A terminating null character is added to the end of the new string. Strcat returns a pointer to the first character in the string s1.

Strcat will append characters until a terminating null is found in the string s2, even if the memory allocated for s1 is not large enough to hold the completed string. It is up to the programmer to ensure that the memory allocated for s1 is large enough to hold the concatenated string and the terminating null character.

Strncat is a similar function that limits the total number of characters that can be copied. No more than n characters will be appended to the end of s1. If a terminating null character is found in s2 before n characters have been copied, all of the characters in s2, including the terminating null character, are copied to s1. If n characters are copied, and no terminating null character has been found, strncat adds a terminating null character to s1 and stops copying characters. If n is less than or equal to zero, no characters are copied.

The result is unpredictable if the two strings overlap in memory.

```
strcat(fileName, ".obj"); /* add .obj to a file name */

n = 64-strlen(pathName); /* add a file name to a path name, */
strncat(pathName, "/", n--); /* making sure there are no overflows */
strncat(pathName, fileName, n);
```

strchr strpos strrchr strrpos

```
#include <string.h>
char *strchr(char *s1, char c);
int    strpos(const char *s1, int c);
char *strrchr(char *s1, char c);
int    strrpos(const char *s1, int c);
```

These functions scan a null-terminated string for a character. Strchr and strrchr return a pointer to the character if it is found. A null pointer is returned if the character is not found. The null character is considered to be a character for the purposes of the scan, so searching for the null character will return a pointer to the terminating null. The difference between the two functions is that strchr returns a pointer to the first character that matches, while strrchr returns a pointer to the last matching character.

The function strpos returns the position of the first matching character in the string. The position is the number of characters that appear before the matching character. If no matching character is found, -1 is returned.

The function strrpos returns the position of the last matching character. Like strpos, it returns -1 if there are no matches.

See also memchr.

```
nextSpace = strchr(str, ' '); /* find the next space */

lineLen = strpos(line, '.'); /* find the # of chars in the sentence */
```

strcmp strncmp

```
#include <string.h>
int strcmp(char *s1, char *s2);
int strncmp(char *s1, char *s2, size_t n);
```

Strcmp compares two null-terminated strings. The strings are considered equal if all of the characters up to and including the terminating null character in each string matches. The string s1 is less than the string s2 if the ordinal value of the first mismatched character is smaller in s1, or if both strings match up to the terminating null character in s1, but s2 is longer than s1. Strcmp returns zero if the strings are equal, a negative integer if s1 is less than s2, and a positive integer if s1 is greater than s2.

Strncmp is similar to strcmp. The difference is that strncmp will stop comparing the strings after n characters, even if a terminating null character has not been found in either string. In that case, the strings are considered to be equal. If a terminating null character appears in either string before n characters have been processed, strncmp returns the same result strcmp would have returned. If n is zero or negative, strncmp returns zero.

See also memcmp.

```
/* scan a table to find a matching name */
for (i = 0; i < tableLength; ++i)
    if (!strcmp(name, table[i]))
        goto found;

/* scan addresses that start with a five-digit zip code, */
/* stopping if a match is found */
for (i = 0; i < tableLength; ++i)
    if (!strncmp(zipCode, addresses[i], 5))
        goto found;
```

strcpy strncpy

```
#include <string.h>
char *strcpy(char *s1, char *s2);
char *strncpy(char *s1, char *s2, size_t n);
```

Strcpy is a function that copies the contents of s2 to the string s1. All characters up to and including the terminating null character are copied. It is up to the programmer to ensure that the memory available for s1 is large enough to hold all of the characters, including the terminating null character.

Strncpy is similar, but it will stop copying characters after n characters have been copied. If n characters are copied and no terminating null has been found, s1 will not have a terminating null. If s2 contains fewer than n characters, s1 is padded with null characters until n characters have been moved. If n is zero or negative, no copying is performed.

Both functions return s1 as their result.

The result is unpredictable if the two strings overlap in memory.

```
strcpy(str, "This is a test.\n");          /* copy a constant to a string */

pos = strpos(line, ' ');    /* read the first word from a string */
if (pos == -1)
    strcpy(str, line);
else
    strncpy(str, line, pos);
```


strcspn strpbrk strrpbrk strspn

```
#include <string.h>
size_t  strspn(const char *s, const char *set);
size_t  strcspn(const char *s, const char *set);
char    *strpbrk(const char *s, const char *set);
char    *strrpbrk(const char *s, const char *set);
```

These functions all scan the string *s* and check each character to see if it is in the set of characters formed by the null-terminated string *set*. The set of characters can contain no characters, or all of the ASCII characters, and it can contain duplicate characters. The order of the characters in the set does not affect the scan.

The function *strspn* scans the string *s* for the first character that is not in the set. The length of the longest initial segment in *s* containing only characters in *set* is returned. If all of the characters are in the set, the effect is to return the length of the string. If the set is the null string, zero is returned.

The function *strcspn* performs the opposite check: it skips over characters that are not in the set, stopping when a character is found that is in the set.

The function *strpbrk* scans the string, skipping over characters that are in the set, and stopping at the first character that is not in the set, just like *strcspn*. The difference is that *strpbrk* returns a pointer to the character, rather than the number of characters skipped. If the string *s* does not have any characters from the set, NULL is returned.

The function *strrpbrk* works like *strpbrk*, except that it returns a pointer to the last character in the string that is in the set, rather than the first character. If the string *s* does not have any characters from the set, NULL is returned.

```
/* find and process all of the words in the */
/* string pointed to by line                */
strcpy(alpha, "abcdefghijklmnopqrstuvwxyz"
            "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
while (*line) {
    line = strpbrk(line, alpha);
    len = strspn(line, alpha);
    if (len) {
        process(strncpy(word, line, len));
        line += len;
    }
}
```

strlen

```
#include <string.h>
size_t strlen(const char *string);
```

The function *strlen* returns the number of characters in a string. The number of characters is the number of characters that appear before the terminating null character.

In older versions of C, *strlen* returned an *int*, which limited the length of string that could be processed by *strlen*. In ANSI C, *strlen* returns *size_t*, which is unsigned long in ORCA/C. This means that *strlen* can effectively find the length of any string that can be held in memory. If you are porting a program that is built around the assumption that *strlen* is *int*, you can safely change the return type of *strlen* to *int* in a copy of the *string.h* interface file. In that case, all but the least significant two bytes are dropped from the length of the string.

```
len = strlen(myString);
```

strerror

See errno.

strpos

See strchr.

strrchr

See strchr.

strrpos

See strchr.

strstr

```
#include <string.h>
char *strstr(char *src, char *sub);
```

The function strstr scans the string str for the first occurrence of the string sub. If a match is found, a pointer to the first character of the match is returned. If no match is found, a null pointer is returned.

```
/* see if the correct answer occurs in the reply */
if (strstr(reply, "Santa Fe") != NULL)
    printf("Correct! Santa Fe is the capitol of New Mexico.\n");
```

strtod strtol strtoul

```
#include <stdlib.h>
double      strtod(char *str, char **ptr);
long      strtol(char *str, char **ptr, int base);
unsigned long strtoul(char *str, char **ptr, int base);
```

These functions convert numbers represented as ASCII strings to the internal binary format used for calculations. In each case, str is a null-terminated ASCII string. Any leading white space is skipped, and then the longest sequence of characters which can be legally interpreted as a number are read and converted. The number is returned as the function's return value.

If no conversions are possible, either because a null string was passed or because the first non-white space characters could not be interpreted as a number, zero is returned, the global variable errno is set to ERANGE, and ptr, if not NULL, is set to str. If the number causes overflow or underflow, errno is still set to ERANGE. In the case of underflow, zero is returned; in the case of overflow, infinity is returned. (Infinity is also available as the macro HUGE_VAL.)

The second parameter can be null, or it can point to a character pointer. If it is not null, and a valid string for conversion exists, it is set to point to the character immediately after the last character used in the conversion process. This is still true if an overflow or underflow occurred. If str is null, or if no valid numeric string is found, ptr is set to the value of str.

The function strtod converts a floating-point string to an extended value. The string has the same format as a string in a C source file. It can consist of a leading plus or minus sign. This is followed by a sequence of decimal digits, a decimal point, and another sequence of decimal digits.

At least one of these digit sequences must appear, but either can be omitted, as can the decimal point. This is followed by an optional exponent, which, if present, consists of an e or E, an optional plus or minus sign, and a digit sequence. Imbedded white space is not allowed. Some examples of legal strings include:

```
1.0      .1      3      1e+80 1.2e-3      .1e-4 1.e17
```

The functions `strtol` and `strtoul` both have an additional parameter that is used to specify the base of the integer. The base can be zero or any number from 2 to 36. If the base is zero, the format of the number itself is used to determine the base of the number, using the same rules used by the C compiler. Namely, if the number starts with a digit from 1 to 9, it is base 10; if the number starts with 0x or 0X, it is base sixteen; and if the number starts with 0, and is not followed by an x or X, it is base 8. Any of the other values specify the base to use. For bases of 11 and higher, the alphabetic characters are used as digits, with A representing 10 and so forth to Z, which represents 35. Lowercase letters can also be used. If a letter is found which is not valid for the base in use, scanning stops. There is one exception to this rule: if the base is specified as 16, and the number starts with 0x, the first two characters are ignored. `Strtol` also allows an optional leading sign.

The function `strtol` returns a long, while `strtoul` returns an unsigned long. If the number is too large, `strtol` returns `LONG_MAX` or `LONG_MIN`, depending on the sign, and `strtoul` returns `ULONG_MAX`. In both cases, `errno` is set to `ERANGE`.

See also `atof`, `atoi` and `atol`.

```
/* echo the integers in an ASCII buffer to standard out */
while (cp != NULL)
    printf("%f\n", strtol(cp, &cp, 10));
```

strtok

```
#include <string.h>
char *strtok(const char *str, const char *set);
```

The function `strtok` scans the string `str` for tokens separated by the characters contained in the null-terminated string set. The first call to `strtok` should include a pointer to a string of characters for the parameter set. If the set does not change on subsequent calls, a null pointer can be passed, avoiding the overhead of reforming the set on each call to `strtok`.

When called, `strtok` starts by forming an internal representation of the set of characters. If no set was passed, the last set passed to `strtok` is used.

Next, if `str` is not null, `strtok` skips all characters in `str` that occur somewhere in the set of characters. If all of the characters are in the set, an internal pointer is set to null and a null pointer is returned. If a character is found which is not in the set, the internal pointer is set to point to the first character not in the set, and processing continues as if `str` had been null. By this time, though, the local copy `strtok` keeps of `str` is null, even if it was not null when `strtok` was called.

To continue scanning the string, subsequent calls to `strtok` should use a null pointer for `str`. The set may be changed as needed. If `str` is null, and the internal pointer is null, `strtok` returns null. If the internal value is not null, `strtok` skips over all of the characters that are not in the set, overwriting the first character that is in the set with a null character. The internal pointer is then updated to point to the character after the null, and the old value of the internal pointer, which now points to a null-terminated token, is returned.

```

/* find and process words in the string pointed to by line */

/* form the set of separator characters */
j = 0;
for (i = 1; i < 128; ++i)
    if (!isalpha(i))
        set[j++] = i;
set[j] = '\0';

/* find and process the words */
str = strtok(line, set);
while (str != NULL) {
    process(str);
    str = strtok(NULL, NULL);
}

```

system

```

#include <stdlib.h>
int system(char *command);

```

The system function takes a null-terminated string as input and executes the string as a series of shell calls. The character `\r` can be used to separate lines if the argument consists of more than one source line. The only difference between this call and executing an EXEC file with the same commands is that an EXEC file creates a new stack frame for shell variables, while commands executed with this call work directly with the shell variables already in existence. Any command that can be typed from the shell can also be executed using this command.

As with script files, execution will halt early if any of the commands returns a non-zero error code and the `{exit}` shell variable is set to any value. If this happens, the error code is returned by the system function. Execution will also stop if the shell's exit command is used. The exit command also returns a value, and this value is returned by the system function. If the commands execute normally, a zero is returned.

Passing system a null pointer returns a zero if no shell is active, and a non-zero value if a shell is active.

```

/* catalog the disk */
system("catalog");

```

tan

```

#include <math.h>
extended tan(extended x);

```

The function tan returns the trigonometric tangent of the argument. If an error occurs, `errno` is set to `ERANGE`. The argument must be supplied in radians.

```

height = tan(x)*length;

```

tanh

```

#include <math.h>
extended tanh(extended x);

```

The function tanh returns the hyperbolic tangent of the argument.

```
n = tanh(x);
```

time

```
#include <time.h>
typedef unsigned long time_t;

time_t time(time_t *tptr);
```

The time function returns the current time as a coded integer. If the parameter tptr is not null, the time is also stored at the location indicated by tptr. If an error occurs, -1 is returned. (No error is possible in ORCA/C.) See localtime for a way to format the time in a usable fashion.

See also ctime, difftime, gmtime, mktime.

```
t = time(NULL);
```

tmpfile

See tmpnam.

tmpnam tmpfile

```
#include <stdio.h>
char *tmpnam(char *buf);
FILE *tmpfile(void);
```

The tmpnam function is used to create a file name that does not interfere with other file names on the system. If the call is successful, tmpnam returns a pointer to the file name. If buf is not null, the null-terminated file name is also stored in the character array pointed to by buf. The buffer should be at least L_tmpnam characters long. The tmpnam function can create up to TMP_MAX unique file names; under ORCA/C, TMP_MAX is 10000. Note that subsequent calls to tmpnam or tmpfile will destroy the internal copy of any file name created by tmpnam. If subsequent calls to either function will be made, and you will need the file name again, you must save the file name in your own local buffer.

The tmpfile function calls tmpnam to obtain a file name, then opens the file with flags of "w+b". The file is automatically deleted when the program stops.

```
tmpnam(myFileName);
myFile = tmpfile();
```

toascii

```
#include <ctype.h>
int toascii(int c);
```

Toascii is a macro that takes any integer argument and returns a valid ASCII character. It makes the conversion by discarding all bits except the seven least significant bits.

```
/* make sure an integer is ASCII before checking to */
/* see if it is uppercase */
if (isupper(toascii(ch))
    printf("The character is uppercase.\n");
```

toint

```
#include <ctype.h>
int toint(char c);
```

Toint is a function that returns the value of one of the hexadecimal digits. The digits consist of the characters '0' to '9', which have the values 0 to 9, respectively; and the alphabetic characters 'A' to 'F' (or their lowercase equivalents), which have the values 10 to 15, respectively.

```
/* convert a hexadecimal string to an unsigned long value */
val = 0;
len = strlen(hexString);
for (i = 0; i < len; ++i)
    val = (val << 4) | toint(hexString[i]);
```

tolower _tolower

```
#include <ctype.h>
int tolower(char c);
int _tolower(char c);
```

If the argument is an uppercase alphabetic character, the lowercase equivalent of that letter is returned. If the argument is not an uppercase alphabetic character, the original value is returned unchanged.

Tolower is a function. A faster macro called `_tolower` will perform the same operation if the argument is known to be an uppercase character, but it will also modify the argument if it is not an uppercase letter. When you know that the argument is an uppercase letter, use `_tolower` for efficiency. If the argument may not be an uppercase letter, use `tolower`.

```
/* convert a string to lowercase letters */
i = 0;
while (str[i] = tolower(str[i++]));
```

toupper _toupper

```
#include <ctype.h>
int toupper(char c);
int _toupper(char c);
```

If the argument is a lowercase alphabetic character, the uppercase equivalent of that letter is returned. If the argument is not a lowercase alphabetic character, the original value is returned unchanged.

Toupper is a function. A faster macro called `_toupper` will perform the same operation if the argument is known to be a lowercase character, but it will also modify the argument if it is not a lowercase letter. When you know that the argument is a lowercase letter, use `_toupper` for efficiency. If the argument may not be a lowercase letter, use `toupper`.

```
/* convert a string to uppercase letters */
i = 0;
while (str[i] = toupper(str[i++]));
```

ungetc

```
#include <stdio.h>
int ungetc(char c, FILE *stream);
```

The function `ungetc` returns a character to a stream opened for input. The next call that reads a character from the file will read the character placed back into the stream by this call. Subsequent calls will read characters from the file in the normal way. The character is returned by `ungetc` if the call was successful; otherwise, EOF is returned. An attempt to push back EOF is ignored, and EOF is returned.

ORCA/C implements a single character buffer. An attempt to call `ungetc` a second time before processing the first character will result in an error.

```
/* collect a token from a file */
i = 0;
for (;;) {
    ch = fgetc(myFile);
    if (isalpha(ch = fgetc(myFile)))
        name[i++] = ch;
    else {
        ungetc(ch, myFile);
        break;
    }
}
name[i] = '\0';
```

va_arg va_end va_start

```
#include <stdarg.h>
typedef char *va_list[2];

void va_start(va_list ap, ??? LastFixedParm);
??? va_arg(va_list ap, ???);
void va_end(va_list ap);
```

This collection of macros and functions allow variable argument lists to be handled from C. In a function with variable argument lists, the first call is to the macro `va_start`. This call is used to initialize pointers that will be used by `va_arg` and `va_end`. Two parameters are passed: the first is a local variable of type `va_list`, while the second is the name of the last fixed parameter in the parameter list. There must be at least one fixed parameter.

The macro `va_arg` is used to recover the values of the parameters. It takes the same argument variable initialized by `va_start` as the first parameter, and the type of the next argument as the second parameter. The value of the next parameter is returned, and the `ap` variable is incremented past the parameter.

The function `va_end` is called before leaving the function. It must be called after all of the arguments have been processed by `va_arg`. The `va_end` function cleans up the stack by removing the variable arguments from the stack.

These macros will not work if stack repair code is enabled; stack repair code is enabled by default. For an explanation of stack repair code, see Chapter 10. To turn off stack repair code, see the `optimize` pragma.

```

/* return the sum of zero or more integers */

int sum(int count, ...)

{
    va_list list;
    int total;

    va_start(list, count);
    total = 0;
    while (count--)
        total += va_arg(list, int);
    va_end(list);
    return total;
}

```

vfprintf vprintf vsprintf

```

#include <stdio.h>
int vfprintf(FILE *stream, char *format, va_list arg);
int vprintf(char *format, va_list arg);
int vsprintf(char *s, char *format, va_list arg);

```

These functions are basically the same as the functions with the same name, sans the leading *v*. The difference is that these print functions use the *vararg* facility (see *va_arg*) to specify the parameters.

write

```

#include <fcntl.h>
int write(filds, void *buf, unsigned n);

```

Up to *n* bytes are written from the buffer pointed to by *buf* to the file whose file ID is *filds*. If *n* bytes cannot be written due to lack of space, as many bytes are written as possible. The number of bytes actually written is returned. If an error occurs, a -1 is returned and *errno* is set to one of the values shown below.

If the file was not opened with the *O_BINARY* flag set, any *\n* characters in the buffer are converted to *\r* characters before being written.

Possible errors

EIO	A physical I/O error occurred.
EBADF	<i>filds</i> is not a valid file ID.
EBADF	The file is not open for writing.
ENOSPC	There was not enough room on the device to write the bytes.
ENOSPC	The <i>O_BINARY</i> flag was not set, and there was not enough room to allocate a temporary buffer for character conversion.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

See also *fwrite*.

Appendix A – Error Messages

The errors flagged during the development of a program are of three basic types: compilation errors, linking errors, and execution errors. Compilation errors are those that are flagged by the compiler when it is compiling your program. These are generally caused by mistakes in typing or simple omissions in the source code. Compilation errors are divided into four categories: those that are marked with a caret (^) on the line in which they occurred; those where the exact position of the error cannot be determined or is not well defined; those which are due to the restrictions imposed by the ORCA/C compiler; and those which are so serious that compilation cannot continue.

Link errors are reported by the linker when it is processing the object modules produced by the compiler.

Execution errors occur when your program is running. These can be detectable mistakes, such as a stack overflow with debugging enabled, or can be severe enough to cause the computer to crash, such as accessing memory in unexpected ways, as with pointer variables containing invalid addresses.

Compilation Errors

The errors listed below are all errors that the compiler can recover from.

' (' expected

The compiler expected a left parenthesis, and another token was found.

') ' expected

The compiler expected a right parenthesis, and another token was found.

' > ' expected

The compiler expected a greater than sign, and another token was found.

' ; ' expected

The compiler expected a semicolon, and another token was found.

' { ' expected

The compiler expected a left brace, and another token was found.

' } ' expected

The compiler expected a right brace, and another token was found.

'] ' expected

The compiler expected a right bracket, and another token was found.

' :' expected

The compiler expected a colon, and another token was found.

'(', '[', or '*' expected

The compiler expected one of these tokens, and another token was found.

',' expected

The compiler expected a comma, and another token was found.

'.' expected

The compiler expected a period, and another token was found.

'8' and '9' cannot be used in octal constants

Any integer constant starting with a zero is treated as an octal constant. Octal numbers do not use the digits 8 or 9.

A character constant must contain exactly one character

The character constant has zero characters, or more than one character. C character constants are limited to a single character. Use a string for more than one character. To create a character constant for the single quote mark, use `'\''`.

A function cannot be defined here

A function definition cannot appear in another function, a structure, or a union.

A value cannot be zero bits wide

A bit field was declared, and the width of the bit field was given as zero bits. Bit fields must be at least one bit wide, but no more than thirty-two bits wide.

An #if had no closing #endif

A preprocessor `#if` statement was found, but was not completed with an `#endif`.

Assignment to an array is not allowed

You cannot assign a value to an array in C, even if the value you are trying to assign is another array of the same type. Use the `memcpy` function to copy the contents of one array into another array.

Assignment to const is not allowed

An attempt has been made to assign a value to a variable that was declared as being a constant (using the `const` descriptor). Constant variables cannot be changed. You must either change the declaration or eliminate the assignment.

Assignment to void is not allowed

An attempt has been made to assign a value to void.

Auto or register can only be used in a function body

The auto storage class can only be used with variables defined locally in a function.

Bit fields must be less than 32 bits wide

ORCA/C restricts bit fields to 32 bits or less. The size of the bit field must be reduced.

Break must appear in a while, do, for or switch statement

A break statement was found, but it did not appear in a while loop, a do loop, a for loop, or a switch statement. The break statement is used to leave one of these structures, and cannot be used outside of the structure.

Case and default labels must appear in a switch statement

A case label or default label was found, but the label was not in the body of a switch statement. These types of labels cannot be used outside of a switch statement.

Compiler error

This error is generated when the compiler detects an internal problem. This can occur when another error appears in the program, and that error causes conflicting or missing information. This error sometimes appears before the message about the error that caused the problem. If this error appears in a program that has no other error messages, please report the problem. If it appears in conjunction with some other error, correcting the other error will get rid of this error, too.

Cannot redefine a macro

You cannot include a second definition for a macro unless the new definition is token-for-token identical to the original definition. For example, the following two definitions can legally appear in the same program:

```
#define EOF (-1)
#define EOF (-1)
```

The definitions

```
#define EOF (-1)
#define EOF -1
```

would, however, generate this error message. There are four ways to deal with the problem:

1. Remove all but one of the macro definitions.
2. Insure that each definition of the macro exactly matches all of the other definitions.

3. Undefine the old macro before redefining it.
4. Bracket the definition with conditional compilation commands to prevent redefinition, as in

```
#ifndef EOF
#define EOF (-1)
#endif
```

Cannot take the address of a bit field

An attempt has been made to extract the address of a field in a structure or union that is a bit field. Bit fields are not required to start on addressable boundaries, so it is not possible to take the address of a bit field.

Cannot undefine standard macros

The standard macros `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, `__STDC__` and `__ORCAC__` cannot be undefined using the `#undef` command.

Comp data type is not supported by the 68881

ORCA/C uses SANE to perform calculations on float, double, extended and comp numbers. When you use the `#pragma float` command to instruct the compiler to use the 68881 floating-point card, comp numbers can no longer be used, since the 68881 does not support the comp data type. This error message will flag any attempt to use comp numbers with the 68881 card.

Continue must appear in a while, do or for loop

A continue statement was found, but it did not appear in a while loop, a do loop, or a for loop. The continue statement is used to jump to the end of one of these structures, and cannot be used outside of the structure.

Digits expected in the exponent

A floating-point constant was coded with an e designator for the exponent, but no exponent was found. Either code an exponent or remove the e exponent designator.

Duplicate case label

Two identical values have been used for a case label in the same switch statement. Each of the case labels must be unique.

Duplicate label

Two labels with the same name have been defined in the same function. Each statement label must be different from the other statement labels in the function.

Duplicate symbol

Two symbols have been defined in the same overloading class, and the symbols were given the same name. For example, a single function cannot define the variable `i` as an int, and then redefine it as a float variable. One of the names must be changed so that all of the names are unique.

'dynamic' expected

The segment statement allows you to define the segment as dynamic by placing the identifier dynamic in the segment statement. This is the only token allowed in that spot. The compiler found a different token. You must remove it, or change it to dynamic.

The else has no matching if

An else statement appeared in a program, but no matching if statement was found. This can happen when the if statement is omitted entirely, when two else clauses are used with a single if statement, or when compound statements have been used improperly.

End of line expected

Extra characters appeared in a preprocessor command.

Expression expected

A token that cannot be used to start an expression was found in a position where the compiler expected to find an expression.

Expression syntax error

An expression has not been formed correctly. This can occur when an operand or operation has been omitted, or when the operands and operations are not given in the correct order.

Extern variables cannot be initialized

When a variable is declared as extern, the compiler does not create the space for the variable, it only notes that such a variable exists in some other separately compiled module or library. A variable cannot be initialized unless space is created to hold the initial value. You can, of course, initialize the variable in the single location where it is declared.

File name expected

The #include, #append and #pragma keep directive expect an operand of a file name. None was found.

File name is too long

The C compiler limits file names to 255 characters. A longer file name was found in a #include, #append or #pragma keep directive.

Functions cannot return functions or arrays

A function can return a pointer to a function or array, but a function cannot return another function or a whole array.

Further errors suppressed

When more than eight errors occur on a single line, the compiler reports the first seven, then gives this error message rather than continuing with more errors.

Identifier expected

The C language requires that an identifier appear at the indicated place.

Illegal character

A character which is not used in the C language appeared outside of a comment or string constant.

Illegal math operation in a constant expression

An attempt has been made to perform an operation that is not allowed in a constant expression.

Illegal operand for the indirection operator

An attempt has been made to use the selection operator (.) or the indirection operator (*) in a context in which it is not allowed.

Illegal operand in a constant expression

An operand that could not be resolved as a constant at compile time was found in an expression that must yield a constant value.

Illegal type cast

An attempt was made to cast a variable to an illegal type.

Illegal use of forward declaration

An attempt has been made to use a forward declaration in a way that is not allowed. In general, no use can be made of a forward declaration except to declare a pointer to the type.

Implementation restriction: run-time stack space exhausted

In some cases, the compiler can detect at compile time that a stack overflow will occur when a program is executed; when this happens, this error appears. This is generally caused by defining an auto array that is larger than 32K bytes. You can avoid the problem by changing the array to a static array or using a pointer to a dynamically allocated array.

Implementation restriction: string space exhausted

Each function has a buffer that is used for string constants and a few values stored when debug code is created. This buffer is limited to 8000 bytes. You must turn debugging off or split the function into two or more smaller functions to avoid overflowing the buffer.

Implementation restriction: too many local labels

The compiler generates internal labels to handle the flow of control statements. The number of labels that can appear in a single function is sufficient for about 2000 to 3000 lines of code, although the number can vary greatly based on coding style. If the limit is exceeded, you must split the function into two or more functions to avoid the limitation.

Incorrect operand size

Many 65816 operation codes require an operand of a specific size. For example, operands using indirect addressing generally require an expression value in the range 0 to 255. If the expression in the operand results in a value with an inappropriate size, and the operand cannot be legally converted to a correct size, this error will appear.

Incorrect number of macro parameters

A macro was invoked, and too few or too many parameters were supplied. Each of the parameters defined in the macro definition must be supplied exactly one time each time the macro is used.

Integer constant expected

The inline directive, used for creating tool interface files, requires an integer constant and a long integer constant for the two operands. The #pragma nda directive also requires two integer constants. Something other than an integer constant was used.

Integer constants cannot use the f designator

The f or F designator, used to force a floating-point constant to be a float rather than double value, has been used on an integer constant (such as a character or octal value). The f designator can only be used of floating-point values or decimal integer values.

Integer overflow

An integer constant with a value over 2147483647, or an unsigned constant with a value over 4294967295 appeared in the program. The integer is too large for ORCA/C to deal with.

Invalid operand

An operand was used with an instruction that does not support the operand. For example, this error would appear if you code absolute indexed addressing with a jsf instruction.

Invalid storage type for a parameter

Function parameters can have a storage class of auto or register. Some other storage type was specified for a parameter; it must be changed.

Keep must appear before any functions

The #pragma keep directive must appear before the first function definition. Move it to the top of the source file.

L-value required

Assignments and certain operators require an l-value. Something other than an l-value was used in one of these locations.

Lint: missing function type

The lint pragma has been used with bit 1 set, and the function has been declared or defined without an explicit return type.

Lint: parameter list not prototyped

The lint pragma has been used with bit 2 set, and a function has been declared or defined without a prototyped parameter list.

Lint: undefined function

The lint pragma has been used with bit 0 set, and the function has been used, but not declared or defined before the use.

Lint: unknown pragma

The lint pragma has been used with bit 3 set, and a pragma has been found which the compiler does not recognize.

No end was found to the string

A string constant was started, but a closing quote mark was not found.

No matching '?' found for this ':' operator

The compiler found the start of a trinary operator, but could not find the ? character that separates the two expressions.

Only one default label is allowed in a switch statement

A default label has been found in the body of a switch statement for which another default label has already been found. Only one default label can appear in any switch statement; one of them must be removed.

Only one #else may be used per #if

A #else command was found in the body of an #if statement that already has a #else clause. One of the #else statements must be removed or changed to an #elif command.

Only parameters or types may be declared here

An attempt was made to declare a variable in the declarations that follow the function header and precede the function body. Only parameters can be declared in this location.

Operand expected

The compiler expected an operand (variable, parenthesized expression, etc.) during a function evaluation, but found some other token.

Operand syntax error

An operand was used for a 65816 instruction, but the miniassembler could not compile the operand. The error message will point to the problem area.

Operation expected

The compiler expected an operation (+, *, etc.) during a function evaluation, but found some other token.

Pascal qualifier is only allowed on functions

The pascal qualifier can only be used on a function declaration or definition. This error appears when it is used on a type or variable.

Pointer initializers must resolve to an integer, address or string

The value provided in the initializer for a pointer must be an integer, an address, or a string, and the type must be correct for the pointer.

Real constants cannot be unsigned

The u or U designator, used to indicate that an integer is unsigned, has been used on a real constant. Real constants cannot be unsigned.

Statement expected

A statement was expected in the body of a function or compound statement, but a token has been found which cannot be used to start a statement.

String constant expected

String constants are required when the name of a CDA, NDA or segment is expected by the compiler. Something other than a string constant was found in one of these situations.

String too long

ORCA/C limits each individual string constant to 4000 characters. A string constant longer than 4000 characters has appeared in the source file.

Switch expressions must evaluate to integers

The operand of a switch statement must evaluate to one of the forms of integer (including characters). The function evaluates to some other type, and must be changed.

The array size could not be determined

In some instances, arrays sizes must be given, and in others, the exact size can be left unspecified. This error notes that an attempt has been made to leave the size of an array unspecified when a specific size is required.

The & operator cannot be applied to arrays

An array name is an address; applying the & operator to an address is not allowed.

The function's type must match the previous declaration

A function declaration or definition has been found for a function that has been declared earlier. This is allowed, but the type returned by the function must match the earlier declaration.

The last compound statement was not finished

The function body contains the start of a compound statement, but the source file ended before the compound statement was completed.

The last do statement was not finished

The function body contains the start of a do statement, but the function ended before the do statement was completed.

The last for statement was not finished

The function body contains the start of a for statement, but the function ended before the for statement was completed.

The last else clause was not finished

The function body contains the start of an else clause for an if statement, but the function ended before the else clause was completed.

The last if statement was not finished

The function body contains the start of an if statement, but the function ended before the if statement was completed.

The last switch statement was not finished

The function body contains the start of a switch statement, but the function ended before the switch statement was completed.

The last while statement was not finished

The function body contains the start of a while statement, but the function ended before the while statement was completed.

The number of array elements must be greater than 0

An array was declared with 0 elements. All arrays must have one or more elements.

The number of parameters does not agree with the prototype

A function call had been made, but the number of parameters supplied in the function call does not match the number of parameters declared by the function prototype. Exactly one parameter must be supplied for each parameter in the function prototype, and the types of the parameters in the call must match the types of the parameters in the function prototype.

The operation cannot be performed on operands of the type given

Not all operations can be used with all types. An attempt has been made to use an operation with an operand of an illegal type. For example, this error would occur if you attempt to use a bit manipulation operator on a floating-point number.

The selected field does not exist in the structure or union

A field has been selected from a structure or union using the `->` or `.` selection operators, but the field does not exist.

The selection operator must be used on a structure or union

The `.` or `->` selection operator has been used on a variable that is not a structure or union.

The structure has already been defined

An attempt has been made to redefine a structure or union. Once a set of fields has been associated with a structure or union, the structure or union tag can be used to define variables, but the fields must not be redefined.

There is no #if for this directive

A `#else` or `#elif` command has been used, but cannot be associated with a `#if` command.

Token merging produced an illegal token

The `##` token merging operator was used in a macro, but the tokens merged did not result in a legal token. For example, merging `-` and `1` produces the string `-1`, but this string is not a legal token. (The characters `"-1"` form two separate tokens, the unary subtraction operator and the signed integer one.)

Too many initializers

Too many initializers have been specified for an array or structure.

Type conflict

Incompatible types have been used with one of the operators in an expression, or an attempt has been made to pass a parameter with an incorrect type to a prototyped function.

Type expected

The compiler expected a type in a declaration, but none was found.

Undeclared identifier

An identifier has been used, but it has not been declared. Except for some limited cases where a function may be called before it is used, all identifiers must be declared before they are used.

Unidentified operation code

An operation code appeared in the miniassembler, but it was not a legal 65816 operation code. Be sure the name of the operation code has not been redefined with a macro. If the error appears in an operand field, make sure the operation code requires an operand. If the error appears in a label field, be sure the label is followed by a colon.

Unions cannot have bit fields

Bit fields can only be used in structures.

Unknown cc= option on command line

There is a cc= command line option that the compiler does not recognize, possibly due to a malformed option.

Unknown preprocessor command

The preprocessor has encountered a command it does not recognize. The command must be removed.

'while' expected

A while clause was expected at the end of a do statement, but none was found.

You cannot initialize a parameter

Parameters cannot be initialized.

You cannot initialize a type

Only variables can be initialized, not an entire class of variables represented by a type.

You must initialize the individual elements of a struct, union, or non-char array

This error generally occurs when an initializer has been mistyped. When initializing a complex type, it is necessary to specify the values of each element or field, except for some cases when the compiler will finish initializing an area with zeros.

Terminal Compilation Errors

A terminal error encountered during compilation will abort the compile.

Error purging *file*

An error occurred when the compiler was releasing a file. This could be a source file or an object file created by the compiler. This error can occur due to corrupted memory, a full disk, or an error while writing the file to disk.

Error reading *file*

A file read error occurred while reading the contents of one of the source files. If an exit is made to the editor, the location may not be correct. This error is reported by the operating system; it is generally caused by a corrupted disk or a disk with a bad block.

ORCA/C requires version 1.1 or later of the shell

The ORCA/C compiler has been executed from APW 1.0 or ORCA/M 1.0. You must update the shell.

Out Of Memory

There is not enough memory to compile the program.


Source files must have a file type of SRC

An attempt has been made to open a source file with a #include or #append directive, but the file was not a source file.

Terminal compiler error

An internal error has occurred that the compiler cannot recover from. This error should be accompanied by some other error. When the other errors have been corrected, this error should go away. If it does not, please contact the Byte Works.

User Termination

The user has entered the two-key abort command . (hold down the open-Apple key and then type a period). This does no harm to the program, it merely terminates compilation.

You cannot change languages from an included file

An attempt has been made to open another SRC file with a #append statement, but the file that contains the #append statement is an included file. If you are trying to do a multi-lingual compile, move the #append statement to the end of the C file that is passed to the compiler. If the file is supposed to be a C source file, change the language stamp using the Languages menu or the shell CHANGE command.

You cannot change languages with an include directive

An attempt has been made to open another SRC file with a #include statement, but the file is not a C source file. If you are trying to do a multi-lingual compile, use the #append statement. If the file is supposed to be a C source file, change the language stamp using the Languages menu or the shell CHANGE command.

Appendix B – ANSI C

This appendix summarizes the differences between ORCA/C and ANSI C. For a complete description of the feature, see the manual.

- In ORCA/C, the following identifiers are reserved words.

reserved word	use
asm	Used to create assembly language functions and to imbed assembly language code within a C function.
comp	Used for SANE comp numbers.
extended	Used for SANE extended numbers.
inline	Used to declare functions called at a specific location with a given value in the X register. On the Apple IIGS, this is used to declare tool header files.
pascal	Instructs the compiler to use Pascal calling conventions.
segment	Used to place functions in a specific load segment.

- The #append directive works like #include, but does not return to the original source file.
- There are several pragmas in ORCA/C. Pragmas are allowed by ANSI C, but none are required.
- // comments are allowed.
- A small number of ANSI C library functions are not included. This includes some variable argument I/O functions, a time function, and multi-character support.
- The formatted I/O functions (e.g. printf) use %p as a format specifier for Pascal strings, not pointers.

Appendix C – ORCA/C on the Apple IIGS

This appendix summarizes the differences between MPW IIGS ORCA/C and the native implementation of ORCA/C on the Apple IIGS. For a complete description of any feature mentioned here, see the respective manuals.

MPW IIGS ORCA/C is a port of ORCA/C 2.0.3 for the Apple IIGS. Other than the differences listed in this appendix, the two compilers should be identical—creating identical code from identical source, and even showing identical bugs.

In general, new features you see here will eventually be available in ORCA/C for the Apple IIGS. The one big difference is that the MPW based compiler relaxes the ANSI standard whenever the compiler offers an option to enforce or ignore the standard, while the native Apple IIGS compiler defaults to strict compliance with the standard. An explicit `#pragma ignore` at the start of each source file will cause both compilers to behave the same way.

- The inline directive (generally used for creating tool header files) has been extended in the MPW IIGS compiler to allow calling functions that use C calling conventions.
- The MPW IIGS compiler allows enumerated constants as the first term of an inline directive.
- When a `#line` directive is used, error messages in the MPW IIGS compiler show the source file and line number set by the `#line` directive, not the source file and line number of the physical file.
- All of the bits in the `#pragma ignore` directive default to 1 in the MPW IIGS compiler, and to 0 in the Apple IIGS compiler. This has the effect of relaxing the ANSI standard when possible on the MPW IIGS compiler, and enforcing the ANSI standard strictly on the Apple IIGS compiler.
- The MPW IIGS compiler accepts multi-byte character constants.
- The MPW IIGS compiler ignores spurious tokens after a `#endif`. (See `#pragma ignore`.)
- The MPW IIGS compiler allows `//` comments. (See `#pragma ignore`.)
- The MPW IIGS compiler supports the extended Apple character set (e.g. ©, ü, etc.). Characters that are obvious graphical equivalents of alphabetic characters, like ç and î, are allowed in identifiers. Characters that are abbreviations for mathematical symbols are supported as equivalents to the C tokens; for example, ≠ can be used instead of !=.

Special Characters

~ operator 84, 88
^ operator 84
! operator 83, 88
!= operator 83
#append **25**
#define **21**
#elif **28**
#else **27**
#endif , 27
#error 29
#if **26-27**
 expressions 27
 nesting 27
#ifdef **28**
#ifndef **28**
#include **24-25**, 39, 47
#line 29
#pragma 30
 cda **30**
 cdev **30**
 databank **31**
 debug **31**
 expand **32**
 float **33**
 ignore **33**
 keep **34**
 lint **34**
 memorymodel **35**
 nba **36**
 nda **36**
 noroot **37**
 optimize **37**
 path **39**
 rtl **39**
 stacksize **40**
 toolparms **40**
 xcmd **41**
#undef **23**, 24
% operator 80
%= operator 85
& operator 83, 86
&& operator 83
&= operator 85
* operator 80, 86, 88
*= operator 85
+ operator 79, 81, 88
++ operator 78, 81
+= operator 85
, operator 86
- operator 79, 81, 88
-- operator 78, 81
-= operator 85

/ operator 80
/= operator 85
68881 33, 43, 88
< operator 83
<< operator 84, 88
<<= operator 85
<= operator 83
^= operator 85
= operator 84, 87
== operator 83
> operator 83
>= operator 83
>> operator 84, 88
>>= operator 85
? operator 87, 88
_exit 116
_tolower 158
_toupper 158
__DATE__ 24
__FILE__ 24
__LINE__ 24
__ORCAC__ 24
__STDC__ 24, 26
__TIME__ 24
__VERSION__ 24
| operator 83
|= operator 85
|| operator 83

A

abort 116
abs 107
acos 107
alias command 46
ANSI C 24, 175, 177
APW 46, 173
APW C 28, 113
arctan 108
argc 45
argv 45
arrays 35, **54-55**, 61, 75-76, 84, 91
 as pointers 75
 indexing 75
 initialization 60
 memory requirements 55
 multiple subscripts 54
 size limits 55
 storage 55
ASCII character set 15
asctime 113
asin 108
asm statement 66, 100
assembly language 7, 25, 43, 46, 66, 100

- accessing global variables 9
- calling C procedures and functions 10
- passing parameters 8
- returning function values 8
- assert 108
- assignment conversions 87
- assignments
 - compound 85
 - multiple 86
- atan 108
- atan2 108
- atexit 109
- atof 109
- atoi 109
- atol 109
- auto storage class 49

B

- binary conversion rules 88
- bit fields 57-58
- break statement 94, 98
- bsearch 110

C

- c2pstr 110
- calloc 140
- case label 97, 98
- case sensitivity 11, 72
- cast 89
- CDev 30
- ceil 111
- cfree 127
- char 43, 51, 89
- character constants 15, 17
- character set **11**
- characters
 - formatting 124
 - scanning 130
- chmod 111
- clalloc 140
- classic desk accessories 30
- clearerr 118
- clock 112
- close 112
- command line 45
- comments 19
- comp 44, 51
- compilation errors 161
- compiler directives 30
- compound statement 93
- conditional compilation 26
- const 52, 55

- constants 11, 53, 75
 - character 15
 - floating-point 18
 - integer 13, 14, 15
 - long integer 14, 15
 - strings 16
 - unsigned integer 14, 15
 - unsigned long integer 14, 15
- continuation lines 13, 16, 21
- continue statement 94, 98
- control characters 17
- control panel 30
- conversions 87
- cos 112
- cosh 113
- crashes 38
- creat 113
- ctime 113

D

- data formats 43
- databank 31
- date 24
- debugger 31
- declarations 93
- default label 97, 98
- defined operator 27
- difftime 114
- direct page 40
- direct selection 77
- div 114
- do statement 95, 98
- double 43, 89, 90
- dup 115
- dynamic segments 100

E

- EDOM 107, 108, 139, 143
- else statement 96
- enum 53, 63
- enumerations
 - initialization 60
- EOF 115
- ERANGE 113, 117, 127, 139, 143, 149, 150, 154, 155, 156
- errno **116**, 118, 121, 154, 155
- errors 29, 32, 33, 116
- escape sequences **17**
- exit 116
- exp 117
- expressions 53, 73, 94
 - syntax 73

extended 44, 89, 90
 extended characters 11
 extern 49, 61

F

fabs 107
 fclose 117
 fcntl 117
 feof 118
 ferror 118
 fflush 118
 fgetc 118
 fgetpos 119
 fgets 119
 file names 24, 29
 float 43, 89, 90
 floating-point 33, 43, 51, 88, 89
 constants 18
 formatting 124
 initialization 60
 scanning 130
 floor 111
 fmod 120
 fopen 120
 for statement 95, 98
 fprintf 121
 fputc 126
 fputs 126
 fread 126
 free 127
 freopen 120
 frexp 127
 fscanf 127
 fseek 132
 fsetpos 119
 ftell 132
 function prototypes 68
 functions 51, 52, 65, 76, 88, 91, 94
 asm 66
 calling through a pointer 76
 declaration 65
 extern 66
 inline 66
 parameters - see parameters 67
 return type 62, 65
 return value 98
 returning a value 72
 static 66
 fwrite 133

G

getc 118

getchar 118
 getenv 133
 gets 119
 gmtime 138
 goto statement 94, 96

H

header files 25
 hexadecimal 15, 17, 75, 100, 123, 129, 138,
 158
 HUGE_VAL 154
 HyperCard 41
 HyperStudio 36

I

I/O redirection 46
 identifiers 11
 case sensitivity 11
 length 11
 if statement 96
 indirect selection 77
 initializers **59-61**, 93
 Inits 39
 inline 66
 Innovative Systems 43
 int 43, 51, 89
 integers 43, 51, 89, 90
 constants 13, 14, 15
 formatting 123
 initialization 59
 scanning 129
 storage 43
 interface files 46
 isalnum 133
 isalpha 134
 isascii 134
 iscntrl 134
 iscsym 135
 iscsymf 135
 isdigit 135
 isgraph 136
 islower 136
 isodigit 136
 isprint 136
 ispunct 137
 isspace 137
 isupper 137
 isxdigit 138

K

keyboard input 119

L

- l-values 74
- labels 96
 - scope 63
- labs 107
- large memory model 35, 99
- ldexp 127
- ldiv 114
- libraries 35, 47, 49, 103
 - see also toolbox
- line numbers 24, 29
- link errors 161
- linker 161
- localtime 138
- log 139
- log10 139
- long 43, 51, 89
- long integers
 - constants 14, 15
- longjmp 148
- lseek 139

M

- macro stringization 23
- macros 21-24, 25
 - defined operator 27
 - expanding 32
 - in if statements
 - nesting 23
 - parameters 22
 - predefined 24
 - scope 63
- main 45
- malloc 140
- memchr 140
- memcmp 141
- memcpy 141
- memmove 141
- memory
 - see also large memory model
 - see also small memory compiler
 - see also small memory model
- memory manager 140
- memory model 99
- memset 141
- mktime 138
- mlalloc 140
- modf 142
- MPW IIGS Assembler 7
- multiple assignments 86
- multiple languages 25

N

- New Button Actions 36
- new desk accessories 36
- newline character 17, 120
- null character 16, 18
- null statement 94

O

- object files 34, 46
- octal 14, 15, 17, 18, 75, 100, 123, 129, 136, 162
- offsetof 142
- open 142
- operator precedence 74, 78
- optimizer 37
- ORCA/M 46, 173
- ORCA/Pascal 46
- overloading classes 63

P

- p-strings 17, 18, 110
- p2cstr 110
- parameters 32, 34, 62, 67-70, 88, 159
 - checking 68
 - function prototypes 68
 - passing 8, 70
 - variable length lists 69
 - void prototypes 68
- parentheses 78
- Pascal 46
- pascal qualifier 41, **72**, 169
- perror 116
- pointers 44, 55-56, 84, 89, 90, 91
 - initialization 60
- pow 143
- pragma 30
- precompiled headers 25
- preprocessor 21-41
- preprocessor macros
 - see macros
- printf 121
- profiler 32
- putc 126
- putchar 126
- puts 126

Q

- qsort 144

R

raise 144
 rand 144
 range checking 31
 read 145
 realloc 145
 register 50
 relalloc 145
 remove 146
 rename 146
 reserved symbols **12**
 reserved words **12**
 return key 120
 return statement 94, 98
 rewind 132
 root file 37
 run-time errors 32, 116, 161
 run-time stack 35, 40

S

SANE 18, 33, 40, 43, 44, 88, 175
 scanf 127
 scope 62-64, 93
 segment statement 35, 99
 separate compilation 46, 47, 49, 66
 setbuf 147
 setjmp 148
 setvbuf 147
 shell
 variables 46
 shell window 46
 short 43, 51, 89
 signal 148
 signed 43
 sin 149
 sinh 149
 sizeof operator 82, 91
 small memory compiler 26, 38
 small memory model 35, 99
 source-level debugger 31
 sprintf 121
 sqrt 150
 srand 144
 sscanf 127
 stack overflow 31
 stack repair code 32, 38, 69, 159
 stack size 31
 standard input 119
 standard libraries 24
 see also libraries
 standard output 126
 startdesk 40

startgraph 40
 statements 94
 static 50
 static segments 99
 stderr 150
 stdin 150
 stdout 150
 storage class 49
 strcat 151
 strchr 151
 strcmp 152
 strcpy 152
 strcspn 153
 strerror 116
 string constants 16, 17
 trigraphs 12
 strings **16**
 continuation 16
 formatting 124
 in expressions 16
 maximum length 17
 p-strings 18
 scanning 130, 131
 strlen 153
 strncat 151
 strncmp 152
 strncpy 152
 strpbrk 153
 strpos 151
 strrchr 151
 strpbrk 153
 strrpos 151
 strspn 153
 strstr 154
 strtod 154
 strtok 155
 strtol 154
 strtoul 154
 structures 35, **56-58**, 63, 77, 84
 accessing elements 57
 assigning 57
 in functions 57
 initialization 61
 size 56
 switch statement 94, 97, 98
 sym files 25
 SysCharErrout 103
 SysCharOut 103
 SysIOShutdown 103
 SysIOStartup 104
 SysKeyAvail 104
 SysKeyin 104
 SysPutback 104
 system 156

SystemEnvironmentInit 104
SystemError 105
SystemErrorLocation 105
SystemMinStack 105
SystemMMShutDown 106
SystemPrintError 106
SystemQuitFlags 106
SystemQuitPath 106
SystemSANEInit 106
SystemSANEShutDown 106
SystemUserID 107

T

tan 156
tanh 156
terminal errors 173
terms 74
time 24, 157
tmpfile 157
tmpnam 157
toascii 157
toint 158
token merging 24
tokens **11**, 19, 27
tolower 158
toolbox 24, 40, 47, 66
toupper 158
trace backs 32
trigraphs **12**
type casting 89
type specifier **51-53**
typedef 50, 52
types 43

U

unary conversion rules 88
ungetc 159
unions 35, **59**, 63, 77, 84
 accessing elements 59
 in functions 59
 initialization 61
 size 59
unsigned 43, 89
unsigned integers
 constants 14, 15
unsigned long integers
 constants 14, 15

V

variable scope
 see scope

variables 51, 75
va_arg 69, 159
va_end 159
va_start 159
vfprintf 160
void 52, 61, 91
volatile 52, 55
vprintf 160
vsprintf 160
W

while statement 94, 95, 96, 98
white space 19
write 160

X

XCFN 41
XCMD 41