R040-02

VOS C Language Manual

Stratus Computer, Inc.

Notice

Manual Name: *VOS C Language Manual*

Part Number: R040
Revision Number: 02
VOS Release Number: 11.0
Printing Date: May 1991

Stratus Computer, Inc.
55 Fairbanks Blvd.
Marlboro, Massachusetts 01752

Preface

The *VOS C Language Manual (R040)* documents the VOS implementation of the C language. The manual is intended as a reference document, not a tutorial.

Audience

This manual is intended for experienced application programmers who may or may not be knowledgeable C programmers.

Before using the *VOS C Language Manual (R040)*, you should be familiar with the *Introduction to VOS (R001)* manual and the *VOS Reference Manual (R002)*.

Revision Information

This manual is a revision. For information on which release of the software this manual documents, see the Notice page.

This revision incorporates many corrections and changes, including the following new functionality:

- `signed` keyword
- `volatile` and `const` type qualifiers
- `$shortmap` and `$longmap` alignment specifiers
- function prototypes
- new version of the VOS C library functions
- new predefined macro names
- `#pragma`| preprocessor directive and new or modified `#pragma` options.

For Release 11.0, the version of the VOS C library functions described in this manual is not the version that is associated with the standard |stdio.h| header file or the standard object library, the `c_object_library`. To use the new version of the library functions (the version described in this manual), you must follow the procedure explained in the ``Using the New Version of the VOS C Library Functions'' section in Chapter 11.

The manual now documents the following functions and macros, some of which are new and some of which existed in earlier versions of the VOS C library.

```
access                    memmove
alloca                    offsetof
chdir                     open
close                     putw
creat                     qsort
fdopen                    read
```

```
fgetpos                   s$c_get_portid
fileno                    s$c_get_portid_from_fildes
fsetpos                   sleep
getw                      strstr
isatty                    strtoul
lockf                     _tolower
lseek                     _toupper
memccpy                   write
```

The functionality of the following functions has been expanded or modified significantly.

```
fopen
freopen
fseek
ftell
```

The manual now documents the following built-in functions.

```
$charcode                 $lockingshiftselector
$charwidth                $shift
$iclen                    $singleshiftchar
$lockingcharcode          $substr
$lockingshiftintroducer   $unshift
```

In addition, the manual has been reorganized and expanded to provide a more comprehensive description of VOS C.

Notation Conventions

This manual uses the following notation conventions.

- Italics introduces or defines new terms. For example:

  The *master disk* is the name of the member disk from which the module was booted.

- Boldface emphasizes words in text. For example:

  Every module **must** have a copy of the `module_start_up.cm` file.

- Monospace represents text that would appear on your terminal's screen (such as commands, subroutines, code fragments, and names of files and directories).
  For example:

  ```
  change_current_dir (master_disk)>system>doc
  ```

- Monospace italic represents terms that are to be replaced by literal values. In the following example, the user must replace the monospace-italic term with a literal value.

  `list_users -module` *`module_name`*

- Monospace bold represents user input in examples and figures that contain both user input and system output (which appears in monospace). For example:

```
display_access_list system_default

%dev#m1>system>acl>system_default

w  *.*
```

Key Mappings for VOS Functions

VOS provides several command-line and display-form functions. Each function is mapped to a particular key or combination of keys on the terminal keyboard. To perform a function, you press the appropriate key(s) from the command-line or display form. For an explanation of the command-line and display-form functions, see the manual *Introduction to VOS (R001)*.

The keys that perform specific VOS functions vary depending on the terminal. For example, on a V103 ASCII terminal, you press the Shift and F20 keys simultaneously to perform the INTERRUPT function; on a V105 PC/+ 106 terminal, you press the 1 key on the numeric keypad to perform the INTERRUPT function.

Note: Certain applications may define these keys differently. Refer to the documentation for the application for the specific key mappings.

The following table lists several VOS functions and the keys to which they are mapped on commonly used Stratus terminals and on an IBM PC® or compatible PC that is running the Stratus PC/Connect-2 software. (If your PC is running another type of software to connect to a Stratus host computer, the key mappings may be different.) For information about the key mappings for a terminal that is not listed in this table, refer to the documentation for that terminal.

VOS Function	V103 ASCII	V103 EPC	IBM PC or Compatible PC	V105 PC/+ 106	V105 ANSI
CANCEL	F18	* †	* †	5 † or * †	F18
CYCLE	F17	F12	Alt-C	4 †	F17
CYCLE BACK	Shift-F17	Shift-F12	Alt-B	7 †	Shift-F17
DISPLAY FORM	F19	– †	– †	6 † or – †	F19 or Shift–Help
HELP	Shift-F8	Shift-F2	Shift-F2	Shift-F8	Help
INSERT DEFAULT	Shift-F11	Shift-F10	Shift-F10	Shift-F11	F11
INSERT SAVED	F11	F10	F10	F11	Insert_Here
INTERRUPT	Shift-F20	Shift-Delete	Alt-I	1 †	Shift-F20
NO PAUSE	Shift-F18	Shift– * †	Alt-P	8 †	Shift-F18

† Numeric-keypad key

Syntax Notation

A *language format* shows the syntax of a VOS **LANGUAGE_NAME** statement, portion of a statement, declaration, or definition. When VOS **LANGUAGE_NAME** allows more than one format for a language construct, the documentation presents each format consecutively. For complex language constructs, the text may supply additional information about the syntax.

The following table explains the notation used in language formats.

The Notation Used in Language Formats

Notation	**Meaning**
`element`	Required element.
`element``...`	Required element that can be repeated.
$\left\{\texttt{element\_1 element\_2}\right\}$	List of required elements.
$\left\{\texttt{element\_1 element\_2}\right\}\texttt{...}$	List of required elements that can be repeated.
$\left\{\begin{matrix}\textit{element\_1}\\ \textit{element\_2}\end{matrix}\right\}$	Set of elements that are mutually exclusive; you must specify one of these elements.
$\left[\texttt{element}\right]$	Optional element.
$\left[\texttt{element}\right]\texttt{...}$	Optional element that can be repeated.
$\left[\texttt{element\_1 } \textit{element\_2}\right]$	List of optional elements.
$\left[\texttt{element\_1 } \textit{element\_2}\right]\texttt{...}$	List of optional elements that can be repeated.
$\left[\begin{matrix}\textit{element\_1}\\ \textit{element\_2}\end{matrix}\right]$	Set of optional elements that are mutually exclusive; you can specify only one of these elements.
Note: Dots, brackets, and braces are not literal characters; you should **not** type them. Any list or set of elements can contain more than two elements. Brackets and braces are sometimes nested.	

In the preceding table, *`element`* represents one of the following VOS **LANGUAGE_NAME** language constructs.

- reserved words (which appear in monospace)
- generic terms (which appear in monospace italic) that are to be replaced by items such as expressions, identifiers, literals, constants, or statements
- statements or portions of statements

A reserved word has special meaning for the compiler; you cannot define a reserved word as an identifier. A keyword is a reserved word that is underlined in a language format. The compiler uses keywords to generate the code. Reserved words that are not underlined enhance readability but have no effect on compilation.

The elements in a list of elements must be entered in the order shown, unless the text specifies otherwise. An element or a list of elements followed by a set of three dots indicates that the element(s) can be repeated.

The following example shows a sample language format.

In examples, a set of three vertically aligned dots indicates that a portion of a language construct or program has been omitted. For example:

Format for Commands and Requests

Stratus manuals use the following format conventions for documenting commands and requests. (A *request* is typically a command used within a subsystem, such as `analyze_system`.) Note that the command and request descriptions do not necessarily include each of the following sections.

`name`

The name of the command or request is at the top of the first page of the description.

Privileged

This notation appears after the name of a command or request that can be issued only from a privileged process. (See the online glossary, which is located in the file `>system>doc>glossary.doc`, for the definition of privileged process.)

Purpose

Explains briefly what the command or request does.

Display Form

Shows the form that is displayed when you type the command or request name followed by `-form` or when you press the key that performs the `DISPLAY FORM` function. Each field in the form represents a command or request argument. If an argument has a default value, that value is displayed in the form. (See the online glossary for the definition of default value.)

The following table explains the notation used in display forms.

The Notation Used in Display Forms

Notation	Meaning
████████	Required field with no default value.
█	The cursor, which indicates the current position on the screen. For example, the cursor may be positioned on the first character of a value, as in `all`.

Notation	**Meaning**
`current_user` *`current_module`* *`current_system`* *`current_disk`*	The default value is the current user, module, system, or disk. The actual name is displayed in the display form of the command or request.

Command-Line Form

Shows the syntax of the command or request with its arguments. You can display an online version of the command-line form of a command or request by typing the command or request name followed by `-usage`.

The following table explains the notation used in command-line forms. In the table, the term *multiple values* refers to explicitly stated separate values, such as two or more object names. Specifying multiple values is **not** the same as specifying a star name. (See the online glossary for the definition of star name.) When you specify multiple values, you must separate each value with a space.

The Notation Used in Command-Line Forms

Notation	**Meaning**
`argument_1`	Required argument.
`argument_1...`	Required argument for which you can specify multiple values.
$\left\{ \begin{array}{l} \mathit{argument\_1} \\ \mathit{argument\_2} \end{array} \right\}$	Set of arguments that are mutually exclusive; you must specify one of these arguments.
$\left[\mathtt{argument\_1} \right]$	Optional argument.
$\left[\mathit{argument\_1} \right]$...	Optional argument for which you can specify multiple values.
$\left[\begin{array}{l} \mathit{argument\_1} \\ \mathit{argument\_2} \end{array} \right]$	Set of optional arguments that are mutually exclusive; you can specify only one of these arguments.
Note: Dots, brackets, and braces are not literal characters; you should **not** type them. Any list or set of arguments can contain more than two elements. Brackets and braces are sometimes nested.	

Arguments

Describes the command or request arguments. The following table explains the notation used in argument descriptions.

The Notation Used in Argument Descriptions

Notation	Meaning
CYCLE	There are predefined values for this argument. In the display form, you display these values in sequence by pressing the key that performs the `CYCLE` function.
Required	You cannot issue the command or request without specifying a value for this argument. If an argument is required but has a default value, it is not labeled **Required** since you do not need to specify it in the command-line form. However, in the display form, a required field must have a value—either the displayed default value or a value that you specify.
(Privileged)	Only a privileged process can specify a value for this argument.

Explanation
Explains how to use the command or request and provides supplementary information.

Error Messages
Lists common error messages with a short explanation.

Examples
Illustrates uses of the command or request.

Related Information
Refers you to related information (in this manual or other manuals), including descriptions of commands, subroutines, and requests that you can use with or in place of this command or request.

Format for Subroutines

Stratus manuals use the following format conventions for documenting subroutines. Note that the subroutine descriptions do not necessarily include each of the following sections.

`subroutine_name`
The name of the subroutine is at the top of the first page of the subroutine description.

Purpose
Explains briefly what the subroutine does.

Usage
Shows how to declare the variables passed as arguments to the subroutine, declare the subroutine entry in a program, and call the subroutine.

Arguments
Describes the subroutine arguments.

Explanation
Provides information about how to use the subroutine.

Error Codes
Explains some error codes that the subroutine can return.

Examples
Illustrates uses of the subroutine or provides sample input to and output from the subroutine.

Related Information
Refers you to other subroutines and commands similar to or useful with this subroutine.

Online Documentation

Stratus provides the following types of online documentation.

- The directory `>system>doc` provides supplemental online documentation. It contains the latest information available, including updates and corrections to Stratus manuals and a glossary of terms.

- Stratus offers some of its manuals online, via StrataDOC, an online-documentation product that consists of online manuals and StrataDOC Viewer, delivered on a CD-ROM (note that you must order StrataDOC separately). StrataDOC Viewer allows you to access online manuals from an IBM PC or compatible PC, a Sun® or Hewlett-Packard™ workstation, or an Apple® Macintosh® computer. StrataDOC provides such features as hypertext links and, on the workstations and PCs, text search and retrieval across the manual collection. The online and printed versions of a manual are identical.

If you have StrataDOC, you can view this manual online.

For a complete list of the manuals that are available online as well as more information about StrataDOC, contact your Stratus account representative.

For more information about StrataDOC as well as a complete list of the manuals that are available online, contact your Stratus account representative.

Ordering Manuals

You can order manuals in the following ways.

- If your system is connected to the Remote Service Network (RSN), issue the `maint_request` command at the system prompt. Complete the on-screen form with all of the information necessary to process your manual order.

- Customers in North America can call the Stratus Customer Assistance Center (CAC) at (800) 221-6588 or (800) 828-8513, 24 hours a day, 7 days a week. All other customers can contact their nearest Stratus sales office, CAC office, or distributor; see the file `cac_phones.doc` in the directory `>system>doc` for CAC phone numbers outside the U.S.

Manual orders will be forwarded to Order Administration.

Commenting on This Manual

You can comment on this manual by using the command `comment_on_manual` or by completing the customer survey that appears at the end of this manual. To use the `comment_on_manual` command, your system must be connected to the RSN. If your system is **not** connected to the RSN, you must use the customer survey to comment on this manual.

The `comment_on_manual` command is documented in the manual *VOS System Administration: Administering and Customizing a System (R281)* and the *VOS Commands Reference Manual (R098)*. There are two ways you can use this command to send your comments.

- If your comments are brief, type `comment_on_manual`, press Enter or Return, and complete the data-entry form that appears on your screen. When you have completed the form, press Enter.
- If your comments are lengthy, save them in a file before you issue the command. Type `comment_on_manual` followed by `-form`, then press Enter or Return. Enter this manual's part number, `R040`, then enter the name of your comments file in the `-comments_path` field. Press the key that performs the `CYCLE` function to change the value of `-use_form` to `no` and then press Enter.

  Note: If `comment_on_manual` does not accept the part number of this manual (which may occur if the manual is not yet registered in the `manual_info.table` file), you can use the `mail` request of the `maint_request` command to send your comments.

Your comments (along with your name) are sent to Stratus over the RSN.

Stratus welcomes any corrections and suggestions for improving this manual.

Contents

Tables

Figures

Chapter 1: An Overview of the VOS C Programming Language

This chapter provides an overview of the VOS C programing language, including information on the following:

- the C programming language
- the VOS C programming language
- C terminology
- program files and organization.

In addition, this chapter contains an example C program.

See the *VOS C User's Guide (R141)* for information on program development topics, such as compiling and binding.

The C Programming Language

The C programming language was developed at Bell Laboratories in 1972 by Dennis Ritchie. Similar in some ways to PL/I and Pascal, C is a modern language in that it has a clearly defined syntax and modular structure. Many early implementations of C were based on the description of the language found in the book *The C Programming Language* (Prentice-Hall, Inc., 1978) by Brian Kernighan and Dennis Ritchie.

In 1989, the American National Standards Institute (ANSI) approved a clearly delineated definition of the C language. That definition is documented in the *Programming Language C* (ANSI Std X.3.159-1989). This manual refers to the ANSI definition of the C language as the "ANSI C Standard." Many descriptions in this manual are derived from the ANSI C Standard.

Originally used for UNIX system programming, C has become an increasingly popular language for application programming for a number of reasons.

- Preprocessing. The C preprocessor allows you to create programs that are easier to read, easier to change, and easier to port to a different system.

- Portability. The C language, now that it is standardized, should facilitate the development of machine-independent programs.

- Extensive Data Types. The C language has a full set of data types, including pointers, structures, unions, and bit fields.

- Varied Operators. The C language has a large and varied assortment of operators.

- Pointer Operations. The C language allows extensive use of pointers and pointer arithmetic.
- Library Functions. The C library provides functions for a large variety of tasks, such input and output, string handling, memory allocation, and mathematics.

The VOS C Programming Language

The VOS implementation of the C language is, with some exceptions, compatible with the definition of C specified by the ANSI C Standard. In addition, VOS C has extensions that go beyond the definition given in the ANSI C Standard. Some of the major VOS C extensions include the following:

- use of the alignment specifiers, `$shortmap` and `$longmap`, in the declaration of an identifier
- varying-length character string (`char_varying`) data, which is useful in interfacing with the VOS service subroutines and the other VOS languages
- generic, string-manipulation functions, such as `strcpy`, which allow arguments that are pointers to `char_varying` strings, pointers to `char`, or a combination of the two pointer types
- UNIX I/O functions, such as `read` and `write`
- several built-in functions for handling National Language Support characters and for generating `char_varying` values.

VOS C extensions are **not transportable** to other operating systems. See Appendix B for a comprehensive summary of the VOS extensions to ANSI Standard C.

In addition to the VOS C extensions, the ANSI C Standard specifies that certain aspects of the C language are undefined or implementation-defined. For an undefined behavior, the Standard explicitly imposes no requirements. For an implementation-defined behavior, the Standard states that each implementation can determine what it will do. A program that depends on an undefined or implementation-defined behavior may or may not be transportable to other operating systems.

In this manual, the description of VOS C extensions as well as undefined and implementation-defined behaviors are accompanied by wording such as “in VOS C” or “as a VOS C extension.”

C Terminology

To understand the C language, you need to know a few new terms. The following three terms are central to many of the explanations in this manual.

A *function* is a subprogram invoked during the evaluation of a function-call expression that designates the function. A function takes zero or more arguments as input values, and returns a single value to the point of invocation. In addition to returning a value, functions often produce *side effects*, such as modifying data defined outside the function.

An *object* is a region of storage, the contents of which can represent values.

An *lvalue* is an expression that designates a region of storage (object). For example, the name of a variable is an lvalue designating a particular region of storage.

This manual defines new terms as they are needed for particular explanations. In addition, the Glossary contains definitions for many of the terms used in this manual.

Program Files and Organization

In developing a C program, you create or generate three files or modules: source, object, and program.

A C source program is a text file, called a *source module*. The file name of a VOS C source module must have the suffix `.c`. You write and update source modules with an editor such as the VOS Word Processing Editor or Emacs. A source module can have up to 32,767 lines of text. Each line can be up to 300 characters in length. See the *Introduction to VOS (R001)* manual for the naming conventions used for a VOS file name.

A VOS C program can consist of one or more source modules. Each source module contains zero or more function definitions. The VOS C compiler translates the code in a source module into object code. If the compilation is successful, the compiler generates an object module.

An *object module* contains the object code and has the name of the source module file and the `.obj` suffix. The object modules resulting from previously compiled source modules can be preserved either individually or in libraries of object modules.

The VOS binder combines a set of one or more object modules into a program module. If the binding is successful, the binder generates an executable program module. A *program module* contains executable code and has the suffix `.pm`. Binding compacts the code and resolves symbolic references to external programs and variables that are used by the object modules in a set and in the object library.

The object modules comprising a program module must contain at least one function definition with external linkage. Typically, though not necessarily, the object modules comprising a program module contain a function called `main` that has external linkage. See the “Program Startup and the Program Entry Point” section in Chapter 6 for information on the entry point of a program module.

For detailed information on compiling and binding a C program, see the *VOS C User’s Guide (R141)*.

Program Example

The program in Figure 1-1 is a short example that shows some of the parts of a typical C program.

Preprocessor Control Lines

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

#define ERROR 1
#define NO_ERROR 0
```

Function and Object Declarations

```
int get_name(void);
char name[30];
```

Function Definition for `main`

```
main()}
{
   int ret_value;}

   {ret_value = get_name();

   if (ret_value == ERROR)
      {
      puts(“You did not enter a name.”);
      return(ERROR);
      }
   else
      {
      printf(“The name that you entered is \%s\n”,
name);
      return(NO_ERROR);
      }
}
```

Other Function Definition

```
int get_name(void)
{
   char buffer[30];

   puts(“Enter a name to display.”);
   if ( strlen(gets(buffer)) > 0 )
      {
      strncpy(name, buffer, 30);
      return(NO_ERROR);
      }
   else
      return(ERROR);
}
```

Figure 1-1. `Sample C Program`

As shown in Figure 1-1 a C program typically contains the following parts.

Preprocessor Control Lines. These lines often appear at the beginning of a program. The program example uses the `#include` preprocessor directive to incorporate two header files, `stdio.h` and `string.h`, into the program module. A *header file* is a file that the compiler incorporates into the source module. A header file is sometimes called an include file. The program example uses the `#define` directive to declare two *macros* or defined constants, `ERROR` and `NO_ERROR`. See Chapter 9 for information on preprocessor directives.

Function and Object Declarations. In a C program, declarations for identifiers can appear either outside or inside a function. The location of a declaration can affect the characteristics of the data item that is declared. In C, an object or function is typically declared before it is referenced in the program. See Chapter 3 for information on object declarations. See Chapter 6 for information on function declarations.

Function Definition for `main`. The ANSI C Standard requires that the function called at program startup be named `main`. The `main` function, in turn, can call other functions. Although the example of `main` shown in Figure 1-1 does not declare or use parameters, the `main` function has two program parameters, `argc` and `argv`, that allow it to access command-line arguments. The `main` function can return a value to indicate whether the program has terminated normally or abnormally. See "The `main` Function" section in Chapter 6 for information on that function.

Other Function Definitions. The source modules comprising a program can contain definitions for functions other than `main`. The programmer determines the names, return values, and parameters of these other functions based on the tasks to be performed by the particular program. These other functions are invoked through the use of a function-call expression, such as `get_name()` in Figure 1-1, that appears either in `main` or in another function that is executing. See Chapter 6 for information on function return values and parameters. See the "Function-Call Operator" section in Chapter 7 for information on invoking a function.

Chapter 2:
Basic Elements of the VOS C Language

A source module in C contains elementary constructs such as identifiers, constants, keywords, and operators. These are constructed from the symbols of the source character set. This chapter describes the source and execution character sets and the elementary language constructs.

The VOS C Character Set

Programs written in C use two character sets:

- a source character set
- an execution character set.

The *source character set* consists of the characters that you use to write a C source module. For example, all alphabetic characters used in the keyword `register` are part of the source character set.

The *execution character set* consists of the characters, including nongraphic characters, that are available during the execution of a program. For example, you cannot enter a literal audible alert (beep) when creating a source module. But you can represent an audible alert using the `\a` escape sequence from the execution character set. When the program executes, the terminal beeps when it encounters this escape sequence.

The VOS C source character set conforms to the ANSI C Standard except that VOS C also allows the use of the dollar sign character (`$`). To permit National Language Support (NLS) characters, the VOS C source character set includes characters in the range 80 through FF hexadecimal. The VOS C execution character set includes all escape sequences prescribed by the ANSI C Standard.

Source Character Set

The source character set for a VOS C source module includes the 52 uppercase and lowercase alphabetic characters, the 10 decimal digits, and 30 graphic characters. Figure 2-1 shows the VOS C source character set.

Uppercase alphabetic characters:

```
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
```

Lowercase alphabetic characters:

```
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
```

Decimal digits:

```
0 1 2 3 4 5 6 7 8 9
```

Graphic characters:

```
! " # % & ' ( ) * $ + , - . / : ; < = > ? [ \ ] ^ _ { | } ~
```

Figure 2-1. VOS C Source Character Set

In addition, the source character set includes the space character, and control characters representing a horizontal tab, vertical tab, form feed, carriage return, and newline. These six characters (and comments) are called *white-space characters*. The compiler treats white-space characters as if they were space characters except when they occur in character constants or character-string literals. Within a source module, you can use these white-space characters as separators between adjacent tokens, or you can use them within character constants or character-string literals. Otherwise, the compiler ignores these characters.

The source character set can include NLS characters within character-string literals, character constants, and comments. NLS characters are unique to the VOS implementation of the C language. For more information on National Language Support and VOS C, see the *National Language Support User's Guide (R212).*

Execution Character Set

The execution character set consists of all characters found in the source character set as well as the following:

- certain escape sequences used to represent graphic and nongraphic characters
- the null character used to mark the end of a string.

Characters in the execution character set are encoded using ASCII codes.

Escape Sequences

An *escape sequence* can represent a graphic or nongraphic character within a character constant or character-string literal. An escape sequence is represented by a reverse slant (`\`) followed by one or more characters. Some escape sequences represent nongraphic characters that you use to display output on the terminal's screen.

The VOS C escape sequences are shown in Table 2-1. In the table, the term *active position* is that location on the terminal's screen where the next character would be output.

Table 2-1. Escape Sequences *(Page 1 of 2)*

Escape Sequence	**Character Represented**	**Effect When Displayed**
`\a`	audible alert	Causes the terminal to beep.
`\b`	backspace	Moves the active position to the previous position on the current line.
`\f`	form feed	Moves the active position to the beginning of a new page.
`\n`	newline	Moves the active position to the beginning of the next line.
`\r`	carriage return	Moves the active position to the beginning of the current line.
`\t`	horizontal tab	Moves the active position to the next horizontal tabulation location of the current line.
`\v`	vertical tab	Moves the active position to the initial position of the next vertical tabulation location.
`\"`	quotation marks	Generates the `"` character.
`\'`	apostrophe	Generates the `'` character.
`\\`	reverse slant	Generates the `\` character.

`\`*`ddd`*
`\`*`dd`*
`\`*`d`*
	–	Generates the character represented by the specified octal code. The code can be one to three digits and can consist of the octal digits `0` through `7`. For example: `\101`.

Table 2-1. Escape Sequences *(Page 2 of 2)*

Escape Sequence	Character Represented	Effect When Displayed
`\x`*dd* `\x`*d* `\X`*dd* `\X`*d*	–	Generates the character represented by the specified hexadecimal code. The code can be one or two digits and can consist of the hexadecimal digits `0` through `9`, the lowercase letters `a` through `f`, or the uppercase letters `A` through `F`. For example: `\x41`.

As shown in Table 2-1, one escape sequence format allows you to represent any valid character by using a reverse slant followed by the character's hexadecimal or octal ASCII code. Notice that you **cannot** use a decimal ASCII code to represent a character in an escape sequence. For example, `\65` cannot be used to represent an `A` character.

In addition to the escape sequences shown in Table 2-1, another escape sequence format allows you to represent most characters in the following form: character, where *`character`* represents a specified character from the execution character set. For example, the `?` escape sequence represents the question mark character (`?`).

> **Note:** You must use the `\\` escape sequence to represent the reverse slant character (`\`). In character constants, you use the `'` escape sequence to represent the apostrophe character (`'`). In character-string literals, you use the `"` escape sequence to represent the quotation mark character (`"`).

The following table shows some examples of escape sequences as well as the characters and ASCII codes that they represent. Notice that you can represent an uppercase `A` character by using an escape sequence with the ASCII code in either octal or hexadecimal format.

Escape Sequence	Character Represented	ASCII Value (in decimal)
`\101`	uppercase `A`	65
`\x41`	uppercase `A`	65
`\A`	uppercase `A`	65
`\\`	reverse slant	92
`\0`	null character	0
`\n`	newline	10
`\a`	audible alert	7

The Null Character

The execution character set includes the *null character*, a byte with all bits set to 0. The null character represents the ASCII code 0. In C, the null character marks the end of a string. In addition, the compiler appends a null character after the last character in a character-string literal.

Tokens

Tokens are the minimal lexical units of the C language. They are the building blocks from which you construct expressions, statements, functions, and preprocessor control lines. Tokens are categorized into six classes:

- keywords
- identifiers
- operators
- punctuators
- constants
- character-string literals.

The following sections describe each category of tokens. The last section in the chapter describes comments within source code.

Keywords

A *keyword* is a word that has special meaning to the compiler. For example, keywords identify data types, storage classes, and statements. You cannot use keywords as identifiers. The VOS C keywords are shown in Figure 2-2. An asterisk (*) precedes each keyword that is a VOS C extension.

```
*accept           do                 if                struct
 auto             double             int               switch
 break            else               long              typedef
 case             enum               register          union
 char            *ext_shared         return            unsigned
*char_varying     extern             short             void
 const            float              signed            volatile
 continue         for                sizeof            while
 default          goto               static
```

Figure 2-2. VOS C Keywords

The `accept` statement is used to display and manage forms in an application program. Refer to the *VOS C Forms Management System (R070)* for more information on these statements.

Identifiers

An *identifier* denotes one of the following elements of a C program: an object; a function; a tag or a member of a structure, union, or enumeration; a `typedef` name; a label name; or a macro name or parameter. An identifier, also called a *name*, is a sequence of 1 to 2048 alphabetic characters, digits, underline characters (_) and dollar sign characters (`$`). The first character of an identifier must be a letter, underline character, or dollar sign character. An identifier cannot be one of the keywords listed in Figure 2-2.

Identifiers can contain uppercase letters, lowercase letters, or both. By default, the compiler distinguishes uppercase from lowercase letters. For example, `abc` and `ABC` denote different identifiers. An identifier cannot contain a space character because space characters serve to separate adjacent tokens.

The dollar sign character is normally reserved for use with an extended set of VOS C keywords such as the `$longmap` and `$shortmap` specifiers. The `$` character is also used with the VOS C built-in function names such as `$unshift`. See Chapter 12 for more information on the use of the dollar sign character.

Notes:

1. If an identifier with **external linkage** exceeds 32 characters, the compiler truncates the identifier to 32 characters and issues an error message.

2. A macro name cannot exceed 256 characters. In a function-like macro, a macro parameter cannot exceed 300 characters.

3. The VOS Symbolic Debugger recognizes the identifiers of various entities defined in the source modules that comprise the program if you specify the `-table` or `-production_table` compiler argument. However, because macro names are replaced during the preprocessing phase of compilation, the debugger does not recognize macro names.

The following table shows examples of valid and invalid identifiers.

Valid Identifiers	**Invalid Identifiers**
`count`	`register`
`s$error`	`12calc`
`link_list_item_12`	`mail-list`
`MAX_COUNT`	`won't_work`

An identifier has scope and linkage. *Scope* is the region of a program's source text in which an identifier is visible. An identifier has one of the following scopes: file, block, function, or function prototype. *Linkage* is the process by which two lexically identical identifiers in different or the same scopes can be made to refer to the same function or object. An identifier has one of the following types of linkage: external, internal, or no linkage. See Chapter 3 for information on scope and linkage.

It is recommended that you adhere to the following optional conventions for creating identifiers.

- In `#define` preprocessor control lines, use uppercase letters for macro names, especially those that define numeric constants.

- Use lowercase letters for all other identifiers and keywords.

- Avoid using identifiers that begin with an underline character (_) unless you are coding an application that uses C subsystems.

- Do not use identifiers that begin with an underline character (_) followed by either an uppercase letter or another underline. These names are reserved for VOS C.

Operators

The C language has a varied assortment of operators. An *operator* performs an operation (evaluation) on one or more operands. An *operand* is a subexpression on which an operator acts. The result of the operation specifies the computation of a value, designates an object or function, generates side effects, or produces a combination of these actions.

The symbol and name for each C operator are shown in Figure 2-2. See Chapter 7 for detailed information on using operators, including the rules of operator precedence and associativity.

Table 2-2. Symbols and Names for the VOS C Operators *(Page 1 of 2)*

Postfix Operators	
`[]`	Array subscript
`()`	Function call
`.`	Structure/union member
`->`	Structure/union pointer
`++`	Postincrement
`--`	Postdecrement
Unary Operators	
`&`	Address of
`*`	Indirection
`sizeof`	Size of
`()`	Cast
`-`	Unary minus
`!`	Logical negation
`~`	Bitwise complement
`++`	Preincrement
`--`	Predecrement
Conditional Expression	
Binary Operators	
`*`	Multiplication
`/`	Division
`%`	Remainder
`+`	Addition
`-`	Subtraction

Table 2-2. Symbols and Names for the VOS C Operators *(Page 2 of 2)*

`<<`	Left shift
`>>`	Right shift
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to
`==`	Equality
`!=`	Inequality
`&`	Bitwise AND
`^`	Bitwise exclusive OR
`|`	Bitwise inclusive OR
`&&`	Logical AND
`||`	Logical OR
`? :`	Conditional
Assignment Operators	
`=`	Assignment
`*=`	Multiplication assignment
`/=`	Division assignment
`%=`	Remainder assignment
`+=`	Addition assignment
`-=`	Subtraction assignment
`<<=`	Left-shift assignment
`>>=`	Right-shift assignment
`&=`	Bitwise-AND assignment
`^=`	Bitwise exclusive-OR assignment
`|=`	Bitwise inclusive-OR assignment
Comma Operator	
`,`	Comma

In macro-defining preprocessor control lines, two additional operators are available: the stringize operator (`#`) and the concatenation operator (`##`). See Chapter 9 for information on these two operators.

In VOS C `char_varying` string expressions, you can use the `+` operator to concatenate two or more strings. See the "Concatenation with the `char_varying` Type" section in Chapter 4 for more information on this use of the `+` operator.

Punctuators

A *punctuator* is a token that has special syntactic and semantic significance to the compiler but does not specify an operation that yields a value. Typically, a punctuator indicates how an entity will be used, or it delimits an identifier or block of code. Depending on the context, the same symbol that is used to represent a punctuator can also be used as an operator or part of an operator. The C punctuators and their uses are shown in Table 2-3.

Table 2-3. Punctuators

Punctuator	Use
`[]`	Indicates that a variable is an array; delimits an array's size.
`()`	Indicates that an identifier is a function. Also, specifies a function's parameter or argument list.
`{}`	Marks the beginning and end of a compound statement, which is either a block of code or a function body.
`*`	Indicates that a variable is being declared as a pointer.
`,`	Separates parameters in a function declaration or in a function-like macro, or arguments in a function call. Also, separates items in a list of declarations.
`:`	Marks an identifier as a label. Also, marks an integer constant specifying the size of a bit field in a structure or union declaration.
`=`	Separates a declarator from an initializer in a declaration.
`;`	Marks the end of a statement.
`...`	Specifies (in a function prototype) that a function takes a varying number of arguments or arguments of varying types.
`#`	Indicates a preprocessor directive.

The `[]`, `()`, and `{ }` punctuators occur only in pairs, possibly enclosing expressions, declarations, or statements.

Constants

A *constant* is a sequence of characters that represents a fixed numerical value. Every constant has an associated data type. Constants are grouped into the following categories:

- integer constants
- floating-point constants
- character constants
- enumeration constants.

Integer Constants

An *integer constant* is a constant that holds a whole number (that is, a number with no decimal point or exponent). The value of an integer constant is non-negative. A leading minus sign is interpreted as a unary minus operator. You can specify the base of an integer constant as decimal, octal, or hexadecimal.

- A *decimal* integer constant consists of one or more digits, the first of which is not `0`.

- An *octal* integer constant consists of the optional prefix `0` followed by one or more of the octal digits `0` through `7`.

- A *hexadecimal* integer constant consists of the prefix `0x` or `0X` followed by the hexadecimal digits `0` through `9`, the lowercase letters `a` through `f`, or the uppercase letters `A` through `F`.

In VOS C, an integer constant can be a whole number in the range 0 through 4,294,967,295. If an integer constant's value can be represented in a `signed int`, the constant has the type `signed int`. Otherwise, if the integer constant's value cannot be represented in a `signed int`, the constant has the type `unsigned int`. The compiler issues an error message if an integer constant falls outside of its allowed range.

The following table shows some examples of integer constants as well as the base and corresponding decimal value of each constant.

Base	Integer Constant	Decimal Value
octal	`060`	48
octal	`012`	10
hexadecimal	`0x12`	18
decimal	`12`	12
hexadecimal	`0xFFFFFFFF`	4,294,967,295

Floating-Point Constants

A *floating-point constant* is an approximation of a real number (that is, a number expressed as a decimal fraction, such as 7.3567). You can specify a floating-point constant by using either a decimal fraction or exponential notation. A floating-point constant has the following format:

$$\mathit{value\_part}\left[\begin{Bmatrix}\texttt{E}\\\texttt{e}\end{Bmatrix}\begin{bmatrix}\texttt{+}\\\texttt{-}\end{bmatrix}\mathit{exponent}\right]$$

The components of the `value_part` may include an integral part, followed by a decimal point, followed by a fractional part. In exponential notation, *`value_part`* is called the *mantissa*. Both the integral and fractional parts consist of one or more digits. In *`value_part`*, either the integral part or the decimal point and fractional part must be present. Also, either the decimal point or the *`exponent`* must be present.

The letter `e` or `E` separates *`value_part`* from the optional *`exponent`*. If the floating-point constant uses exponential notation, you specify an exponent. You can precede the exponent by a plus or minus sign to indicate its sign. The exponent represents the power of 10 by which *`value_part`* is multiplied.

For example, in the floating-point constant `1.23e10`, the *`value_part`* is 1.23, and the *`exponent`* is 10. This floating-point constant represents 1.23 multiplied by 10^{10} or 12,300,000,000. As another example, the floating-point constant `1.23e-10` represents 1.23 multiplied by 10^{-10}, or 0.000000000123.

A floating-point constant is of the type `double`. A floating-point constant can have about 15 significant decimal digits in its mantissa and can be a positive or negative number in the approximate range 10^{-308} through 10^{307}.

The following table shows some examples of floating-point constants and the approximate decimal value that each constant represents.

Floating-Point Constant	**Decimal Value Represented**
`3.3`	3.3
`5.0e5`	500,000
`5E-10`	0.0000000005
`.160`	0.160
`0.160e4`	1600

Character Constants

A *character constant* is one or two characters or escape sequences enclosed in apostrophes. For example, `'x'` and `'\0'` are character constants. The value of a character constant is its numeric rank in the VOS internal character coding system. For ASCII characters, this value is the same as the character's ASCII code.

A character constant is **not** a character-string literal. A character-string literal is enclosed in quotation marks to indicate an array of characters ending with a null character. See the "Character-String Literals" section later in this chapter for more information on character-string literals.

To use an apostrophe, reverse slant, or newline character as a character constant, enter the corresponding escape sequence. See the "Escape Sequences" section earlier in this chapter for more information on entering characters using escape sequences.

By default, the compiler considers a character constant to be an `unsigned char`. An `unsigned char` is an 8-bit unsigned value stored in one byte. If you change the default by specifying the `signed` value for the `default_char` option in a `#pragma` preprocessor control line or in the corresponding command-line argument, the compiler considers a character constant to be a `signed char`. In either case, a character constant is promoted to `int` before it participates in an expression. Therefore, you can use a character constant in any expression where you can use an `int`. See the "Usual Arithmetic Conversions" section in Chapter 5 for information on promotions that involve the `char` type.

> **Note:** For certain characters, signed and unsigned character constants specifying the same value yield different results. The character constant `'\xFF'` is equal to -1 if it is signed, but is equal to 255 if it is unsigned.

The following table shows some examples of character constants and the corresponding decimal values and characters.

Character Constant	**Decimal Value**	**Character Represented**
`'\0'`	0	null character
`'0'`	48	zero
`'a'`	97	lowercase `a`
`'\a'`	7	audible alert
`'\''`	39	apostrophe
`'\\'`	92	reverse slant
`'\n'`	10	newline

In VOS C, double-byte character constants are of the type `short int`. For example, the double-byte character constant `'00'` is stored in two bytes. Each byte stores the hexadecimal ASCII code (30 hexadecimal) for the `0` character.

Enumeration Constants

An *enumeration constant* is one of a set of identifiers declared as members of a user-defined `enum` type. Variables of that `enum` type can be assigned any one of the specified set of identifiers. In effect, each identifier is a named integer constant value. Figure 2-3 shows an example of an enum containing six enumeration constants.

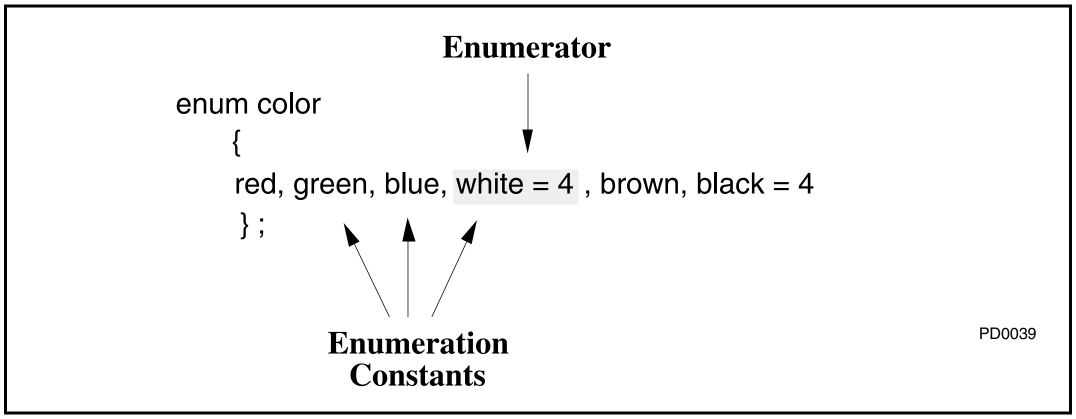


Figure 2-3. Sample Enumeration Constants

Enumeration constants are of the type `int` and can be in the range -2,147,483,648 through 2,147,483,647. You can use an enumeration constant wherever an integer constant is valid unless you specify `check_enumeration` in a `#pragma` preprocessor control line or the corresponding compiler argument. Some programs use enumeration constants as an alternative to constants specified with `#define` directives.

The following rules describe how the compiler determines the value of an enumeration constant.

- If the first enumerator in the list does not contain an equals sign followed by an integral constant expression, the enumeration constant has the value 0. In the example shown in Figure 2-3, the enumeration constant `red` equals the value 0.
- If any enumerator contains an equals sign followed by an integral constant expression, the enumeration constant has the specified value. In the example, the enumeration constants `white` and `black` equal the value 4.
- Each subsequent enumerator with no specified value has the value of the preceding enumeration constant plus 1. In the example, `green` equals the value 1, `blue` equals the value 2, and `brown` equals the value 5.

See the "Enumerated Types" section in Chapter 4 for more information on enumeration constants and examples of how to use enumerations.

Character-String Literals

A *character-string literal*, also called a string literal, is a sequence of zero or more characters or escape sequences enclosed in quotation marks. For example, `"bcd"` and `"AMOUNT = %3.2f\n"` are character-string literals. A character-string literal can have one or more null characters (`\0`) embedded in it. The quotation marks delimit the character-string literal but are not part of it.

To use a quotation marks, reverse slant, or newline character within a character-string literal, enter the corresponding escape sequence. See the "Escape Sequences" section earlier in this chapter for more information on entering characters using escape sequences.

Identical character-string literals may be represented by a single copy of the string in memory. A character-string literal has the data type array of `char`, which is identical to pointer to `char`. It has static storage duration. The characters delimited by quotation marks initialize the array. The compiler appends a null character after the last character.

Except when a character-string literal is the operand to the `sizeof` operator or when it initializes a character array, the compiler treats a character-string literal as a **pointer constant** to the address where the first character of the array is stored (see Figure 2-4). Therefore, a character-string literal can appear anywhere a pointer constant can appear.

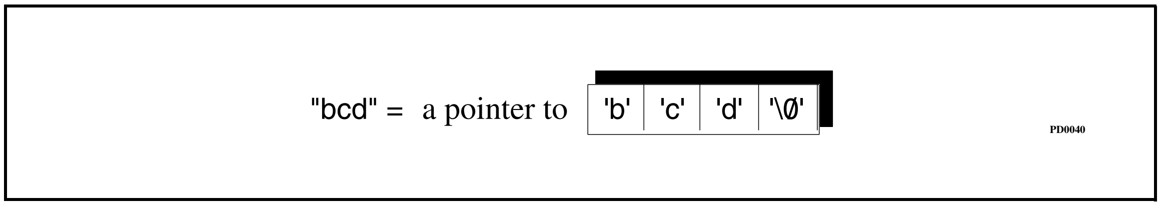


Figure 2-4. Character-String Literal

The size of a character-string literal is equal to the number of characters enclosed within the quotation marks plus 1 for the appended null character. For example, `sizeof("bcd")` yields the value 4.

A program **cannot** write to a character-string literal without specifying the `no_common_constants` option in a `#pragma` preprocessor control line. This option affects the storage of character-string literals in the following way:

- If you specify the `common_constants` option, character-string literals are stored in the program's code region. Therefore, trying to overwrite a character-string literal is an attempt to modify a protected page, which causes an error condition to occur. By default, character-string literals are stored in the program's code region.
- If you specify the `no_common_constants` option, character-string literals are stored in the program's static region. Therefore, the program can overwrite a character-string literal. However, overwriting a character-string literal in this way is not considered a good programming practice.

Use the reverse slant character (\) to continue a line for a character-string literal that is too large to represent on a single line. A reverse slant at the end of a line has the effect of concatenating the last character on that line with the first character on the next line. The following example shows this use of the reverse slant character.

```
"Some character-string literals are just too long to fit comfortably \
on a single line."
```

The compiler concatenates adjacent character-string literals separated only by white space. The first two character-string literals in the following example are concatenated.

```
printf("Character"  "-string", "literal");
```

The preceding `printf` format is equivalent to the following:

```
printf("Character-string", "literal");
```

The token representing a character-string literal can contain up to 2048 bytes. All characters count toward this total, including the enclosing quotation marks, any newline characters, and the terminating null character. A reverse slant character, indicating that the string continues on the next line, does not count toward the total. For example, the following character-string literal token contains 10 bytes.

```
"abc\n\
xyz"
```

The resulting character-string literal (excluding the quotation marks) consists of the following eight characters.

```
abc nxyz 0
```

Comments

Comments are embedded in the source code to help a reader understand the code. The characters `/*` begin a comment, and the characters `*/` end the comment. Anything between these delimiters is part of the comment. For example:

```
/*  This is an
    example comment  */
```

The compiler ignores comments, treating them as a single white-space character. A comment can appear anywhere a token can appear, including within preprocessor control lines. Comments cannot be nested.

Chapter 3: Declarations

This chapter explains how you declare identifiers in VOS C.

The "Declaring Identifiers: An Overview" section at the beginning of the chapter provides an overview of how to declare identifiers in C. The overview also explains general syntax rules and characteristics such as scope, linkage, and storage duration.

This chapter includes information on the following:

- storage classes and storage-class specifiers
- type specifiers
- type qualifiers
- alignment specifiers
- declarators
- initializers
- type definitions.

In addition, the "Other Considerations" section at the end of the chapter provides information on identifier name space and incomplete types.

Function declarations and definitions as well as varying-length character string declarations are unique and require a somewhat different set of rules. See Chapter 6 for more information on function declarations. See the "Varying-Length Character String Type" section in Chapter 4 for information on `char_varying` string declarations.

Enumeration, structure, and union declarations can use tags to identify a particular `enum`, `struct`, or `union` type. See Chapter 4 for information on declarations for enumerated, structure, and union types.

Declaring Identifiers: An Overview

This section explains the distinction between a declaration and a definition. It also explains how a declaration's location within a source module is significant, and describes the syntax of a declaration. Lastly, this section explains the following characteristics of an identifier:

- scope
- linkage
- storage duration.

Declarations and Definitions

A C *declaration* specifies the attributes of one or more identifiers: storage class, data type, type qualifiers, alignment, scope, linkage, and storage duration. A declaration merely announces the attributes of a data item defined elsewhere, perhaps in another source module that will be bound with the source module containing the declaration.

A *definition* is a declaration that provides a complete description of the data item, and also causes storage to be reserved for an object or function. For example, the following declaration is a definition because the `extern` keyword is not used. This definition causes storage to be allocated for `c_array` when the program executes.

```
char c_array[80];
```

In contrast, the following declaration is not a definition because the `extern` keyword tells the compiler that `c_array` is defined elsewhere. This declaration causes no storage to be allocated.

```
extern char c_array[];
```

Unless the distinction between declaration and definition is significant to the discussion, this manual uses the term *declaration* to refer to both declarations and definitions.

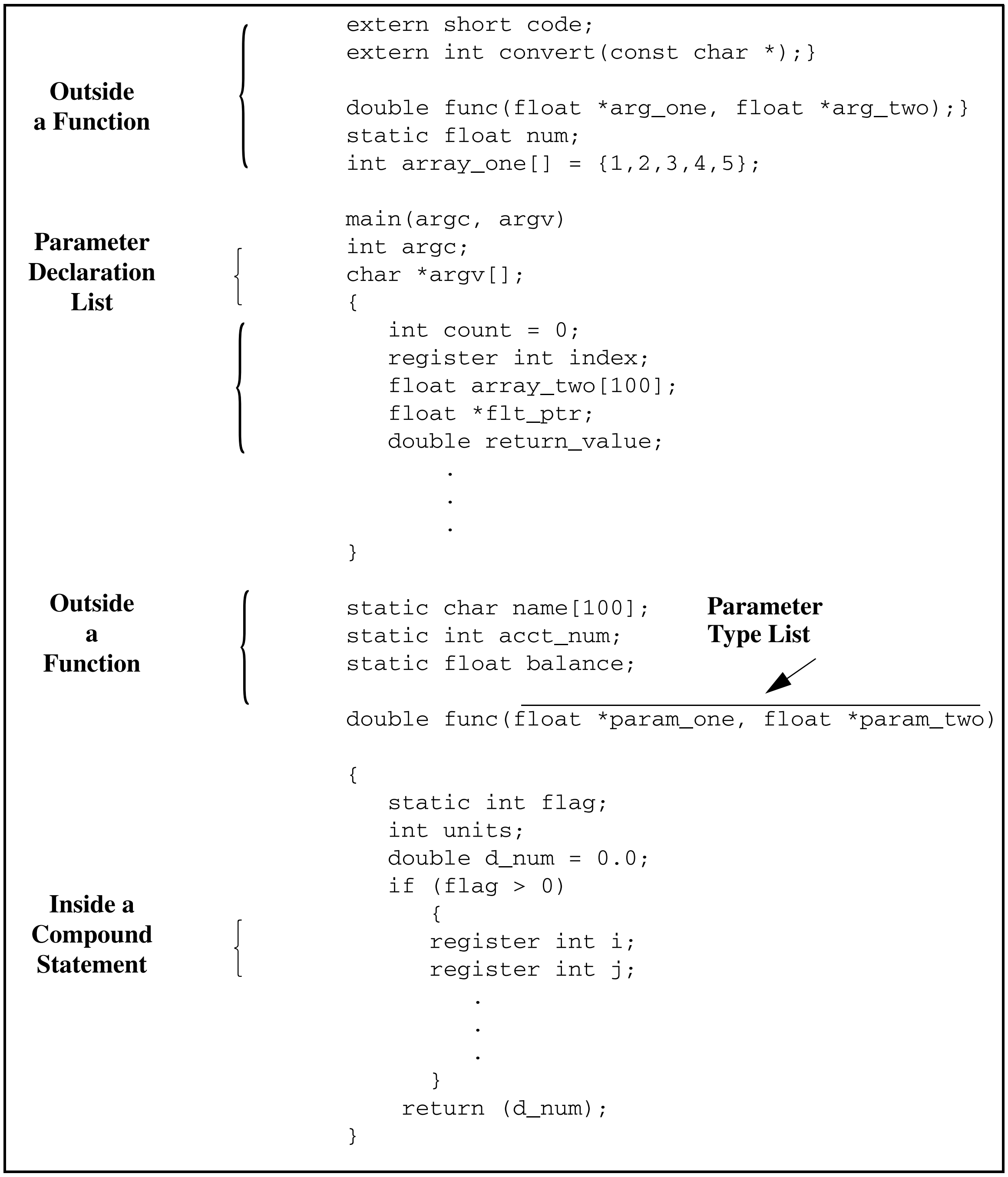
Program Structure and Declarations

To understand how program structure affects declarations, you need to know a few terms that describe the location of a declaration.

- A *source module* is a single text file that can contain language statements, preprocessor control lines, and comments. This file can be compiled to produce an object module. A source module is often called a source file.
- A *program* is a set of one or more source modules that, when compiled and bound together, comprise an executable program module.
- A *block* is a compound statement, a set of statements grouped into one syntactic unit and delimited by braces (`{}`). For example, a function definition's body is a block. Also, some statements, such as the `if` and `while` statements, can contain one or more blocks if they consist of a compound statement.
- When an identifier is declared *outside a function*, it is declared outside any function's formal parameter list and outside any block.
- When an identifier is declared *inside a function*, it is declared as part of a function's formal parameter list or inside a block.

Declarations can be placed in several locations within the structure of a C program. Where you place a declaration affects the characteristics of the entity that is declared. Scope, linkage, and storage duration are, to some extent, determined by where you declare a variable. For example, a variable declared **outside** a function without using a storage-class specifier is a global variable, available to all source modules that comprise the program. On the other hand, a variable declared **inside** a function without using a storage-class specifier is a local variable, available only to that function.

Figure 3-1 shows where declarations and definitions might appear in a typical C source module.



```
extern short code;
extern int convert(const char *);}

double func(float *arg_one, float *arg_two);}
static float num;
int array_one[] = {1,2,3,4,5};

main(argc, argv)
int argc;
char *argv[];
{
   int count = 0;
   register int index;
   float array_two[100];
   float *flt_ptr;
   double return_value;
       .
       .
       .
}

static char name[100];
static int acct_num;
static float balance;

double func(float *param_one, float *param_two)

{
   static int flag;
   int units;
   double d_num = 0.0;
   if (flag > 0)
      {
      register int i;
      register int j;
         .
         .
         .
      }
    return (d_num);
}
```

Figure 3-1. **`Program Structure and Declarations`**

As shown in Figure 3-1, declarations can appear in the following locations within a program:

- outside a function
- inside a function
  - in a function’s parameter type list (function-prototype form)
  - in a function’s parameter declaration list (nonprototype form)
  - in a function’s declaration list
  - inside a compound statement, such as a compound statement appearing in an `if` or `while` statement.

If declarations appear inside a function’s body or a compound statement, they must come immediately after the opening brace (`{`) and before any executable statement.

Syntax of a Declaration

In C, declarations can be simple or quite complex. Figure 3-2 shows the syntax that you use when you declare identifiers in VOS C. The overview shown in the figure is not a comprehensive explanation of declaration syntax. It only provides you with a general view of how to declare an identifier. The sections in this chapter comprehensively explain the rules for declarations.

Syntax of a Declaration:

```
[storage_class_specifier][type_qualifier][alignment_specifier]

    [type_specifiers][declarator][= initializer];
```

Sample Declaration:

```
auto volatile $longmap unsigned int count = 0;
```

The *`storage_class_specifier`* can be one of the following:

```
auto          ext_shared          static
extern        register            typedef
```

The *`type_qualifier`* can be one or both of the following:

```
const
volatile
```

The *`alignment_specifier`* can be one of the following:

```
$shortmap
$longmap
```

The *`type_specifiers`* must form a valid data type. You construct the type from one or more of the following specifiers:

```
char              float           short           union tag
char_varying(n)   int             signed          unsigned
double            long            struct tag       void
enum tag
```

The *`declarator`* consists of one identifier for the object or function that you are declaring and zero or more of the following modifiers:

```
[]     ()     *
```

The *`initializer`* sets the initial value of a variable. Depending on the data type, the *`initializer`* takes one of several forms. See the “Initializers” section later in this chapter for information on initializers.

Figure 3-2. Syntax of a Declaration

In Figure 3-2, the `typedef` specifier is categorized as a storage-class specifier for convenience only. See the "Type Definitions" section later in this chapter for information on the `typedef` specifier. The following paragraphs explain additional points about C declarations.

As shown by the syntax (Figure 3-2), no one element of a declaration is required in all contexts. However, every declaration must have a combination of one or more elements sufficient to form a valid declaration. The position of each element within a declaration can vary, but the declarator and initializer, if present, must be in the last position, immediately before the semicolon.

You can specify declarators and declarator-initializer combinations within a comma-separated list. For example, the following declaration declares two variables. It specifies that `letter` is a variable of the type `char`, and `char_ptr` is a pointer to an object of the type `char`.

```
char   letter, *char_ptr;
```

As another example, the following declaration initializes three `int` variables: `code`, `status`, and `flag`. The initializers set the initial value of each variable to a different value.

```
static int   code = 5, status = 100, flag = 1;
```

When you use an array (`[]`), function (`()`), or pointer (`*`) modifier within a declarator, the rules of operator precedence and associativity apply to these modifiers. These rules determine the type of the object or function that you declare. As with arithmetic expressions, you can use parentheses to override the order of precedence in a complex declarator. The parentheses cause the compiler to change the normal order of evaluation.

See the "Complex Declarators" section later in this chapter for more information on complex declarators.

Scope

Scope defines the region of the program's source text in which an identifier is visible (that is, can be used). An identifier can have file, block, function, or function prototype scope.

File Scope

If the declarator or type specifier that declares an identifier appears outside any function's parameter list or block, the identifier has *file scope*. File scope extends from the point where the identifier is complete to the end of the source module, and throughout all source modules included by preprocessor directives. Identifiers with file scope can be referenced from parts of the source module that come **after**, but not before, the declaration.

In the following example, `id_with_file_scope` is declared outside any function.

```
void calc(void);

main()
{
   calc();
   id_with_file_scope = (3.3 * 4.4);  /*  Identifier is not visible:
*/
                                     /*  error occurs              */
      .
      .
      .
}

float id_with_file_scope;             /*  Declaration  */

void calc(void)
{
   id_with_file_scope = (1.1 * 2.2);  /*  Identifier is visible:  no
error  */
      .
      .
      .
}
```

Notice that, because `id_with_file_scope` is declared after the `main` function, you can reference this variable from within the `calc` function but not from within the `main` function.

Block Scope

If the declarator or type specifier that declares an identifier appears inside a block or inside a function's parameter list, the identifier has *block scope*. Block scope extends from the point where the identifier is complete to the closing brace that terminates the associated block.

Identifiers can go in and out of scope. When an identifier is in scope, it is visible. When an identifier is out of scope, it is hidden. As an example, assume that two lexically identical identifiers have been declared, one in an outer block and one in an inner block. In the inner block, the outer-block name is hidden until the inner block containing the redefining declaration ends.

As another example, consider the following program in which the variable `identifier` is declared at three different points within the source module.

```
void func(void);
int identifier = 1;          /*  Global variable with file scope  */

main()
{
   int identifier = 2;        /*  Local variable with block scope  */

   printf("In main, identifier = %d\n", identifier);
   func();
}

void func(void)
{
   printf("In func, identifier = %d\n", identifier);
   while (1)
      {
      int  identifier = 3;    /*  Variable with block scope that is
known  */
                         /*  only within this compound statement     */

      printf("In the while compound statement, identifier = %d\n",
identifier);
      return;
      }
}
```

As the program runs, separate storage is created for three distinct variables. The object associated with `identifier` changes as the three variables move in and out of scope. The output from the preceding program is as follows:

```
In main, identifier = 2
In func, identifier = 1
In the while compound statement, identifier = 3
```

Function Scope

Function scope extends from the opening brace to the closing brace of a function body. Only labels have function scope. The presence of a colon (`:`) following an identifier contextually indicates that it is a label. A label can be used (with a `goto` statement) anywhere in a function, even before the label is defined. Labels have a name space separate from all other identifiers visible within the function.

Function Prototype Scope

If the declarator or type specifier that declares an identifier appears within a list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*. Function prototype scope is very limited. The identifier's scope ends when the function declarator ends. In the following example, the identifier `string` has function prototype scope.

```
long atol(const char *string);
```

In a function declaration that includes a prototype, no storage is allocated for the identifiers in the parameter type list. In the prototype form of a function declaration, identifiers are optional, serving as comments that enhance the code's readability.

A function prototype in a function declaration is very different from a prototype in a function definition. When a prototype appears in a function definition, you specify the code that defines the function, and the identifiers associated with formal parameters are objects allocated by the caller and have block scope.

Linkage

Two lexically identical identifiers declared in different scopes or in the same scope more than once can refer to the same object or function depending on a characteristic called *linkage*. Thus, references to objects or functions can be matched across blocks of code and among source modules, libraries, and object modules.

In VOS C, there are three kinds of linkage:

- external linkage
- internal linkage
- no linkage.

External Linkage

In a set of source modules and libraries that constitute a program, every instance of a particular identifier with *external linkage* denotes the same object or function. An identifier with external linkage is visible across multiple source modules.

If a definition for an identifier that you declare with the `extern` storage-class specifier appears in the source module and the identifier has either file scope or external linkage, the identifier refers to this definition and has the same linkage and, if an object, the same shared characteristics. If no such definition for the identifier appears in the source module, the identifier has external linkage.

Identifiers for objects and functions can be declared without indicating a storage-class specifier in the declaration.

- If an object identifier is declared outside any function's parameter list or block without using any storage-class specifier, its linkage is external.

- If a function identifier is declared without using any storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier `extern`.

Figure 3-3 shows two separate source modules. In each module, the `ext_var` variable has external linkage. When the two source modules are compiled and bound, and the program runs, storage is created for one object named `ext_var`. The `main` and `func` functions reference the same variable.

Source Module #1:

```
int ext_var = 10;
extern void func(void);

main()
{
   printf("ext_var = %d\n", ext_var);         /*  ext_var = 10  */
   func();
   printf("ext_var = %d\n", ext_var);         /*  ext_var = 5   */
}
```

Source Module #2:

```
extern int ext_var;

void func(void)
{
   printf("ext_var = %d\n", ext_var);         /*  ext_var = 10  */
   ext_var = 5;
   printf("ext_var = %d\n", ext_var);         /*  ext_var = 5   */
}
```

Figure 3-3. External Linkage Example

In Source Module 2 (Figure 3-3), `ext_var` is declared `extern` and then assigned the value 5 in the `func` function. Because both source modules declare `ext_var` to have external linkage, the object that is written to in `func` is the same object that was defined in Source Module 1. After `func` is called and program control returns to `main`, the value stored in `ext_var` has been modified.

In VOS C, an object (but not a function) can have multiple definitions **if** the definitions are identical in all respects, including initializers. In addition, all `extern` declarations of an object or function must be consistent with the object or function's definition.

See "The `extern` Storage Class" section later in this chapter for information on using the `extern` storage-class specifier to declare a data item with external linkage.

Internal Linkage

Within one source module, each instance of an identifier with *internal linkage* denotes the same object or function. In contrast to an identifier with external linkage, an identifier with internal linkage is visible within only one source module. Identifiers that you declare outside any function's parameter list or block using the `static` storage-class specifier have internal linkage.

Figure 3-4 shows two separate source modules. In each module, the `internal_var` variable has internal linkage. When the two source modules are compiled and bound, and the program runs, storage is created for two distinct entities named `internal_var`. The `main` and `func` functions reference two different variables.

Source Module #1:

```
static int internal_var = 10;
extern void func(void);

main()
{
 printf("internal_var = %d\n", internal_var);   /*  internal_var = 10  */
 func();
 printf("internal_var = %d\n", internal_var);   /*  internal_var = 10  */
  }
```

Source Module #2:

```
static int internal_var;        /*  static storage class variables are  */
                                /*  initialized to 0                    */
void func(void)
{
 printf("internal_var = %d\n", internal_var);    /*  internal_var = 0  */
 internal_var = 5;
 printf("internal_var = %d\n", internal_var);    /*  internal_var = 5  */
}
```

Figure 3-4. Internal Linkage Example

In Source Module 2 (Figure 3-4), `internal_var` is declared with the `static` keyword and then assigned the value 5 in the `func` function. Because Source Module 2 declares `internal_var` to have internal linkage, the object that is written to in `func` is not the same object that was defined in Source Module 1. After `func` is called and program control returns to `main`, the value stored in `internal_var` (defined in Source Module 1) has not been modified.

You use the `static` storage-class specifier to indicate internal linkage when declaring a data object outside a function's parameter list or block. All data objects declared outside a function's parameter list or block, with or without the `static` keyword, have static storage duration.

Identifiers with internal linkage are particularly useful when you want to use an identifier within one source module but hide the identifier from other source modules. This type of situation is called *data hiding*. In Figure 3-4, the `internal_var` object defined in Source Module 1 is hidden from Source Module 2.

See "The `static` Storage Class" section later in this chapter for information on using the `static` storage-class specifier to define and declare a data item with internal linkage.

No Linkage

Identifiers with no linkage denote unique entities. An identifier with no linkage is visible within only one function or block. No permanent storage is created for the entity. The following identifiers have no linkage:

- an identifier declared to be anything other than an object or function
- an identifier declared to be a function parameter
- an identifier declared to be an object inside a block if the declaration does not contain the `extern` or `ext_shared` storage-class specifier.

For example, variables declared as `auto` have no linkage. Identifiers that do not declare objects or functions include structure, union, or enumeration tags, type definitions, enumeration constants, and labels. Each of these identifiers has no linkage.

Storage Duration

Storage duration is the amount of time an object exists. Only objects have storage duration. There are two storage durations: static and automatic.

Static Storage Duration

An object with *static storage duration* exists throughout the execution of the entire program. Static data is permanent, retaining its value even when the program is executing outside the object's scope. An object has static storage duration if one of the following conditions is true.

- The object is declared with external or internal linkage.
- The object is declared with no linkage but with an explicit `static` storage-class specifier.

An object with static storage duration is allocated and initialized once, when the program module is loaded for execution. If an object with static storage duration has no explicit initialization, the compiler sets the initial value to binary 0. If a `char_varying` object with static storage duration has no explicit initialization, the compiler sets all bytes in the current-length field and in the string itself to binary 0.

Automatic Storage Duration

An object with *automatic storage duration* exists only for the duration of a block of code. Automatic data is temporary. An object has automatic storage duration if the object is declared with no linkage and no explicit `static` storage-class specifier.

An object with automatic storage duration is automatically allocated storage and an initial value is determined when program control enters the block (compound statement or function) in which the object is defined. Likewise, storage is freed when execution of the block ends in any way, such as when a function ends with a `return` statement or with a call to the `longjmp` function. Entering an enclosed block suspends but does not end execution of the enclosing block. Similarly, calling a function suspends but does not end execution of the function containing the call.

The value of a pointer is unpredictable and should not be used if the pointer points to an automatic object that is no longer guaranteed to be reserved.

For an object with automatic storage duration, an initial value, if specified, is set upon each **normal entry** into the block: when the function is invoked or when control enters a compound statement. If control enters a compound statement by a transfer to a labeled statement, an initial value, if specified, is not set for automatic data within that block.

An object with automatic storage duration and no explicit initialization has an **unpredictable** initial value. If a `char_varying` object with automatic storage duration has no explicit initialization, both the current-length field and the string itself have unpredictable values.

Storage Class and Storage-Class Specifiers

Storage-class specifiers are keywords included in declarations. Storage class is determined by where the declaration of the object or function appears as well as by what storage-class specifiers are used in the declaration. Storage class directly influences the linkage and storage duration of an identifier.

You can use only one storage-class specifier in a declaration. A storage-class specifier, if one is used, is typically the first specifier in a declaration.

Based on the location of the declaration, the following three tables provide an overview of the VOS C storage-class specifiers allowed for objects.

- Table 3-1 lists the storage-class specifiers that are available for object identifiers declared outside a function.
- Table 3-2 lists the storage-class specifiers that are available for object identifiers declared as formal parameters in a function definition.
- Table 3-3 lists the storage-class specifiers that are available for object identifiers declared inside the body of a function definition.

For detailed information on each storage class, see the appropriate section later in this chapter.

Table 3-1. Storage-Class Specifiers Used Outside a Function

Storage-Class Specifier	Linkage	Duration
none	external	static
`extern`	external †	static
`ext_shared`	external	static
`static`	internal	static

† If a definition of the object appears in the source module and has either file scope or external linkage, the identifier refers to this object and has the same linkage and the same shared characteristics. If no such definition exists, the identifier has external linkage.

Table 3-2. Storage-Class Specifiers Used for Formal Parameters

Storage-Class Specifier	Linkage	Duration
none †	no linkage	automatic
`register`	no linkage	automatic

† The object has the parameter storage class.

Table 3-3. Storage-Class Specifiers Used Inside a Function or Block

Storage-Class Specifier	Linkage	Duration
none	no linkage	automatic
`auto`	no linkage	automatic
`extern`	external †	static
`ext_shared`	external	static
`register`	no linkage	automatic
`static`	no linkage	static

† If a definition of the object appears in the source module and has either file scope or external linkage, the identifier refers to this object and has the same linkage and the same shared characteristics. If no such definition exists, the identifier has external linkage.

With the `extern` storage class, you use the `extern` specifier to **declare** objects and functions that have been defined outside a function elsewhere in the source module or in some other source module.

Some other rules concerning storage-class specifiers are as follows:

- Objects and functions declared or defined outside a function without a storage-class specifier are assumed to have `extern` storage class.
- The `auto` and `register` storage-class specifiers are invalid outside a function.
- Only the `register` storage-class specifier is valid within a function's parameter type list or parameter declaration list.

For syntactic convenience, the `typedef` specifier is sometimes listed as a storage-class specifier. See the "Type Definitions" section later in this chapter for information on how to use the `typedef` specifier.

The `auto` Storage Class

The `auto` storage-class specifier tells the compiler that an object identifier has automatic storage duration. Automatic storage duration is temporary. With this storage class, an object is allocated storage on each normal entry into the block with which it is associated, or on a jump from outside the block to a labeled statement in the block or in an enclosed block. Storage is freed when execution of the block ends in any way.

You can use the `auto` specifier only for identifiers that are located inside a block (function body or compound statement). It is a common programming practice to omit the `auto` specifier because the `auto` storage class is the default inside a block for declarations of identifiers other than functions. The `extern` storage class is the default for function declarations.

In the following example, the `temp` and `char_array` variables are of the `auto` storage class. The `sort` function, like all functions declared with no storage-class specifier, is assumed to be `extern`.

```
void func(void)
{
   auto int temp;                /*  Object explicitly declared
automatic  */
  char char_array[80];        /*  Object assumed to be automatic
*/
  int sort(void);            /*  Function assumed to be extern
*/
       .
       .
       .
}
```

See the “Automatic Storage Duration” section earlier in this chapter for more information on automatic data.

The `extern` Storage Class

This explanation of the `extern` storage class is divided as follows:

- using the `extern` storage-class specifier
- omitting the `extern` storage-class specifier
- declaring operating system status codes.

The explanation of how you declare operating system status codes might be helpful if your program uses VOS C subroutines or if your program reads status codes returned by the operating system.

Using the `extern` Storage-Class Specifier

The `extern` storage-class specifier tells the compiler that an identifier of an object or function is defined elsewhere in the current source module or in another source module. Typically, you use the `extern` keyword in a declaration in the current source module only when that identifier is declared in the current source module but defined in another source module.

Consider the following example. Assume that, in another source module, `array_of_numbers` is defined outside any function with no storage-class specifier. In the current source module, you could reference the array by including the following declaration:

```
extern float array_of_numbers[];
```

A declaration of an object that includes the `extern` specifier does not allocate storage, cannot contain an initializer, and need not specify the extent of an array. However, you must specify the array's extent in the external definition. A declaration that includes the `extern` specifier must assign the same type to the identifier as the type specified in the external definition.

For another example of how to use an `extern` declaration to reference an external definition, see the "Omitting the `extern` Storage-Class Specifier" section later in this chapter.

When you declare an object or function and specify the `extern` storage-class specifier, the identifier's linkage is determined as follows:

- If a definition for the identifier appears in the source module and has either file scope or external linkage, the identifier refers to this definition and has the same linkage and, if an object, the same shared characteristics.
- If no such definition appears in the source module, the identifier has external linkage.

In the following example, the object identifier `count` is declared with the `extern` specifier and then, in the same source module, is defined with the `static` specifier.

```
extern int count;         /*  Declaration with extern      */
    .
    .
    .
static int count;         /*  Definition with file scope   */
```

In the preceding example, because a definition of `count` appears in the source module and that definition has file scope, the `extern` declaration of `count` refers to this `static` object and has the same internal linkage and the same shared characteristics (that is, it is unshared) as the definition of `count`.

An `extern` object declaration must have a corresponding definition in the program if the object is referenced in an expression.

An `extern` function declaration must have a corresponding definition in the program if it is referenced in an expression or if the `-table` argument was used when compiling the program. The `-table` argument causes all functions declared within the source module to be treated as though they were referenced.

Certain functions and objects that are declared `extern`, such as `s$` subroutines, `e$` error messages, and VOS C library functions, are defined in the operating system or are support routines. Therefore, the source modules that comprise the program module do not need to contain the definitions of these objects and functions. See the "Declaring Operating System Status Codes" section, later in this chapter, for an example of this type of declaration.

Omitting the `extern` Storage-Class Specifier

When you omit a storage-class specifier in a declaration, an object declared outside a function or a function declared inside or outside a function has the `extern` storage class by default. In these cases, the compiler interprets the `extern` storage-class (without an explicit `extern` specifier) as follows:

- When you declare an **object** outside a function and do not include a storage-class specifier, you are defining the object. This object has file scope and external linkage. Elsewhere in the current source module or in another source module, you can use the `extern` specifier in a declaration to reference the object.
- When you declare a **function** and do not include a storage-class specifier, you define the function elsewhere in the current source module or in another program source module. The linkage of the function identifier is determined exactly as if the identifier were declared with the `extern` specifier. See the preceding section, "Using the `extern` Storage-Class Specifier," for information on how the compiler determines the linkage of a function identifier declared `extern`.

With the `extern` storage class, you define a data item exactly once in all source modules comprising the program. An *external definition* is located outside any function and cannot include any storage-class specifier. Only the external definition actually allocates storage and can contain initializers. See the example in Figure 3-5.

Source Module #1:

```
char c_array[5] = "abcd";  /*  Definition can contain an initializer
*/

int int_array[10];        /*  Definition must specify the          */
                         /*  number of elements in the array       */
extern void func(void);

main()
{
   .
   .
   .
}
```

Source Module #2:

```
extern char c_array[5];       /*  Declaration cannot contain        */
                             /*  an initializer                     */
void func(void)
{
   extern int int_array[];    /*  Declaration need not specify the  */
                             /*  number of elements in the array    */
   .
   .
   .
}
```

Figure 3-5. External Definition Example

Notice that, in Source Module 1 (Figure 3-5), an external **definition** does not include the `extern` keyword. The definition must be fully specified. The definition must indicate the number of elements in an array either explicitly in brackets following the array name or implicitly with an initializer list. Any specified initialization occurs only once before program execution. An initializer must be a constant expression.

In Source Module 2, the declarations of `c_array` and `int_array` include the `extern` keyword and reference the external definitions specified in Source Module 1.

Declaring Operating System Status Codes

A special case related to `extern` declarations and external definitions involves declaring operating system status codes. An operating system status code is a 2-byte integer with a name and, often, an associated line of text. There are four types of status codes:

- error messages, whose names begin with `e$`
- message codes, whose names begin with `m$`
- query codes, whose names begin with `q$`
- response codes, whose names begin with `r$`.

If you declare a status code name as an object having the `extern` storage class, you can use the more meaningful name, rather than a status code number, in conditional statements and other constructs that test for a specific condition. For example:

```
if (read_code == e$end_of_file)
      then exit(-1);
```

With the appropriate declaration, the variable identified by the status code name is initialized to the appropriate status code number. For example, with the following declaration, the `e$end_of_file` variable is initialized to the value 1025, the appropriate error code number.

```
short e$end_of_file;        /*  Declared outside any function  */

main()
{
   .
   .
   .
}
```

In the preceding example, the declaration of `e$end_of_file` appears **outside** any function and can include or omit the `extern` keyword. When the variable's identifier is declared as a `short` having external linkage and the variable is not initialized, the binder recognizes these external names as status codes. It initializes each variable to the associated status code.

The `ext_shared` Storage Class

The `ext_shared` storage-class specifier tells the compiler that an object will be shared by multiple tasks in a tasking application. This specifier indicates that, if a set of tasks executes the same program, each reference to the `ext_shared` identifier is a reference to the same object. An `ext_shared` object has external linkage and static storage duration.

An external-shared object can be defined outside or inside a function using the `ext_shared` specifier. In another source module, you can reference the definition for an external-shared object by declaring the same identifier with the `ext_shared` specifier. When you use the `ext_shared` specifier in a declaration, you are always a defining the object. In a tasking program, an `ext_shared` object cannot be initialized with the value of an object having external linkage.

In the following example, if the declarations and function definition were part of a tasking application, each task that executes `func` would reference the same allocation of `shared_variable` and `shared_file_ptr` but would reference a separate allocation of `unshared_variable`.

```
ext_shared int shared_variable;
ext_shared FILE *shared_file_ptr;
int unshared_variable;

void func(void)
{
   shared_variable = unshared variable;
   fprintf(shared_file_ptr, "%d", shared_variable);
       .
       .
       .
}
```

In other source modules, you could reference the variables in the preceding example by declaring them and specifying the `ext_shared` storage class for each variable.

Parameter Storage Class

All objects declared as formal parameters to a function have the parameter storage class. With this storage class, an object is allocated storage by the calling function. An identifier of the parameter storage class has automatic storage duration. Thus, parameters are temporary objects, available only during the function's execution.

An identifier of the parameter storage class has no linkage and is, in effect, declared at the beginning of the function body's compound statement. Therefore, such an identifier cannot be redeclared in the function body except in an enclosed block.

With the parameter storage class, initialization is not meaningful. When a function is called, the arguments are evaluated, and each parameter is assigned the value of its corresponding argument. Certain argument and parameter conversions may take place before the parameters are assigned a value. If a parameter has no corresponding argument, the parameter's initial value is unpredictable. A reference to a parameter that has not been assigned an initial value can cause an error condition to occur or will produce unpredictable results. See the "Argument and Parameter Conversions" section in Chapter 6 for information on these conversions.

In the following program, the `i_num` and `pointer` variables are parameters of `func` and are of the parameter storage class.

```
void func(int, char *);

main()
{
   int num = 99;
   char string[80];

   strcpy(string, "abcde");

   func(num, string);
}

void func(int i_num, char *pointer)
{
  printf("The value = %d\n", i_num);         /*  The value = 99     */
  printf("The string = %s\n", pointer);       /*  The string = abcde
*/
}
```

When `func` is called, `i_num` is assigned the value of the `num` variable, and `pointer` is assigned the address of the `string` array.

The **`register`** Storage Class

The `register` storage-class specifier tells the compiler that an object will be used frequently during the execution of a function or compound statement. Therefore, the `register` specifier indicates that, if possible, the object's value should be stored in a machine register. You can use the `register` specifier for variables that appear inside a block (function body or compound statement) and for function parameters. For example:

```
float func(register float score)   /*  Parameter using register
specifier */
{
   register float multiplier;      /*  Variable using register
specifier  */
   int i;
   for (i = 0; i > 999; i++)
        {
        register int count = 0;    /*  Variable inside compound
statement */
         .                   /*  using register specifier          */
           .
           .
        }

}
```

The compiler ignores the `register` designation if the object's data type cannot be stored in one of the machine's registers or if there are too few machine registers available. When the object cannot be allocated a machine register, the object is treated as an `auto` or parameter data item, depending on where it is declared.

Whether or not an object declared with the `register` specifier is allocated a machine register or space on the stack, the address of any part of the object cannot be calculated. This restriction means that, on an object declared as `register`, you cannot use the address-of operator (`&`). Furthermore, you cannot reference an array declared using the `register` specifier with any operator other than the `sizeof` operator.

The `static` Storage Class

The `static` storage-class specifier indicates internal linkage and tells the compiler one of the following:

- When an object's declaration appears outside a function, the `static` storage-class specifier indicates that the identifier has static storage duration and file scope (that is, the static data is permanent and is visible only in the current source module).
- When an object's declaration appears inside a function, the `static` storage-class specifier indicates that the identifier has static storage duration and block scope (that is, the static data is permanent and is visible only within the block where it is declared).
- When a function's declaration appears outside or inside a function, the `static` storage-class specifier indicates that the function is defined in the source module where the declaration is located. Also, a function definition with the `static` specifier says that the function will not be visible in other source modules bound into the program module.

The following example contains three identifiers specified as `static`.

```
static void function_1(void);      /*  Function defined in and known
only    */
                              /*  in this source module             */

static int number_var;           /*  Variable with permanent storage
that  */
                            /*  is known only in this source module   */

static void function_1(void)
{
  static char char_array[100];   /* Variable with permanent storage
*/
     .                       /*  that is known only in this block     */
       .
       .
}
```

Static data, such as `number_var` and `char_array` in the preceding example, is permanent. Static data retains its value even when the program is executing outside the object's scope. When it is important to retain the contents of a variable after program control has exited the function that defines the variable, use the `static` storage class. Notice that the `static` specifier does not affect the scope of `char_array`. This variable is known only within the block in which it is declared.

Objects specified `static` are allocated and initialized once, when the program module is loaded for execution. For this reason, to initialize objects declared with the `static` specifier, you use a constant expression, one whose value can be determined at compile time.

See the “Static Storage Duration” section earlier in this chapter for more information on static data.

Type Specifiers

In a C declaration, *type specifiers* are keywords that can be combined to indicate the data type of an object or function. The following list contains the combinations of type specifiers that are valid in VOS C. Each data type on the same line is internally represented in an identical manner. The keywords that specify each data type can appear in any order.

```
signed char
char, unsigned char
short, signed short, short int, signed short int
unsigned short, unsigned short int
int, signed, signed int, or no type specifier
long, signed long, long int, signed long int
unsigned long, unsigned long int
float
double
struct tag
union tag
enum tag
void
char_varying(n)
```

A programmer-defined `typedef` name is a synonym for a type and can be used wherever a type is used. If you do not indicate a type specifier in a declaration, the type is assumed to be `int`. The only exception to this default type is with bit-field declarations, where the type is assumed to be `unsigned int`. In any declaration, specifying each identifier’s type explicitly is a preferred programming style.

See Chapter 4 for information on each data type.

Type Qualifiers

In a C declaration, *type qualifiers* are keywords that modify certain characteristics of a type associated with an object. The two type qualifiers are `const` and `volatile`.

This explanation of type qualifiers is divided as follows:

- the `const` type qualifier
- the `volatile` type qualifier
- syntax and type qualifiers.

The `const` Type Qualifier

The `const` type qualifier tells the compiler that an object must not be modified. The program must not assign a value to the object or change the object's value in any way, such as through the use of the `++` or `--` operators.

The compiler may place a `const` object that is not declared as `volatile` in a read-only region of storage. In addition, the compiler may not allocate storage for such an object if its address is never used.

You can initialize an object declared with the `const` type qualifier. For example:

```
const int num = 100;
```

However, after an object is declared to be a `const`-qualified type, its value cannot be modified in any way.

The `volatile` Type Qualifier

The `volatile` type qualifier tells the compiler that an object may be modified in ways not represented in the program itself. The compiler does **not** optimize out any reference to an object declared as `volatile`. In addition, the object's memory location is accessed on every operation involving that variable.

You might use the `volatile` type qualifier for situations when the compiler would not be aware of modifications or side effects involving the variable. These situations might include an object that shares memory locations among tasks, an object that corresponds to a memory-mapped I/O port, or an object accessed by a special hardware-control register or modified by an asynchronously interrupting function.

Consider the following declarations that use the `volatile` qualifier.

```
extern volatile short int lock_word;

extern const volatile int hardware_reg2;
```

The first declaration uses the `volatile` qualifier for `lock_word`. This variable might be an object that will be modified by the program or by another program in an application using shared virtual memory. The second declaration uses the `const` and `volatile` qualifiers for `hardware_reg2`. This variable might be a special hardware-control register that will be frequently modified by system hardware but that cannot be modified by the program.

In VOS C, you can also use the `volatile` qualifier in the declaration of a function type. A `volatile` function is one that can return program control to some point in the calling routine other than the point of the function invocation. Also, with a `volatile` function, the state of the calling routine is unknown (and possibly changed) when program control returns to the calling routine. The `setjmp` library function is declared with the `volatile` type qualifier.

Syntax and Type Qualifiers

When a type qualifier is used to modify the characteristics of an **object**, the qualifier affects what can be done with the object, but it does not change the basic data type. For example, a `const`-qualified object of the type `int` has the characteristics of the `int` data type. However, you cannot modify the object.

Some general rules concerning the use of type qualifiers are as follows:

- Type qualifiers, specified for a basic type, modify the type associated with **all** objects or functions identified in a declarator list, not just the first declarator.
- Type qualifiers can be used together in the same specifier list or qualifier list.
- The same type qualifier cannot appear more than once in the same specifier list or qualifier list, either directly or through one or more type definitions.
- The order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.
- Two types are compatible if they are identically qualified versions of a compatible type.

The two sections that follow explain the syntax rules for using a type qualifier in the declaration of a derived type and a pointer type.

> **Note:** Though most of the examples in the next two sections use the `const` type qualifier, the VOS C compiler uses the same syntactic and semantic rules to interpret declarations that include the `volatile` type qualifier.

Type Qualifiers and Derived Types

This section tells you how to use type qualifiers when declaring derived types other than pointers. Though enumerated types are not considered derived types, most of the syntax rules that apply to `struct` and `union` types also apply to `enum` types. Therefore, the syntax rules for enumerated types are also included in this section. See the next section, "Type Qualifiers and Pointer Types," for information on using type qualifiers with pointers.

In VOS C, if you specify the `volatile` type qualifier in the declaration of a function type, the compiler associates the qualifier with the function itself and **not** the function's return type. Thus, it is possible to have a `volatile` function. It is illegal to specify the `const` type qualifier in the declaration of a function type.

Except with a type definition, if you specify a type qualifier in the declaration of an array, the qualifier modifies the array's individual elements, not the array name.

You can use type qualifiers in the declaration of structures, unions, and enumerations. In a `struct`, `union`, or `enum` declaration, the type qualifier immediately precedes the type keyword or the variable name. The type qualifier **cannot** appear immediately before or after the tag. The following equivalent declarations show the allowed locations for a type qualifier in a structure declaration.

```
struct s_tag
   {
   int num1;
   int num2;
   } type_qualifier ex_struct = {100, 200};


type_qualifier struct s_tag
   {
   int num1;
   int num2;
   } ex_struct = {100, 200};
```

With both of the preceding declarations, the type qualifier applies to all members of the structure or union. **Type qualifiers do not affect** `struct`, `union`, or `enum` tag definitions. In effect, the qualifier modifies one or more declarators but does not define a new type. Consider the following structure declarations.

```
const struct s_tag
   {
   int num1;
   int num2;
   } one_struct = {100, 200};

struct s_tag another_struct;
```

In the first declaration, the `one_struct` structure variable is qualified by `const`. You cannot change the values stored in the members of `one_struct`. In the second declaration, the `another_struct` structure variable is not qualified by `const`. Therefore, you can change the values stored in the members of `another_struct`.

It is illegal to declare a tag for a structure, union, or enumerated type with a type qualifier but no declarator. If you want a type qualifier to be associated with a type, use a type definition. For example:

```
typedef const struct s_tag
   {
   int num1;
   int num2;
   } s_type;

s_type a_struct;
```

In this example, the `const` type qualifier is associated with the newly defined type, `s_type`. Therefore, `a_struct` is qualified by `const`.

Type Qualifiers and Pointer Types

The use of a type qualifier to declare pointers can be a source of confusion. The position of the qualifier determines the entity that is qualified. For example, in the two declarations that follow, the `const` qualifier modifies the `int` type specifier, not the pointer.

```
const int *ptr;

int const *ptr;
```

In both declarations, `ptr` is a variable pointer to a constant `int`. Notice that it does not matter whether a type qualifier precedes or follows a type specifier. The resulting type is the same. With the two preceding declarations, you can modify the address stored in `ptr`, but you cannot modify the value of the pointed-to `int` object. In contrast, consider the following declaration.

```
int * const ptr;
```

The preceding example declares `ptr` to be a `const` pointer to a variable `int`. You can modify the pointed-to `int` object, but you cannot modify the address stored in `ptr`. In this case, notice that the `const` keyword modifies the `*` modifier **immediately to its left**. Therefore, the following example declares `ptr` to be a variable pointer to a `const` pointer to a variable `int`.

```
int * const * ptr;
```

With this declaration, you can modify the value stored in `ptr` and the pointed-to `int` object. However, you cannot modify the pointer object pointed to by `ptr`.

> **Note:** The Release 11.0 VOS C compiler's interpretation of how type qualifiers affect pointer types is **not compatible** with its interpretation of the same constructs prior to Release 11.0. This change was made so that the VOS C compiler would be compatible with the ANSI C Standard.

Alignment Specifiers

In a VOS C declaration, *alignment specifiers* are keywords that indicate the alignment rules for an object or for a function return type. The two alignment specifiers are `$shortmap` and `$longmap`. The `$shortmap` specifier indicates that the shortmap alignment rules are to be used for a particular data item. The `$longmap` specifier indicates that the longmap alignment rules are to be used for a particular data item.

With a few exceptions, the `$shortmap` and `$longmap` specifiers follow the same syntax and semantic rules that apply to the `const` and `volatile` type qualifiers. See the "Syntax and Type Qualifiers" section earlier in this chapter for information on the syntax and interpretation of type qualifiers.

The exceptions are as follows:

- In the declaration of a function type, the alignment specifier applies to the **function's return type** and not to the function itself.
- In the declaration of a tag, the alignment specifier applies to the **structure, union, or enumeration tag**. In declarations that specify the tagged type for a structure, union, or enumeration object, the object inherits the alignment rules associated with the tag.
- In the declaration of a structure, union, or enumeration tag or object, the alignment specifier must appear immediately after the `struct`, `union`, or `enum` keyword.
- An alignment specifier cannot be used in a type definition unless it is a `typedef` for a structure, union, or enumeration **and** the contents of the `struct`, `union`, or `enum` type are defined within the type definition.

For information on the shortmap and longmap alignment rules and on the procedure that the compiler employs to determine which alignment method to use, see the "Data Alignment" section in Chapter 5.

Alignment Specifiers for Tagged `struct`, `union`, and `enum` Types

With a structure, union, or enumerated type that includes a tag, you place the `$shortmap` or `$longmap` specifier immediately after the `struct`, `union`, or `enum` keyword and before the tag. The compiler associates the alignment specifier with the tagged type. You **cannot** use an alignment specifier other than in the definition that specifies the contents of the tagged type. Also, when a tagged `struct`, `union` or `enum` type is defined, it is illegal to use an alignment specifier in any location other than after the type keyword and before the tag.

The following examples illustrate how to use the `$longmap` specifier in the declaration of a union tag and union object declared using that tag.

```
union $longmap u_type1
   {
   .
   .
   .
   } union1;

union u_type1 union2;                /*  Legal:  union2 inherits
longmap     */
                                  /*  alignment from the tag           */

union $shortmap u_type1 union3;      /*  Illegal:  the alignment
associated  */
                                 /*  with a tag cannot be changed       */
```

Once the `u_type1` tag is declared, all subsequent union declarations that use the `u_type1` tag are aligned according to the longmap rules. In the preceding declarations, `union1` and `union2` use longmap alignment. **Once a tag is declared, the alignment rules for that tagged type cannot be changed, even if the default alignment rules applied when the tag was declared.** In the preceding examples, the declaration of `union3` attempts to change the alignment rules associated with the tag `u_type1`. You can use an alignment specifier only when a tag is defined. A tag is defined when the contents of the `struct`, `union`, or `enum` type are specified.

The "Alignment for Nested Structures and Unions" section in Chapter 5 contains numerous examples of how to specify the `$shortmap` and `$longmap` specifiers for structures and how those alignment methods affect structure and union alignment.

Alignment Specifiers for Tagless `struct`, `union`, and `enum` Types

With a structure, union, or enumerated type that does not include a tag, you specify the `$shortmap` or `$longmap` specifier immediately after the `struct`, `union`, or `enum` keyword. The compiler associates the alignment specifier with the declared object. When a tagless `struct`, `union`, or `enum` is defined, it is illegal to specify an alignment specifier in any location other than after the type keyword.

The following examples illustrate how to use the `$shortmap` specifier in the declaration of a structure object declared without using a tag.

```
struct $shortmap
   {
   .
   .
  .                          /*  Legal:  struct1 is aligned using   */
  } struct1;                   /*  the shortmap alignment rules
*/

struct
   {
   .
   .
   .                                  /*  Illegal:  the alignment
specifier   */
  } $shortmap struct2;          /*  must follow the struct keyword
*/
```

Notice that, in the preceding examples, it is illegal to specify an alignment specifier in any location other than after the `struct` keyword.

Alignment Specifiers for Other Object Types and Function Types

To indicate the alignment rules for a function return type or for an object type other than a structure, union, or enumeration, you place the `$shortmap` or `$longmap` specifier anywhere before the declarator. Pointer types are an exception to this rule and are discussed later in this section. The following examples illustrate how to use the `$longmap` specifier in the declaration of a function type, an array type, and an object type other than a `struct`, `union`, or `enum`.

```
int $longmap func(char *s)
   {
      .
      .
      .
   }

$longmap char c_array[80];

static $longmap int s_num;
```

In the declaration of the `func` function, the alignment specifier applies to the function return type, not to the function itself. In the declaration of the array `c_array`, each element of the array is aligned using the longmap alignment rules.

With a pointer type, an alignment specifier can appear within the declarator. When an alignment specifier is used with a pointer type, the syntax and semantics are similar to those used for type qualifiers. The following examples illustrate how to use the `$shortmap` specifier in the declaration of a pointer type.

```
int $shortmap *i_ptr;          /*  Example 1  */

int * $shortmap i_ptr1;        /*  Example 2  */
```

In example 1, `i_ptr` is declared to be a pointer to an `int`. The `int`, not the pointer, is associated with the `$shortmap` specifier. In example 2, `i_ptr1` is also declared to be a pointer to an `int`. However, in this case, the pointer, not the `int`, is associated with the `$shortmap` specifier.

If you specify an alignment specifier for a bit field or a function that returns `void`, the compiler ignores the specifier.

Declarators

Declarators are unique to the C language. In a C declaration, a *declarator* consists of an identifier and optionally one or more array (`[]`), function (`()`), or pointer (`*`) modifiers. In addition, with some derived types, the declarator can contain type qualifiers. Each declarator declares one identifier to have the data type and other attributes specified in the declaration. Specifically, when the identifier named in the declarator appears in an expression, it designates a function or object with the scope, storage duration, linkage, type, and alignment indicated by the declaration specifiers.

Figure 3-6 shows a sample declaration, including a declarator.

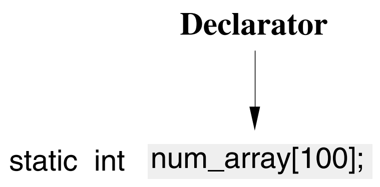


Figure 3-6. Sample Declaration

This section explains how to use declarators in an object or function declaration. See Chapter 6 for more information on declarators that are used for functions.

Simple Declarators

A simple declarator consists of an identifier without any array, function, or pointer modifiers. With a simple declarator, the declaration's specifiers provide all of the type information needed to declare the identifier. You can use simple declarators to declare all data types except arrays, functions, and pointers. Declarations using simple declarators take the following form:

```
specifier_list  identifier ;
```

The *`specifier_list`* can include a storage-class specifier, type specifiers, type qualifiers, and an alignment specifier. The *`identifier`* is any valid identifier.

The following declarations contain simple declarators.

```
char                    letter;

volatile double         d_number;

struct                  template_1
   {
   int                  x;
   int                  f;
   } two_item;
```

In the preceding examples, the declarators `letter`, `d_number`, and `two_item` declare the following:

- `letter` is a variable of the type `char`.
- `d_number` is a `volatile` variable of the type `double`.
- `two_item` is a structure of the type `struct template_1`.

Pointer Declarators

A pointer declarator consists of an identifier with a pointer modifier. With a pointer declarator, the declaration's specifiers and the declarator provide the type information needed to declare the pointer. Declarations using pointer declarators take the following form:

specifier_list * [qualifiers_and_alignment_specifier] *identifier* ;

The *specifier_list* can include a storage-class specifier, type specifiers, type qualifiers, and an alignment specifier. For each * within the pointer declarator, the optional *qualifiers_and_alignment_specifier* can contain type qualifiers and an alignment specifier. The *identifier* is any valid identifier.

The following declarations contain pointer declarators.

```
char      *char_ptr;

double    * const d_num_ptr;

float     *num_ptr[5];
```

The preceding examples declare the following:

- `char_ptr` is a pointer to an object of the type `char`.
- `d_num_ptr` is a `const` pointer to an object of the type `double`.
- `num_ptr` is an array of five pointers, each of the type pointer to an object of the type `float`.

The * modifier and optional qualifier list can be repeated to create more complex pointer types. For example, the following declaration contains a variable pointer to a `const` pointer to a variable object of the type `int`.

```
int * const *variable_ptr;
```

See the "Pointer Types" section in Chapter 4 for more information on declaring and using pointers.

Array Declarators

An array declarator consists of an identifier with an array modifier. With an array declarator, the declaration's specifiers and the declarator provide the type information needed to declare the array. Declarations using array declarators take the following form:

specifier_list identifier [[constant_expression]] ;

The *specifier_list* can include a storage-class specifier, type specifiers, type qualifiers, and an alignment specifier. The *identifier* is any valid identifier followed by brackets. Inside the brackets, *constant_expression* is a constant expression that yields an integer value greater than 0. The constant expression specifies the number of elements in the array.

You do not have to specify an integer constant expression indicating the number of elements in a single-dimensional array or the leftmost dimension in a multidimensional array in the following contexts:

- when an array is declared as `extern`
- when an array is declared with an initializer list
- when an array is declared as a function's formal parameter.

In VOS C, if an array declaration appears outside any function with no storage-class specifier and no initializer list, and the declaration omits the constant expression, the compiler assumes that declaration is `extern` (that is, the declaration is not a definition). The `extern` specifier requires that a definition for the array be located somewhere else in the source module or in another source module that will be bound into the program module.

The following declarations contain array declarators.

```
char            c_array[80];

const float     amt_array[80];

int             *ptr_array[5];
```

The preceding examples declare the following:

- `c_array` is an array containing 80 elements of the type `char`.
- `amt_array` is an array containing 80 elements of the type `const float`.
- `ptr_array` is an array containing five elements of the type pointer to `int`.

Multidimensional arrays can be declared by specifying two or more bracketed integer constant expressions in the declarator, as follows:

```
specifier_list identifier[n] [m] ;
```

Inside the brackets, *n* and *m* are expressions. Each expression yields an integer constant greater than 0, and specifies the number of elements in that dimension of the array.

See the "Array Types" section in Chapter 4 for more information on declaring and using arrays, including multidimensional arrays.

Function Declarators

A function declarator consists of an identifier with a function modifier. With a function declarator, the declaration's specifiers and the declarator provide the type information needed to declare the function. Declarations using function declarators can take one of two forms: function-prototype form or old-style form.

This section briefly discusses function-prototype form. See Chapter 6 for detailed information on both forms of function declarations and definitions.

Function prototypes give C programs the option of having the compiler check the number and types of arguments used when a function is invoked. Declarations that specify function-prototype form have the following format:

```
specifier_list identifier(parameter_type_list) ;
```

The `specifier_list` can include either the `extern` or `static` storage-class specifier, type specifiers, the `volatile` type qualifier, and an alignment specifier.

> **Note:** The `const` type qualifier cannot be specified for a function type.

The type specifier indicates the function's return type. The function's return type can be `void` or any valid type other than an array type or function type. A function's return type can be a pointer to an array or a pointer to a function. If no type specifier is given, a return type of `int` is assumed.

The `identifier` is any valid identifier followed by parentheses. The `identifier`, parentheses, and `parameter_type_list` make up the function's declarator. The `identifier` gives the function's name.

When a function declaration uses function-prototype form, the types and, optionally, the names for up to 127 parameters are enclosed in one set of parentheses. Within this `parameter_type_list`, each parameter declaration can include type qualifiers and an alignment specifier.

The identifiers for parameters within the `parameter_type_list` are optional. In a function declaration with a prototype, an identifier, if included, does **not** declare an object of the specified name. In the declaration of a function prototype, parameter identifiers are included for documentation purposes only. The identifiers allow you to associate a meaningful name with each argument position.

Some other rules concerning `parameter_type_list` are as follows:

- Only the `register` storage-class specifier is allowed for parameters in the list. If `register` is specified for a parameter in a function declaration, the compiler ignores the specifier.
- If no `type_specifier` is given, the `int` type is assumed.
- If `parameter_type_list` contains more than one parameter, they are separated by commas.
- If the `void` keyword appears instead of a list of parameters, it indicates that a function has no parameters.
- If an ellipsis (`...`) appears at the end of `parameter_type_list`, the function takes a varying number of arguments or arguments of varying types.

The following declarations contain function declarators.

```
int calculate(double average);  /* The parameter list specifies an
*/
                                /*  identifier and a type              */

void sort(void);                  /*  The empty parameter list uses
the      */
                               /*  void keyword to specify no parameters  */

char *find_name(int, int);     /* The parameter list specifies only
*/
                                /*  the parameters' types              */
```

The preceding examples declare the following:

- `calculate` is a function that returns an `int` and has a `double` parameter.
- `sort` is a function that returns no value and has no parameters.
- `find_name` is a function that returns a pointer to a `char` and has two `int` parameters.

See the "Functions with Prototypes" section in Chapter 6 for detailed information on and examples of how to declare and define functions with prototypes.

Complex Declarators

In C, you can create complex declarators by using combinations of the array (`[]`), function (`()`), and pointer (`*`) modifiers within the declarator.

The compiler evaluates declarators in much the same way that it evaluates expressions. When you use an array, function, or pointer modifier, the rules of operator precedence and associativity apply to these modifiers, determining the type of the object or function that you declare. The `[]` and `()` operators have a higher precedence than the `*` operator. Since operations within parentheses are performed first, you can use parentheses to change this order of precedence and thus the meaning of the declarator.

When specifying a pointer declarator, you can use parentheses around `*`*`identifier`* to change the order of precedence when you declare a pointer to an array or a pointer to a function. For example, the following declaration specifies that `array_ptr` is a pointer to an array containing five elements of the type `char`.

```
char  (*array_ptr)[5];
```

In the preceding example, the parentheses around `*array_ptr` cause the compiler to interpret this as a pointer to an array. Without the parentheses, the `[]` operator has a higher precedence than the `*` operator. The parentheses cause the compiler to override this normal order of evaluation. Thus, the compiler interprets `(*`*`identifier`*`)[]` as a pointer to an array, but it interprets `*`*`identifier`*`[]` as an array of pointers. Therefore, in contrast to the preceding declaration, the following declaration specifies that `char_ptr` is an array of five pointers, each of which points to a `char`.

```
char  *char_ptr[5];
```

Table 3-4 lists some valid modifier combinations and illustrates each with an example.

Table 3-4. Valid Complex Declarators

Modifier Combination	**Example**	**Description**
`**`	`int **ptr`	`ptr` is a pointer to a pointer to an object of the type `int`.
`*()`	`char *func()`	`func` is a function that returns a pointer to an object of the type `char`.
`*[]`	`int *arry[3]`	`arry` is an array of three pointers, each of which points to an object of the type `int`.
`(*)()`	`void (*ptr)()`	`ptr` is a pointer to a function that returns no value (`void`).
`(*)[]`	`char (*ptr)[3]`	`ptr` is a pointer to an array containing three elements of the type `char`.

You can devise increasingly complex declarators out of the modifier combinations shown in Table 3-4. However, some modifier combinations are invalid. Table 3-5 lists these invalid combinations.

Table 3-5. Invalid Complex Declarators

Modifier Combination	**Explanation**
`()[]`	This is an **invalid** combination, which attempts to declare a function returning an array. As an alternative, a function that returns a pointer to an array is a valid declarator.
`()()`	This is an **invalid** combination, which attempts to declare a function returning a function. As an alternative, a function that returns a pointer to a function is a valid declarator.
`[]()`	This is an **invalid** combination, which attempts to declare an array of functions. As an alternative, an array of pointers to functions is a valid declarator.

Because it is invalid to declare an array of functions or a function returning an array, you must use a parentheses between the `()` and `[]` modifiers when they both appear within a declarator. The following declarations declare the alternative complex declarators described in Table 3-5.

```
int (*func_1()) [5];

double (*func_2()) ();

char (*array[3]) ();
```

The preceding examples declare the following:

- `func_1` is a function that returns a pointer to an array of five elements of the type `int`.
- `func_2` is a function that returns a pointer to a function that returns a `double`.
- `array` is an array of three pointers, each of which is a pointer to a function that returns a `char`.

Initializers

An *initializer* specifies the initial, or first, value that will be stored in an object. In many declarations, when you declare an object, you can include an initializer. The location of a declaration and a variable's storage class and type determine which initializers are allowed in a particular declaration.

You **cannot** include an initializer for an object of the type `void`, for a function, or for a function parameter.

Within a declaration, the syntax for a declarator with an initializer is as follows:

```
declarator[= initializer]
```

For integral, floating-point, union, pointer, and enumeration objects, the initializer can optionally be enclosed in braces. However, initializers for these objects typically are not enclosed in braces.

For arrays, structures, and unions, the initializer list must begin with an opening brace and end with a closing brace. Inner braces can be included within the initializer list to enclose a subaggregate group or cluster in a structure's initializer list. These optional inner braces can be used to delineate more clearly the relationship between the initializer and the contents of the structure.

Initialization and Storage Duration

The initialization performed on objects with static storage duration is different from the initialization performed on objects with automatic storage duration. The next two sections explain these differences.

Initializing Static Data

Objects with static storage duration include variables defined outside a function and variables defined inside a function with the `extern`, `ext_shared`, or `static` storage-class specifier. In contrast to an object definition where the `extern` storage class is assumed, a declaration with an explicit `extern` specifier allocates no storage and cannot include an initializer.

Objects with static storage duration are allocated and initialized once, when the program module is loaded for execution. For this reason, you initialize an object with static storage duration with a constant expression, one whose value can be determined at compile time.

If an object with static storage duration has no explicit initialization, the compiler sets the initial value to binary 0. If a `char_varying` string has static storage duration and no explicit initialization, the compiler sets all bytes in the current-length field and in the string itself to binary 0.

Initializing Automatic Data

Objects with automatic storage duration include variables declared inside a function with no storage-class specifier or with the `automatic` or `register` storage-class specifier. You can include an initializer in the declaration of an automatic object other than an array, structure, or union. You initialize an object with automatic storage duration with an arbitrary expression. Initializers for automatic objects do not have to be constant expressions because the initial value of automatic data is determined at run time, not at compile time.

Automatic data with no initialization specified has unpredictable initial values. In practical terms, this lack of automatic initialization means that you must explicitly initialize an automatic object if its initial value is important. For an automatic object without an explicit initializer, you cannot assume that the initial value is 0. If a `char_varying` string has automatic storage duration and no explicit initialization, both the current-length field and the string itself have unpredictable values.

For automatic data, initial values are set upon each normal entry into the block: when the function is invoked or when control enters a compound statement. If control enters a compound statement by a transfer to a labeled statement, initial values, if any are specified, are not set for automatic data within that block.

> **Note:** In VOS C, the declaration of an array, structure, or union with automatic storage duration cannot include an initializer. If you need to initialize an array, structure, or union that is declared inside a function, you can make an assignment to the object in a statement after the function's declarations have been listed.

Constant Expressions and Arbitrary Expressions

All of the expressions in an initializer or initializer list must be constant expressions if the initializer is specified for an object that has static storage duration.

All of the expressions in an initializer for an object that has automatic storage duration can be arbitrary expressions.

The following sections describe what is meant by a constant expression and an arbitrary expression.

Constant Expressions as Initializers

A *constant expression* is an expression that can be evaluated at **compile time** to a constant. Each constant expression evaluates to a constant that is in the range of representable values for its type. As an initializer, a constant expression can evaluate to one of the following:

- an integral constant expression
- an arithmetic constant expression
- an address constant
- an address constant for an object type plus or minus an integral constant expression
- a null pointer constant.

An *integral constant expression* has an integral type. The operands in an integral constant expression can be character constants, integer constants, enumeration constants, and `sizeof` expressions. If the operand of `sizeof` is a `$substr` expression, the second and third arguments to `$substr` must be constant expressions. Cast operators in integral expressions can only be used to convert arithmetic types to integral types, except when the cast is part of an operand to the `sizeof` operator.

An *arithmetic constant expression* is a constant expression having an arithmetic type. You can, for example, use an arithmetic constant expression to initialize an object having an arithmetic type. The operands in an arithmetic constant expression can be character constants, integer constants, enumeration constants, floating-point constants, and `sizeof` expressions. If the operand of `sizeof` is a `$substr` expression, the second and third arguments to `$substr` must be constant expressions. If you use a cast operator as part of an arithmetic constant expression, the cast can convert an arithmetic type only to another arithmetic type. In VOS C, floating-point arithmetic is **not** allowed in an arithmetic constant expression.

The following declaration contains an arithmetic constant expression that specifies the initial value of a `float` variable.

```
float f_num = (float)999.999;
```

An *address constant* is a pointer to an lvalue designating an object with static storage duration, or is a pointer to a function. You can, for example, use an address constant expression to initialize a pointer variable. You create an address constant explicitly by using an expression containing the address-of operator (`&`), or implicitly by using an expression of array or function type. To create an address constant, you can use pointer casts and the following operators: array subscript (`[]`), structure/union member (`.`), structure/union pointer (`->`), address-of (`&`), and indirection (`*`) operators. However, in creating the address constant, the value (as opposed to the address) of an object cannot be accessed using any of these operators.

The following declarations contain address constants that specify the initial value of a pointer variable.

```
int i_num, *num_ptr = &i_num;

char array[10], *array_ptr = &array[9];
```

A *null pointer constant* is an integer constant expression equal to the value 0, or such an expression cast to `void *`. Initializing a pointer to `NULL` indicates that the pointer holds no valid storage location. In VOS C, the `OS_NULL_PTR` constant is also a constant expression that you can use as an initializer. The "Null Pointer Constants" section in Chapter 4 contains more information on these types of constant expression.

See the "Constant Expressions" section in Chapter 7 for information on the operators allowed in constant expressions.

Arbitrary Expressions as Initializers

As an initializer, an *arbitrary expression* is an expression that can be evaluated at **run time**. An arbitrary expression can contain operands whose values are determined at run time. The following declarations contain arbitrary expressions that specify the initial values of objects declared at the beginning of a function body.

```
void func(int int_num, float flt_num)
{
   int i_num = int_num + 100;

   double d_num = flt_num / 1.1;
       .
       .
       .
}
```

Notice that, in the initializer for `d_num`, floating-point arithmetic is allowed in an arbitrary expression.

Integral Object Initializers

An initializer for an integral object with static storage duration must be a constant expression. An initializer for an integral object with automatic storage duration can be a constant expression or an arbitrary expression. When the initializer's value is assigned to the integral object, the same conversions as for simple assignment apply.

The following code fragment shows examples of valid initializers for integral objects.

```
ext_shared int trans_code = 10;            /*  Constant expression  */
char letter = 'Y';                         /*  Constant expression  */

void func(unsigned int);

main()
{
   static unsigned int state = 99;         /*  Constant expression  */
    func(state);
        .
        .
        .
}

void func(unsigned int state_param)
{
   int num1 = -5 * 456;                    /*  Constant expression   */
   short int num2 = (short)state_param / 3;   /*  Arbitrary expression
*/
   register long num3 = num2 * 10;         /*  Arbitrary expression  */
        .
        .
        .
}
```

As shown in the preceding examples, an initializer for an object with automatic storage duration can be either a constant expression or an arbitrary expression. An arbitrary expression contains operands whose values are determined at run time.

Enumeration Initializers

An *enumeration* is an object of an enumerated type. An initializer for an enumeration with static storage duration must be a constant expression. An initializer for an enumeration with automatic storage duration can be a constant expression or an arbitrary expression. When an initializer's value is assigned to an enumeration, the same conversions as for simple assignment apply.

Typically, an enumeration is initialized with any other object having the same enumerated type or with any of the enumeration constants associated with the enumerated type. By default, the compiler treats an enumeration constant as identical to an expression of the type `int`. Thus, you are not restricted to initializing an enumeration with an object or constant of the associated type.

The following examples show valid and invalid enumeration initializers.

```
enum color {black, red = 1, white, pink} hue = black;

enum color shade = red;

enum color tone = hue;    /*  Illegal:  hue is not a constant expression,  */
                         /*  which is required for static data            */
main()
{
   enum color tone = hue;          /*  Arbitrary expression whose value  */
                                   /*  can be determined at run time     */

   enum color tint = red + white;    /*  Diagnosed by check_enumeration   */

   printf("tint = %d\n", tint);      /*  tint = 3                          */
}
```

As shown in the preceding examples, an enumeration is, in effect, an `int` variable that stores a limited set of values. Therefore, an initializer can be any one of the following:

- an enumeration constant, as in `shade = red`
- another object of the enumerated type, as in `tone = hue`
- an arbitrary expression yielding an `int`, as in `tint = red + white`.

If you specify the `check_enumeration` option in a `#pragma` preprocessor control line or specify the corresponding compiler argument, the compiler checks that an enumeration or enumeration constant is assigned to or compared with another item of the same enumerated type. With the `check_enumeration` option, the compiler also diagnoses all instances where an arithmetic operator has an enumeration or enumeration constant as an operand, such as `red + white` in the preceding examples.

See the “Enumerated Types” section in Chapter 4 for more information on enumerations.

Floating-Point Object Initializers

An initializer for a floating-point object with static storage duration must be a constant expression. The compiler will not allow an initializer that requires any floating-point arithmetic except you can precede the floating-point constant with the unary minus operator (`-`). An initializer for a floating-point object with automatic storage duration can be a constant expression or an arbitrary expression, where floating-point arithmetic is allowed. When the initializer’s value is assigned to the floating-point object, the same conversions as for simple assignment apply.

The following code fragment shows examples of valid and invalid initializers for floating-point objects.

```
ext_shared float flt_num = (float)10;  /*  Constant expression: integer */
                                      /*  constant cast to a float         */

double d_num = -10e10;                 /*  Constant expression             */

main()
{
  static float s_flt1 = .10e5;         /*  Constant expression             */

  double d_num1 = s_flt1 / .22;        /*  Legal:  floating-point          */
                                      /*  arithmetic for automatic objects  */

   static float s_flt2 = .10e5 * .22;  /*  Illegal:  no floating-point  */
                                      /*  arithmetic for static objects     */
       .
       .
       .
}
```

Notice that, in VOS C, for an object with static storage duration such as `s_flt2`, it is illegal to use an initializer that requires the compiler to perform a floating-point computation.

Also, initializers for floating-point objects can be integral constants. In the preceding example, the initializer for `flt_num` uses a cast to convert an integer constant to a `float`, thus avoiding implicit conversion.

Array Initializers

For an array with static storage duration, you can use an initializer list to set the initial values of the array's individual elements. In VOS C, you cannot initialize an array with automatic storage duration. For an array with static storage duration, each initializer within the initializer list must be a constant expression. When an initializer's value is assigned to an array element, the same conversions as for simple assignment apply.

Some other general rules concerning the use of array initializers are as follows:

- If you do not explicitly specify an array's size within brackets following the array name, the compiler determines the size from the number of elements in the initializer list.
- If there are fewer initializers in the list than elements in the array, elements without a specified initial value are set to binary 0.
- If there are more initializers in the list than elements in the array, the compiler issues an error message.

Initializing Single-Dimensional Arrays

The initializer list for a single-dimensional array begins with an opening brace and ends with a closing brace. For example:

```
int array[4] = { 10, 20, 30, 40,} ;
```

The comma following the last initializer in the list is optional. With the preceding initializer list, the initial values of the elements of `array` are as follows: `array[0]` is 10, `array[1]` is 20, `array[2]` is 30, and `array[3]` is 40.

Initializing Multidimensional Arrays

The initializer list for a multidimensional array assigns values to elements using row-major order. The term *row-major order* means that the first subscript after the array name specifies a row and the second subscript specifies a column, as follows:

```
multidimensional_array [row] [column]
```

In row-major order, if elements are accessed in the order in which they are stored, the rightmost subscript varies most frequently, and the leftmost subscript varies least frequently. Consider the following multidimensional array declaration and initializer list.

```
int md_array[3][4] = {
                         {10, 20, 30, 40},
                    {50, 60, 70, 80},    /*  Use a comma at the end of  */
                    {90, 100, 110, 120}  /*  each row except the last   */
                         };
```

This array contains three rows, and each row contains four elements. The initial values of the elements of `md_array` are shown in Table 3-6.

Table 3-6. Two-Dimensional Array `md_array[3][4]`

	[0]	[1]	[2]	[3]
md_array[0]	10	20	30	40
md_array[1]	50	60	70	80
md_array[2]	90	100	110	120

When you initialize a multidimensional array, use braces to enclose a subaggregate cluster or group of initial values. If there are more elements than initial values in a group, all elements in the group without a specified initial value are set to binary 0. Consider the following declaration in which `three_d_array` contains two groups of three elements, each element of which has four elements.

```
int three_d_array[2][3][4] = {
                             {5, 6, 7},
                             {8, 9, 0, 1},
                             {2, 3, 4},

                             {5, 6, 7, 8},
                             {9, 0, 1},
                             {5}
                             };
```

In the preceding example, for some brace-enclosed groups of initializers, there are more elements in the group than initial values in the initializer list. Elements without initializers are set to binary 0. Therefore, the preceding initialization of `three_d_array` is the equivalent of the following:

```
int three_d_array[2][3][4] = {
                             {5, 6, 7, 0},
                             {8, 9, 0, 1},
                             {2, 3, 4, 0},

                             {5, 6, 7, 8},
                             {9, 0, 1, 0},
                             {5, 0, 0, 0}
                              };
```

Even if a multidimensional array's declaration includes an initializer list, you must explicitly specify within brackets the size of all but the first dimension of the array.

Initializing Arrays of Characters

The initializer list for an array of type `char` can be either a brace-enclosed list of character constants or a character-string literal. For example, the following two declarations are equivalent.

```
char array_of_char[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

char array_of_char[6] = "Hello";
```

Notice that, when the initializer is a character-string literal, the compiler automatically appends a null character if the array's size is large enough or if the size is not specified within brackets. Opening and closing braces are optional when the initializer is a character-string literal.

You can also initialize an array of character pointers with a list of character-string literals. For example:

```
char *array_of_ptrs[3] = { "Hello", "Goodbye", "Wait a minute"} ;
```

Initializing Arrays of Structures

An initializer list for an array of structures is similar in shape to an initializer list for a multidimensional array. Consider the following declarations, which include an array of structures with an initializer list.

```
struct s
   {
   char letter;
   int number;
   };

struct s array_of_s[2] = {
                          {'Y', 1},
                          {'N', 0}
                          };
```

In the preceding example, `array_of_s` contains two structures, and each structure has two members. The initial values of the members of `array_of_s` are as follows:

- `array_of_s[0].letter` is `Y`, and `array_of_s[0].number` is `1`.
- `array_of_s[1].letter` is `N`, and `array_of_s[1].number` is `0`.

Structure Initializers

For a structure with static storage duration, you can use an initializer list to set the initial values of the structure's individual members. In VOS C, you cannot initialize a structure with automatic storage duration.

An initializer list for a structure is similar in shape to an initializer list for a multidimensional array. Structure members are initialized in the sequence in which they are declared. Each initializer within the initializer list must be a constant expression. When an initializer's value is assigned to a member, the same conversions as for simple assignment apply.

Many of the rules that apply to the initialization of multidimensional arrays also apply to the initialization of structures.

- Braces enclose a subaggregate cluster or group of initial values.

- If there are fewer initializers in the list than members in the group, all members without a specified initial value are set to binary 0.

- If there are more initializers in the list than members in the group, the compiler issues an error message.

See the "Initializing Multidimensional Arrays" section earlier in this chapter for more information on and examples of the preceding rules.

In the following example, the declaration of the `ex_struct` structure includes an initializer list.

```
struct s
   {
   double d_num[3];
   struct
      {
      char letter;
      int  num1;
      int  num2;
      };
   char ch1;
   char ch2;
   };

struct s ex_struct = {
                     {.333, .666},  /* Initializers for the nested
array     */
                     {'A', 100},    /* Initializers for the nested
structure */
                     'z'
                     };
```

Because there are more structure members than initializers, the preceding declaration of `ex_struct` is the equivalent of the following:

```
struct s ex_struct = {
                     {.333, .666, 0.0},
                     {'A', 100, 0},
                     'z',
                     '\0'
                     };
```

As with the initialization of a multidimensional array, every structure member without a specified initializer is set to binary 0.

Union Initializers

For a union with static storage duration, you can use an initializer to set the initial value of the union's **first member**. In VOS C, you cannot initialize a union with automatic storage duration. The initializer is enclosed in braces and must be a constant expression. When an initializer's value is assigned to the first union member, the same conversions as for simple assignment apply.

In the following example, the declarations of `ex_union1` and `ex_union2` illustrate how to initialize a union.

```
union u1
   {
   double d_num;
   int i_num;
   };

union u1 ex_union1 = {2.22};

union u2
   {
   struct
      {
      int i;
      char ch;
      } inner_struct;
   float f_num;
   };

union u2 ex_union2 = {100, 'X'};
```

The initializer for `ex_union1` is a single value because this union's first member is a variable of the type `double`. In contrast, the initializer list for `ex_union2` contains two values because this union's first member is a structure composed of two members: an `int` variable and a `char` variable.

Pointer Initializers

An initializer for a pointer with static storage duration must be a constant expression. Typically, the initializer expression is an integral constant expression, an address constant, an address constant for an object type plus or minus an integral constant expression, or a null pointer constant. An initializer for a pointer with automatic storage duration can be a constant expression or an arbitrary expression. When an initializer's value is assigned to a pointer, the same conversions as for simple assignment apply.

The type of the pointed-to object stored at an address specified in a pointer initializer should correspond to the type of the pointer. For example, you use the address of an `int` object to initialize a pointer that is declared as a pointer to an `int`.

The following examples show valid pointer initializers.

```
#define NULL ((void *)0)              /*  Defines the NULL pointer
constant  */

int num;
int *num_ptr = &num;                  /*  Points to the address of
num       */
```

(Continued on next page)

```
char array[10];
char *array_ptr = array;              /*  Points to the address of
array[0]  */
char *address_ptr = array + 2;        /*  Points to the address of
array[2]  */

char *string_ptr = "ABCDE";          /*  Points to the address of a
string; */
                             /*  the string may not be modifiable   */
int func(void);
int (*func_ptr)(void) = func;         /*  Points to the address of
the entry */
                              /*  value associated with func        */

float *flt_ptr1 = (float *)0x00E0C18C; /*  Points to the address of
a float  */
float *flt_ptr2 = NULL;              /*  Points to the NULL pointer
constant  */
```

In the preceding examples, notice that a function name such as `func` can be, like an array name, used as an address. In VOS C, a function name does not represent the code location where a function starts executing. Instead, a function name represents an *entry value*: a three-pointer array that holds certain information about the function. See the “VOS C Entry Values” section in Chapter 6 for information on entry values.

In VOS C, character-string literals such as `"ABCDE"` may or may not be modifiable, depending on whether you specify the `no_common_constants` option in a `#pragma` preprocessor control line. See the “Character-String Literals” section in Chapter 2 for more information on declaring and using character-string literals.

Varying-Length Character String Initializers

For a `char_varying` string, you can use an initializer that consists of a character-string literal. The 2-byte length in the `char_varying` variable is automatically set to the number of characters in the character-string literal. The null character on the end of the character-string literal is not included in the current length. For example, the following declaration includes an initializer list for the `cv_string` variable.

```
char_varying(80) cv_string = "0123456789";
```

In the preceding example, the current length field in `cv_string` is set to 10. Notice that the (unseen but assumed) null character, which marks the end of the character-string literal, is not counted as part of the initializer.

Type Definitions

A *type definition* defines or names a synonym for a type. A type definition does not create a new type. An identifier specified in the declarator of a type definition is syntactically equivalent to the type specified. You can use this identifier in any context where a type is valid. Type definitions can simplify the process of declaring complex types.

Though the `typedef` keyword is categorized as a storage-class specifier for syntactic convenience, specifying a `typedef` allocates no storage. In subsequent declarations, storage may be allocated for the object or function declared to be of the defined type.

The syntax for a type definition is as follows:

```
typedef [type_qualifier] [type_specifiers] declarator_list ;
```

The keyword `typedef` introduces a type definition. You cannot include a storage-class specifier in a type definition because the keyword `typedef` is considered a storage-class specifier. The optional *`type_qualifier`* can be `const` or `volatile` or both. The *`type_specifiers`* can form any valid type. If you omit a type specifier, the defined type is a synonym for the `int` type. The *`declarator_list`* contains one or more declarators, each containing an identifier for the defined type.

Although it is not shown in the preceding syntax, an alignment specifier can be used in a type definition **if** it is a `typedef` for a structure, union, or enumeration and the contents of the `struct`, `union`, or `enum` type are defined within the type definition.

The following examples are type definitions.

```
typedef short int S_NUM;

typedef double FUNC(void);
```

Notice that a type definition, such as `FUNC`, can define a function type and can include function-prototype information. Given the preceding type definitions, the following declarations are valid.

```
S_NUM code, code_array[5];

FUNC f, *f_ptr;
```

The preceding examples declare the following:

- `code` is an object of the type `short int`.
- `code_array` is an array containing five elements of the type `short int`.
- `f` is a function that takes no arguments and returns a value of the type `double`.
- `f_ptr` is a pointer to a function that takes no arguments and returns a value of the type `double`.

A `typedef` name's scope is determined in the same manner as the scope of any other ordinary identifier. Once a `typedef` name has been established, you cannot use that name as the identifier of an object or function in an inner scope (for example, within a compound statement) or as the name of a structure or union member in the same scope or in an inner scope **without** including an explicit type specifier in the declaration. In the following example, the second declaration of `i` is invalid.

```
typedef int i;

void func(void)
{
   i;                 /*  Illegal declaration:  i is interpreted  */
   .                  /*  as a type with no identifier specified  */
   .
   .
}
```

In the preceding example, the compiler diagnoses the attempted declaration of `i`. Assuming that `i` will default to `int` results in a compile-time error.

The "Type Definitions for Structures" section in Chapter 4 contains information on how to use type definitions when declaring self-referential structures.

Other Considerations

Two other declaration-related considerations, name space and incomplete types, can affect how you declare and use identifiers.

Name Space

The compiler lists each identifier that you declare in one of four name spaces. A *name space* is a table where a certain category of identifiers is listed. It is illegal to use the same identifier for more than one entity if the identifiers are in the same name space. However, you can use the same identifier for more than one entity within the program if the identifiers are not in the same name space.

The four name spaces are as follows:

- *Statement label names*. A statement label is an identifier that is followed by a colon (`:`) and can be referenced in a `goto` statement. The `case` labels used in a `switch` statement are not statement labels.
- *Tags*. A tag is an identifier that immediately follows the `struct`, `union`, or `enum` keyword in a structure, union, or enumeration definition. In the declaration, the tag identifies the newly defined type. All tags have the same name space.
- *Member names*. This name space contains the identifiers for members of a structure or union. Member names are specified in the `struct` or `union` definition and can be referenced using the structure-member (`.`) or structure-pointer (`->`) operator. Each unique structure or union type has a separate name space for its members. Therefore, you can use the same member name in more than one structure or union definition.
- *Ordinary identifiers*. This name space contains all identifiers not included in the preceding three categories. These identifiers include object and function names, `typedef` names, and enumeration constants.

In practical terms, because an identifier has its own name space, you can use the same identifier to name more than one data item within a program. For example, in the following program, the `ex_id` identifier names a `struct` tag, an `int` variable, and a label.

```
struct ex_id {char c1; int i1;};          /*  ex_id names a structure
tag  */
struct ex_id struct_var;

int ex_id = 100;                         /*  ex_id names a variable  */

main()
{
   ex_id:                                /*  ex_id names a label     */
      printf("struct_var.i1 = %d\n", struct_var.i1);

   struct_var.i1 = ++ex_id;

   if (ex_id < 102)
      goto ex_id;
}
```

From the preceding example, it should be clear that injudicious use of the same identifier to name more than one entity in a program can become a source of confusion.

Within each name space, every identifier has its own scope and can be hidden by other declarations. See the “Internal Linkage” section earlier in this chapter for information on data hiding.

Incomplete Types

An *incomplete type* is one that describes an object whose size is not known. The incomplete types consist of arrays of unknown size, and tagged structure and union types whose contents have not been defined. The `void` type is, by definition, an incomplete type that cannot be completed. An incomplete type other than `void` must be completed within the program before it is used in a context where the incomplete type’s size is required.

- For an identifier with an incomplete array type, you complete the type by specifying the array’s size in a subsequent definition that has internal or external linkage.
- For an identifier with an incomplete tagged structure or union type, you complete the tagged type by specifying the structure or union’s contents in a subsequent definition in the same scope. The incomplete tagged type cannot be completed within an enclosed scope, where a tag declaration is a new type known only in that block.

With an incomplete tagged structure or union type, you can associate an identifier with the incomplete type when, for example, a `typedef` name is declared to be a specifier for the structure or union, or when a pointer to or a function returning the structure or union is declared.

In the following example, the declaration on line 1 associates `ex_array` with an incomplete array type because the array’s size is not known. The type definition on line 3 declares `S` to be a structure of an incomplete type because the contents of the type `struct node` are not defined before the type definition.

```
1   extern char ex_array[];    /*  ex_array is an incomplete type
*/
2
3    typedef struct node S;      /*  struct node type is an incomplete
type  */
4
5    S *struct_ptr;
6
7   char ex_array[10] = "123456789";  /*  ex_array is complete
*/
8
9    struct node                /*  struct node type is complete when
the   */
10     {                    /*  contents of the struct are defined
*/
11      int num;
12      struct node *node_ptr;
13      };
14
15   S struct_type1;

    main()
    {
       .
       .
       .
    }
```

In the preceding example, both `ex_array` and the `struct node` type are completed within the program fragment shown. On line 7, the declaration of `ex_array` specifies the array's size and completes that type. On lines 9 through 13, the definition of the `struct node` type specifies the contents of the structure and completes that type.

On line 5, notice that a pointer to an incomplete tagged structure type can be declared before the contents of the associated structure type are defined. On line 15, a structure object having the defined type `S`, which references the tagged type `struct node`, cannot be allocated until the tagged type is completed. That is, the object cannot be allocated until its size is known.

Object Declarations Summary

Some of the characteristics of a data object depend on where the object's declaration is located and what storage-class specifiers are used in the declaration. The tables in this section provide a summary of the storage-class specifiers and initializers that can be used when declaring an object in VOS C.

- Table 3-7 lists the storage-class specifiers and initializers that can be used when an object is declared outside a function.
- Table 3-8 lists the storage-class specifiers that can be used when an object is declared as a formal parameter in a function definition. The declaration of a formal parameter cannot contain an initializer.
- Table 3-9 lists the storage-class specifiers and initializers that can be used when an object is declared inside the body of a function definition.

Each table also shows the scope, linkage, and storage duration of the declared object.

Table 3-7. Object Declarations Outside a Function

Storage Class Specifier	Scope	Linkage	Duration	Initializer
none	file	external	static	constant expression
`extern`	file	external †	static	constant expression
`ext_shared`	file	external	static	constant expression
`static`	file	internal	static	constant expression

† If a definition of the object appears in the source module and has either file scope or external linkage, the identifier refers to this object and has the same linkage and the same shared characteristics. If no such definition exists, the identifier has external linkage.

Table 3-8. Object Declarations for Formal Parameters

Storage Class Specifier	Scope	Linkage	Duration	Initializer
none †	block	no linkage	automatic	not allowed
`register`	block	no linkage	automatic	not allowed

† The object has the parameter storage class.

Table 3-9. Object Declarations Inside a Function or Block

Storage Class Specifier	Scope	Linkage	Duration	Initializer
none	block	no linkage	automatic	constant expression, or arbitrary expression ‡
`auto`	block	no linkage	automatic	constant expression, or arbitrary expression‡
`extern`	block	external †	static	not allowed
`ext_shared`	block	external	static	constant expression
`register`	block	no linkage	automatic	constant expression, or arbitrary expression‡
`static`	block	no linkage	static	constant expression

† If a definition of the object appears in the source module and has either file scope or external linkage, the identifier refers to this object and has the same linkage and the same shared characteristics. If no such definition exists, the identifier has external linkage.
‡ In VOS C, an array, structure, or union with automatic storage duration cannot include an initializer.

Chapter 4: Data Types

The C language includes a variety of data types. The ANSI C Standard groups these data types into a number of different categories. For example, the C data types are divided into three general categories.

- An *object type* describes an object.
- A *function type* describes a function by specifying the return type, and the number and types of its parameters.
- An *incomplete type* describes an object but lacks the information needed to determine the object's size.

This chapter focuses on object types. Chapter 6 explains function types. The "Incomplete Types" section in Chapter 3 explains incomplete types.

This chapter's discussion of data types concentrates on the range of values that can be stored in and the allocation of each data type. It also explains the set of type-related operations allowed with each data type and any VOS C extensions or restrictions. It does not explain those operations, such as the `sizeof` and address-of (`&`) operation, that can be applied to any object operand. The last section in this chapter contains a summary of the VOS C data types.

See Chapter 7 for detailed information on each operator.

Figure 4-1 shows the scalar and aggregate types.

A *scalar type* is a type that holds a single data object and yields a single value. In VOS C, the scalar types consist of the arithmetic types, pointer types, and the varying-length character string (`char_varying`) type. As shown in Figure 4-1, the *arithmetic types* consist of the integral and floating-point types.

An *aggregate type* is a type that consists of an ordered collection of data objects. The aggregate types consist of the array and structure types.

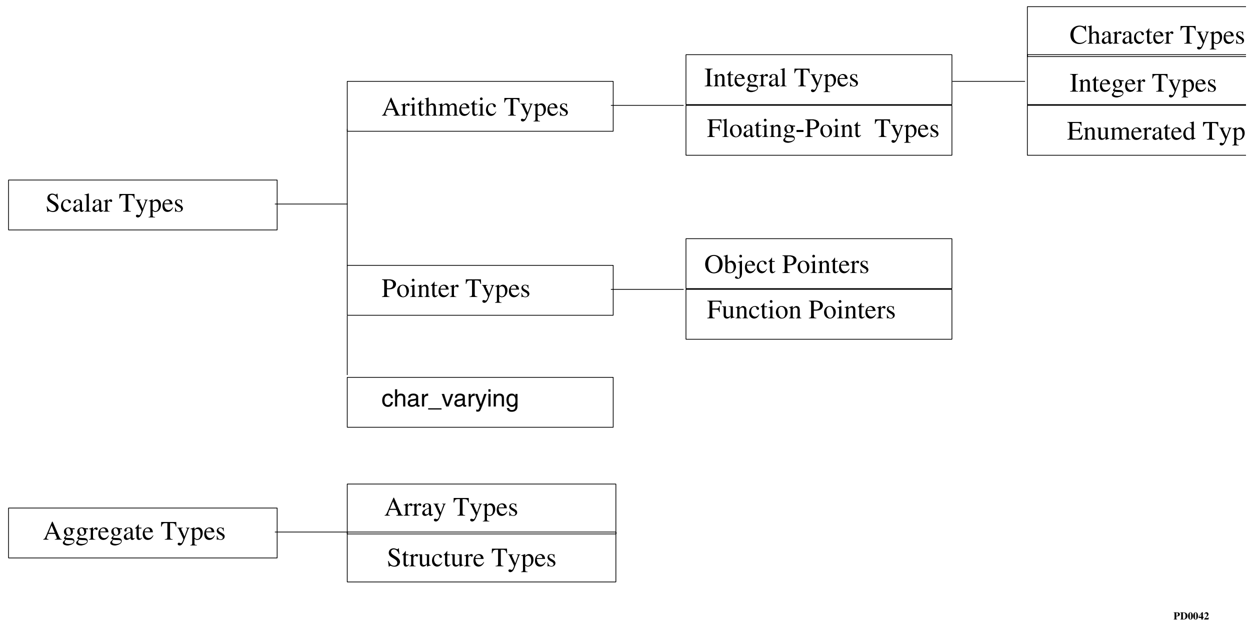


Figure 4-1. Scalar and Aggregate Types

In addition to the scalar and aggregate type categories, the C language has function types, union types, and the `void` type. Chapter 3 contains information on how you declare and initialize variables of each data type.

Integral Types

A variable declared to be an *integral type* holds a whole number, a number with no decimal point or exponent.

In C, the character types and enumerated types are integral types. Therefore, this explanation of integral data types is divided as follows:

- character types
- integer types
- enumerated types.

The VOS C integral data types include the following three sizes of data:

```
char
short int
int
```

For each of these sizes of data, you can use the `signed` or `unsigned` keyword to modify the type. In VOS C, the `long int` type is identical to the `int` type.

The `limits.h` header file lists the range of values allowed with the VOS C integral types.

Operations on Integral Types

Most VOS C operators can be used with the integral types. In addition to assignment, the operations allowed on integral types include the following:

- all arithmetic operations
- all relational and logical operations
- all bitwise operations.

The integral types can be converted to other arithmetic types or to the `char_varying` type. See the "Type Conversions" section in Chapter 5 for information on these conversions.

Character Types

The `char` data type, the smallest integral type, consists of one byte. Variables of the `char` type often hold the ASCII code for a single character from the source or execution character set. However, you can use a `char` variable in an integer expression because a `char` variable is converted to the `int` type before the expression is evaluated.

The two VOS C character types are shown in Table 4-1. For each character type, the table lists the keywords used to declare the type, the range of values allowed, and the data type's internal representation.

Table 4-1. Character Types

Data Type Keyword(s)	Internal Representation	Range
`signed char`	Signed fixed-point binary, 7-bit precision, stored in 1 byte	-128 to 127

`char`
`unsigned char`
	Unsigned fixed-point binary, 8-bit precision, stored in 1 byte	0 to 255

In VOS C, the default `char` type is `unsigned`. When you declare `char` data without an explicit `signed`, it is unsigned. The following two declarations are equivalent. In both declarations, `input_ch` is unsigned and is initialized to the value 65, the ASCII code for an `A` character.

```
char input_ch = 'A';
unsigned char input_ch = 65;
```

To declare a `signed char` variable, use the `signed` keyword. For example:

```
signed char signed_character;
```

You can change the default `char` type to `signed` by specifying the `default_char` option and the `signed` value in a `#pragma` preprocessor control line, or in the corresponding command-line argument. For more information on this option, see the "The `default_char` Option" section in Chapter 9.

It is often useful to group characters in an array. In C, a *string* is an array of type `char` that contains a contiguous sequence of characters terminated by and including the first null character (`'\0'`). For example, the following declaration initializes an array of `char` to a five-character C string.

```
char array_of_char[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The array's name, `array_of_char`, is a pointer to the address of the string's first character (see Figure 4-2). The string's length is the number of characters preceding the first null character. Therefore, the length of the string contained in `array_of_char` is five.

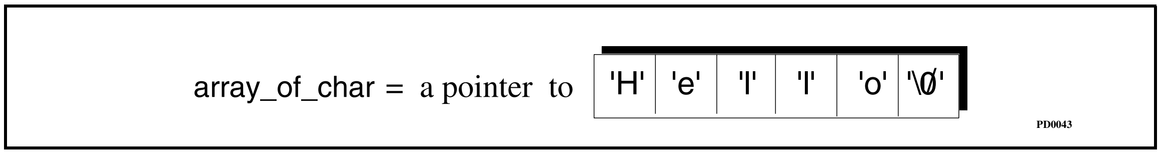


Figure 4-2. Array Name

You can use the library functions to manipulate strings. A number of features of VOS C simplify character manipulation, including the `char_varying` data type, the `$substr` built-in function, and the various string-handling functions described in Chapter 11.

Integer Types

The VOS C integer types are shown in Table 4-2. For each integer type, the table lists the keywords used to declare the type, the range of values allowed, and the data type's internal representation.

Table 4-2. Integer Types *(Page 1 of 2)*

Data Type Keyword(s)	Internal Representation	Range

`short int`
`short`
`signed short int`
`signed short`
	Signed fixed-point binary, 15-bit precision, stored in 2 bytes	-32,768 to 32,767

`unsigned short int`
`unsigned short`
	Unsigned fixed-point binary, 16-bit precision, stored in 2 bytes	0 to 65,535

`int`
`signed int`
`signed`
	Signed fixed-point binary, 31-bit precision, stored in 4 bytes	-2,147,483,648 to 2,147,483,647

`unsigned int`
`unsigned`
	Unsigned fixed-point binary, 32-bit precision, stored in 4 bytes	0 to 4,294,967,295

`long int`
`long`
`signed long int`
`signed long`
	Signed fixed-point binary, 31-bit precision, stored in 4 bytes	-2,147,483,648 to 2,147,483,647

Table 4-2. Integer Types *(Page 2 of 2)*

Data Type Keyword(s)	Internal Representation	Range

`unsigned long int`
`unsigned long`
	Unsigned fixed-point binary, 32-bit precision, stored in 4 bytes	0 to 4,294,967,295

Notice that the VOS C `int` and `long int` types are identical. Variables declared to be `int` or `long int` are four bytes long and hold the same range of values. The following examples are declarations of each of the integer types.

```
short int           index;
unsigned short int  code;
int                 signed_num;
unsigned int        count;
long int            acct_num;
unsigned long int   transaction_num;
```

In Table 4-2, notice that many integer types can be declared with keywords that vary but, nevertheless, specify the same type. In addition, the order of the keywords is not significant as long as all type keywords appear before the declarator. For example, the following declarations of a `short int` variable are equivalent.

```
short int           small_num;
short               small_num;
short int signed    small_num;
signed short        small_num;
```

Integer variables declared as `unsigned` store **non-negative** values only. Specifying `unsigned` extends the range of non-negative values that can be stored in an integer variable because the sign bit is no longer needed to indicate a negative value. If you do not explicitly specify `signed` or `unsigned` for an integer type, a signed type is assumed. For example, in the declaration `short small_num`, the type is assumed to be signed.

The following examples are declarations of and assignments to integer variables.

```
short int small_num;
int total;
unsigned long int big_num;

small_num = 0xFFFF;
total = -1;
big_num = 4294967295;
```

Enumerated Types

An *enumerated type* is the type of a user-defined enumeration, which consists of a specified set of named integer constant values. You use the keyword `enum` to declare the type and any variables of the enumerated type. An *enumeration constant* is a member of the specified set of named integer constants. Figure 4-3 shows a sample enum declaration.

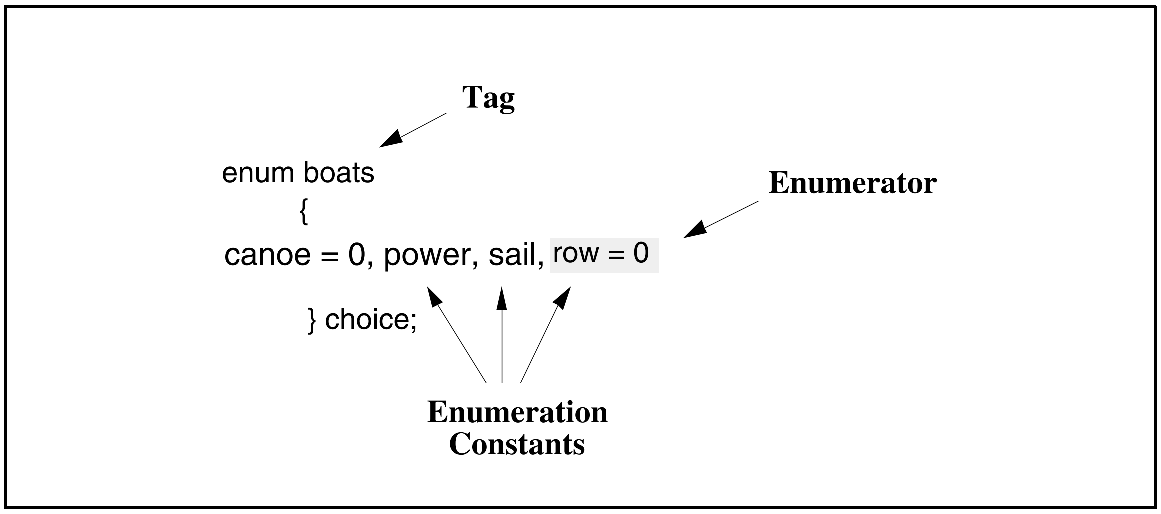


Figure 4-3. Sample Enumeration Declaration

An `enum` declaration that specifies the contents of an enumerated type has the following syntax:

enum[tag] { *enum_constant*[= *constant_expression*]

[, *enum_constant*[= *constant_expression*]] ...} [decl_list];

Although they are not shown in the preceding syntax, the declaration of an enumerated type can include type qualifiers and alignment specifiers. For information on how to use these elements when declaring an enumerated type, see the “Type Qualifiers” and “Alignment Specifiers” sections in Chapter 3.

The keyword `enum` introduces an ordered list of enumerators enclosed in braces. Following the keyword `enum`, the *tag* is an optional name that identifies this particular type of enumeration. You can include an optional *decl_list*, containing one or more identifiers naming variables of the enumerated type.

Each *enumerator* names and explicitly or implicitly defines the value of an enumeration constant. Each enumerator consists of an *enum_constant*, which must be a unique identifier. In an enumerator, the optional *constant_expression* is an integer constant expression that explicitly defines the value of each *enum_constant*. In the sample enumeration shown in Figure 4-3, `canoe` is an enumeration constant explicitly defined to have the value 0.

In the following example, the enumerated type `color` has six enumerators. Each enumerator names and defines the value of one of six enumeration constants: `red`, `green`, `blue`, `white`, `brown`, and `black`.

```
enum color
   {
   red, green, blue, white = 4, brown, black = 4
   };
```

The following rules describe how the compiler determines the value of an enumeration constant.

- If the first enumerator in the list does not contain an equals sign followed by an integer constant expression, the enumeration constant has the value 0. In the preceding example, the enumeration constant `red` equals the value 0.
- If any enumerator contains an equals sign followed by an integer constant expression, the enumeration constant has the specified value. In the preceding example, the enumeration constants `white` and `black` equal the value 4. You can specify the same value for more than one enumeration constant. Also, you can use both positive and negative values.
- Each subsequent enumerator with no specified value has the value of the preceding enumeration constant plus 1. In the preceding example, `green` equals the value 1, `blue` equals the value 2, and `brown` equals the value 5.

When an enumerated type has a tag, you can reference the type after its definition. For example, the following declaration creates two variables, `tone` and `tint`, of the enumerated type `color`.

```
enum color tone, tint;
```

Operations on Enumerated Types

Enumeration objects can be assigned to another object of the same enumerated type. In addition, an enumeration object can be compared to any other object of the same enumerated type or to any of the enumeration constants associated with the enumeration. Enumerated types are integral types. Assignments to and comparisons of enumerated data are treated as integer assignments and comparisons. In an assignment, the compiler, by default, does **not** check to ensure that the enumeration constant being assigned is a member of the enumerated type.

The following program fragment shows some sample uses of enumerated data for assignments and comparisons.

```
enum boats
   {
   canoe = 0, power, sail, row = 0
   } choice;

enum fees
   {
   no_fee, fee, fee_plus
   } fee_status, *fee_status_ptr;

float calculate_fee(enum boats choice, enum fees *fee_status_ptr);

main()
```

(Continued on next page)

```
{
   float amount;

   choice = power;                              /*  Assignments */
   fee_status = fee;
   fee_status_ptr = &fee_status;

   if (choice == canoe || choice == row)        /*  Comparisons */
      printf("Registration is optional.\n");

   if (fee_status != no_fee)
      amount = calculate_fee(choice, fee_status_ptr);
      .
      .
      .
}
```

In the preceding example, enumerated data is used for a variety of assignments and comparisons. For example, the assignment `choice = power` assigns the value of an enumeration constant, `power`, to an enumeration object, `choice`. In the second `if` statement, the comparison `fee_status != no_fee` compares the value of an enumeration constant, `no_fee`, with the value of an enumeration object, `fee_status`.

If you select the `check_enumeration` option in a `#pragma` preprocessor control line or select the corresponding compiler argument, the compiler checks that an enumeration object or constant is assigned only to objects of the same enumerated type or compared against objects or constants of the same enumerated type. The compiler diagnoses implicit conversions where an enumeration object or constant is implicitly converted to another enumerated type in such an assignment or comparison. With the `check_enumeration` option, the compiler also diagnoses all instances where an arithmetic operator has an enumeration object or constant as an operand.

If you do not select the `check_enumeration` option in a `#pragma` preprocessor control line or the corresponding compiler argument, enumerated data is treated as if it were defined as the `int` type. In this case, you are not restricted to assignments and comparisons between enumerated data and an object or constant of the associated enumerated type.

See the "The Enumeration Options" section in Chapter 9 for examples of the usages that are diagnosed with the `check_enumeration` option.

Floating-Point Types

Floating-point data is either single-precision or double-precision. The two floating-point types, `float` and `double`, are shown in Table 4-3. For each floating-point type, the table lists the keywords used to declare the type, the range of values allowed, and the data type's internal representation.

Table 4-3. Floating-Point Types

Data Type Keyword(s)	**Internal Representation**	**Range**
`float`	Floating-point binary, 24-bit precision, stored in 4 bytes	$\pm10^{-38}$ to $\pm10^{38}$
`double`	Floating-point binary, 53-bit precision, stored in 8 bytes	$\pm10^{-308}$ to $\pm10^{307}$

The `float.h` header file lists the VOS C limits for the two floating-point data types.

A variable declared to be a floating-point type holds an approximation of a real number (that is, a number expressed as a decimal fraction, such as 5.7892). Floating-point variables can be assigned values expressed as decimal fractions or values expressed using exponential notation. The value that you assign to a floating-point variable can be expressed in the following format:

$$\left[\texttt{-}\right]\ \textit{value\_part}\ \left[\begin{Bmatrix}\texttt{E}\\ \texttt{e}\end{Bmatrix}\begin{bmatrix}\texttt{+}\\ \texttt{-}\end{bmatrix}\ \textit{exponent}\right]$$

The value that you assign can begin with an optional leading unary minus operator (`-`). The components of the *`value_part`* may include an integral part, followed by a decimal point, followed by a fractional part. In exponential notation, *`value_part`* is called the *mantissa*. Both the integral and fractional parts consist of one or more digits. In *`value_part`*, either the integral part or the decimal point and fractional part must be present. Also, either the decimal point or the *`exponent`* must be present.

For a `float` variable, *`value_part`* can hold about 6 significant decimal digits. For a `double` variable, *`value_part`* can hold about 15 significant decimal digits.

The letter `e` or `E` separates *`value_part`* from the optional *`exponent`*. If the floating-point constant uses exponential notation, you specify an exponent. You can precede the exponent by a plus or minus sign to indicate its sign. The exponent represents the power of 10 by which *`value_part`* is multiplied.

The following examples are declarations of and assignments to floating-point variables.

```
float annual_rate;
double molecules;

annual_rate = 9.75;
molecules = 2.25e10;
```

In the floating-point value `2.25e10`, the *`value_part`* is 2.25, and the *`exponent`* is 10. This floating-point value is equal to 2.25 multiplied by 10^{10}, or 22,500,000,000. As another example, the floating-point value `2.25e-10` is equal to 2.25 multiplied by 10^{-10}, or 0.000000000225.

Operations on Floating-Point Types

Most VOS C operators can be used with the floating-point types. In addition to assignment, the operations allowed on floating-point types include all arithmetic, relational, and logical operations. The bitwise operations are not allowed with floating-point types.

The floating-point types can be converted to other arithmetic types or to the `char_varying` type. See the "Type Conversions" section in Chapter 5 for information on these conversions.

Pointer Types

A *pointer* is an object containing the address of a storage location. A pointer can reference or "point to" any function or object, including another pointer. You declare a pointer by including an asterisk (`*`) within the declarator. The following examples are declarations of pointers.

```
int *int_ptr;

char **str_ptr;

double (*func_ptr)(void);
```

The preceding examples declare the following:

- `int_ptr` is a pointer to an `int`.
- `str_ptr` is a pointer to a pointer to a `char`.
- `func_ptr` is a pointer to a function that has no parameters and returns a value of the type `double`.

These pointers are all pointer variables. A *pointer variable* is a pointer that is a modifiable lvalue. Therefore, you can assign various values to it. In VOS C, pointers of all types are four bytes long, have the same format, and store an unsigned address.

Every pointer (except a pointer to `void`) has an associated data type, that of the function or object to which it points. Mixing pointer types is a common source of error. The type associated with a pointer and the type of an object whose address is assigned to the pointer must be the same. In the preceding examples, `int_ptr` has the type "pointer to `int`." Therefore, only the address of an object of the type `int` should be assigned to this pointer. See the "Pointer Operations" section that follows for more information on using pointers.

The explanation of pointers is divided as follows:

- pointer operations
- pointer arithmetic
- null pointer constants.

The sections that follow do not contain information on function pointers or on pointers to `void`. See the "Function Pointers" section in Chapter 6 for information on function pointers. See the "Void Type" section later in this chapter for information on pointers to `void`.

Pointer Operations

This section explains some typical pointer operations. See the “Pointer Arithmetic” section that follows for information on using pointers for address arithmetic.

The two most common operations performed with pointers are assigning an address with the address-of operator (`&`) and dereferencing a pointer with the indirection operator (`*`). *Dereferencing* a pointer means accessing the object to which the pointer points. An expression using the `*` operator yields an lvalue designating the object if its operand points to an object.

The general procedure for using the address-of and indirection operators is as follows:

1. Use an asterisk (called a pointer punctuator in this context) to declare the pointer.
2. Use the address-of operator to assign an object’s address to the pointer. The object should be the same type as the type associated with the pointer.
3. Use the indirection operator to access the contents of the pointed-to object.

The following example illustrates this three-step procedure.

```
int code, num = 25;

int *num_ptr;     /*  1. Declare the pointer using *
*/

num_ptr = &num;   /*  2. Assign an address to the pointer using &
*/

code = *num_ptr;   /*  3. Access contents of the pointed-to object
using *  */

printf("code = %d\n", code);       /*  code = 25  */
```

Because the result of a pointer operation is based on the type of the pointer, you must use an **explicit cast** in pointer operations, such as an indirection operation, where an object of one type is accessed through a pointer to an object of a different type. Otherwise, the pointer operation may yield unexpected and unwanted results. In the following example, a `short` variable is accessed through a pointer to an `int`.

```
1   short s_num, *s_ptr;
2   int i_num, *i_ptr;
3
4   main()
5   {
6       s_num = 100;
7
8       i_ptr = &s_num;
9
10     printf("s_num = %d\n", *(short *)i_ptr);    /*  Valid results
*/
11
12      printf("s_num = %d\n", *i_ptr);              /*  Invalid
results  */
13   }
```

In the preceding example, the pointer used to access `s_num` is declared as a "pointer to `int`." Notice that the results of the pointer dereference vary depending on whether `i_ptr` is explicitly cast to a `short *`. The output from the preceding program was as follows:

```
s_num = 100
s_num = 6553600
```

The indirection operation on line 10, `*(short *)i_ptr`, yields valid results because `i_ptr` is converted to a `short *` through the use of an explicit cast. In contrast, the indirection operation on line 12, `*i_ptr`, yields invalid results because `i_ptr` is not cast to a `short *`.

When you specify at least the `minimum` value for the `type_checking` option in a `#pragma` preprocessor control line or in the corresponding command-line argument, the compiler diagnoses implicit conversions of a pointer to an object of one type into a pointer to an object of a different data type.

In addition to the operations explained in the "Pointer Arithmetic" section, the other operations allowed with pointers include the following:

- comparing two pointers of compatible types (for example, pointers to members of the same array) with relational operators
- using the logical AND (`&&`) and logical OR (||) operators
- converting a pointer into an `int` type
- converting an expression of the type `int` into a pointer.

All pointer conversions must be specified using the appropriate casts. In the following example, the hexadecimal address must be cast to `short *` before it can be assigned to the pointer `shrt_ptr`.

```
shrt_ptr = (short *)0x00E0D048;
```

In VOS C, variables of the type `int` can store the address of an object or function.

Pointer Arithmetic

Pointer arithmetic refers to a subset of operations that can be performed with pointers. These operations consist of the following:

- adding an integer value to a pointer
- subtracting an integer value from a pointer
- subtracting one pointer from another pointer
- incrementing a pointer with the ++ operator
- decrementing a pointer with the – operator.

No other pointer arithmetic is allowed. You cannot, for instance, add two pointers together.

Note: Careless use of pointer arithmetic yields incorrect results. For example, you can erroneously reference a location beyond the limits of an array without generating a compile-time or run-time error message.

Adding and Subtracting Integer Values from Pointers

The first type of pointer arithmetic involves adding an integer value to a pointer or subtracting an integer value from a pointer. In these operations, the integer value can be an integer constant, an `int` variable, or an expression yielding an `int`. For example, if `ptr` is a pointer to an array member, the following pointer expressions represent addresses in memory after and before the object pointed to by `ptr`.

```
ptr + 3

ptr - 1
```

When an integer value is added to or subtracted from a pointer, the compiler calculates an address offset as follows:

```
integer_value * sizeof(pointed_to_object)
```

The compiler multiplies the *`integer_value`* by a scale factor. A *scale factor* is a value equal to the size of the pointed-to object. The compiler uses scale factors to take into account the size of the pointed-to object. As an example, for a pointer to a `short`, the compiler multiplies *`integer_value`* by the size of a `short` (two bytes) to calculate an address offset.

In the following example of a pointer expression, assume that `num_ptr` points to an `int`.

```
num_ptr + 3
```

This pointer expression yields an address offset that is 12 bytes (`3 * sizeof(int)`) after the address pointed to by `num_ptr`. The assumption behind this type of pointer arithmetic is that the pointer contains the address of an array element. Thus, calculating an address offset using the size of the pointed-to object moves the address from element to element in the array.

> **Note:** When you add an integer value to a pointer or subtract an integer value from a pointer, you could unintentionally access a storage location beyond the limits of the array. In this case, the operation yields an unexpected and unwanted result.

The following program illustrates how pointer arithmetic uses a scale factor to determine an address offset within an array.

```
short s_array[10], *s_ptr = &s_array[0];
int i_array[10], *i_ptr = &i_array[0];

main()
{
   printf("s_ptr contains the address of s_array[0]:  %u\n", s_ptr);
   printf("s_ptr + 3 is the address of s_array[3]:  %u\n\n", s_ptr +
3);

   printf("i_ptr contains the address of i_array[0]:  %u\n", i_ptr);
   printf("i_ptr + 3 is the address of i_array[3]:  %u\n", i_ptr + 3);
}
```

The output from the preceding program was as follows:

```
s_ptr contains the address of s_array[0]:  14737480
s_ptr + 3 is the address of s_array[3]:    14737486

i_ptr contains the address of i_array[0]:  14737504
i_ptr + 3 is the address of i_array[3]:    14737516
```

To calculate the correct address offsets, the VOS C compiler uses a scale factor of 2 for an array of `short` variables, but it uses a scale factor of 4 for an array of `int` variables. Therefore, the address specified by `s_ptr + 3` is 6 bytes after the address of the beginning of `s_array`. Similarly, the address specified by `i_ptr + 3` is 12 bytes after the address of the beginning of `i_array`.

The “Array Operations” section later in this chapter contains more information on arrays and address arithmetic.

Subtracting One Pointer from Another Pointer

The second type of pointer arithmetic involves subtracting one pointer from another pointer. Both pointers must point to objects that are members of the same array. After the compiler subtracts the addresses of two pointers, it divides the result (in bytes) by the size of the pointed-to object. Therefore, subtracting one pointer from another yields the number of elements between the two pointers.

The following program illustrates how pointer subtraction yields the number of elements between two pointers.

```
short s_array[10], *s_ptr1 = &s_array[0], *s_ptr2 = &s_array[3];

main()
{
   printf("The address in s_ptr1 is %u\n", s_ptr1);
   printf("The address in s_ptr2 is %u\n", s_ptr2);

   printf("\nSubtracting s_ptr2 from s_ptr1 yields %d\n", s_ptr1 -
s_ptr2);
   printf("Subtracting s_ptr1 from s_ptr2 yields %d\n", s_ptr2 -
s_ptr1);
}
```

The output from the preceding program was as follows:

```
The address in s_ptr1 is 14737480
The address in s_ptr2 is 14737486

Subtracting s_ptr2 from s_ptr1 yields -3
Subtracting s_ptr1 from s_ptr2 yields 3
```

Notice that, when pointer `s_ptr2` is subtracted from pointer `s_ptr1`, the pointer subtraction yields a **negative** value because `s_ptr2` locates an object with a higher address than the object pointed to by `s_ptr1`.

When one pointer is subtracted from another pointer, the number of elements between the two pointers always includes the element pointed to by one of the pointers. As an example, for `s_ptr2 - s_ptr1`, the difference is 3. Figure 4-4 shows this calculation.

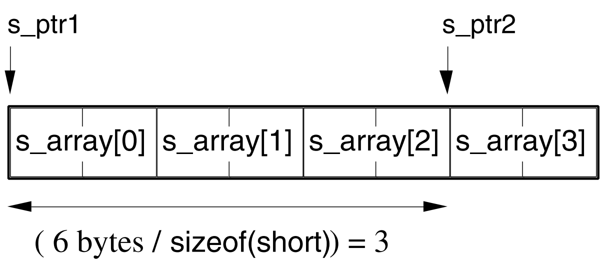


Figure 4-4. Pointer Subtraction

Subtracting `s_ptr1` from `s_ptr2` yields the number of elements between the two pointers: 3 elements. The compiler subtracts the addresses of the two pointers and divides the result (6 bytes) by the size of the pointed-to object. In this example, the compiler divides by 2 bytes (`sizeof(short)`).

Incrementing and Decrementing Pointers

The third type of pointer arithmetic involves incrementing a pointer with the `++` operator and decrementing a pointer with the `--` operator. The `++` operation adds the integer value 1 to a pointer. The `--` operation subtracts the integer value 1 from a pointer. For example, if `ptr` is a pointer to an array object, the following pointer expressions represent addresses in memory of the objects immediately after and before the object pointed to by `ptr`.

```
++ptr

--ptr
```

To calculate the address offset specified by these expressions, the compiler multiplies the value 1 by the following scale factor:

```
1 * sizeof(pointed_to_object)
```

With the `++` operator, the address offset is added to the address stored in the pointer. With the `--` operator, the address offset is subtracted from the address stored in the pointer.

The expressions `++ptr` and `--ptr` use the preincrement and predecrement operators. Therefore, each pointer is incremented or decremented **before** the expression in which it occurs is evaluated. The expressions `ptr++` and `ptr--` use the postincrement and postdecrement operators. Therefore, each pointer is incremented or decremented **after** the expression in which it occurs is evaluated.

When the `++` or `--` operator is used in a pointer expression that also contains the indirection operator, the rules of operator precedence and associativity determine whether the pointer or the pointed-to object is incremented or decremented. Using parentheses in complex pointer expressions helps to clarify the order of evaluation.

The following program illustrates how the compiler evaluates two pointer expressions: `*(++i_ptr)` and `++(*i_ptr).`

```
int i_array[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
int i_ptr = &i_array[5];

main()
{
   printf("The contents of i_array[5] is %d\n", *i_ptr);

   printf("\nThe expression *(++i_ptr) yields %d\n", *(++i_ptr) );
   printf("The expression ++(*i_ptr) yields %d\n", ++(*i_ptr) );
}
```

The output from the preceding program is as follows:

```
The contents of i_array[5] is 60

The expression *(++i_ptr) yields 70
The expression ++(*i_ptr) yields 71
```

In the expression `*(++i_ptr)`, first the pointer is incremented so that `i_ptr` points to `i_array[6]`. Then, the indirection operation yields the value stored in the pointed-to object: 70.

In the expression `++(*i_ptr)`, first the indirection operation is applied to `i_ptr`, yielding the value 70. Then, that value is incremented by 1. Thus, the second pointer expression evaluates to the value 71.

Null Pointer Constants

The null pointer constant appears with great frequency in pointer operations. A *null pointer constant* is an integer constant expression equal to the value 0, or such an expression cast to the type `void *`. Setting a pointer to `NULL` indicates that the pointer does not currently locate any object or function. The `NULL` pointer constant is defined in the ANSI C Standard and is commonly used in many C implementations.

In VOS C, the `OS_NULL_PTR` constant is an integer constant expression equal to the value 1, or such an expression cast to the type `void *`. Setting a pointer to `OS_NULL_PTR` indicates that the pointer does not currently locate any object or function. The `OS_NULL_PTR` constant is used in VOS PL/I for the value of the `null` built-in function. It is also used in VOS Pascal for the value of `nil`. The VOS operating system subroutines return the `OS_NULL_PTR` constant **not** the `NULL` pointer constant.

> **Note:** Be aware that `NULL` is **not equal to** `OS_NULL_PTR`. Also, functions that specify `OS_NULL_PTR` are not portable. You should, therefore, use the `NULL` pointer constant except in those situations where the operating system requires `OS_NULL_PTR`.

Both `NULL` and `OS_NULL_PTR` can be assigned to a pointer or compared for equality with a pointer. Both constants are defined in the `stddef.h` header file.

Using Null Pointer Constants

If a null pointer constant is compared with or assigned to a pointer, the compiler converts the null pointer constant to the type of that pointer. Two `NULL` pointer constants or two `OS_NULL_PTR` pointer constants are always equal regardless of how they are cast.

A pointer that is equal to `NULL` or `OS_NULL_PTR` is guaranteed **not** to be equal to a pointer to any object or function. Therefore, setting a pointer equal to `NULL` is often used to indicate a pointer that currently holds no valid storage location. For example, the following assignment sets the pointer `queue_head` equal to `NULL`.

```
queue_head = NULL;
```

After the preceding assignment, you could compare the `queue_head` pointer with `NULL` to find out whether the pointer has been assigned the address of a valid storage location.

Attempting to Access Data through a Null Pointer

A pointer that has static storage duration and that has not been initialized with an address is equal to `NULL`. A pointer that is equal to `NULL` is sometimes called a *null pointer*. A null pointer does not contain the address of a valid storage location. Using the indirection operator with a null pointer often results in a run-time error:

```
Attempt to reference a page that is not in your virtual space.
Referencing address 00000000x.
```

Using the indirection operator with a pointer that is equal to `OS_NULL_PTR` also often results in a run-time error. You must assign the address of a valid storage location to a pointer before dereferencing the pointer.

Array Types

An *array* is a set of objects of the same data type. In an array, the individual objects, or *elements*, are stored contiguously at increasing addresses in memory. Array types are characterized by their element type and by the number of elements in the array.

You declare an array by including, after the identifier, a set of brackets (`[]`) that typically contains an integer constant expression. The integer constant expression specifies the number of elements in the array. The following examples are declarations of arrays.

```
char name[20];

long int totals[100];
```

The preceding examples declare the following:

- `name` is an array containing 20 elements of the type `char`.
- `totals` is an array containing 100 elements of the type `long int`.

Arrays in C are zero-based. The subscript for the last element in a single-dimensional array is the size specified in the declaration minus 1. In the `totals` array, the first element is `totals[0]`, and the last element is `totals[99]`.

The elements of an array can be of any data type except a function type. Elements of an array can be pointers to functions. In addition, you can define arrays to have elements of a data type specified in a type definition. For example:

```
typedef char_varying(80) LINE;

LINE input_lines[99];
```

In the preceding example, `input_lines` is an array containing 99 elements of the type `char_varying(80)`.

Multidimensional Arrays

Multidimensional arrays have two or more bracketed integer constant expressions as part of the declarator. For example:

```
int matrix[3][5];
```

Each bracketed integer constant expression specifies the extent of one dimension of the array. Therefore, `matrix` is a two-dimensional array of `int`, consisting of three rows, each row containing five elements of the type `int`.

Multidimensional arrays in C are stored in row-major order. The term *row-major order* means that the first subscript after the array name specifies a row and the second subscript specifies a column:

```
multidimensional_array[row] [column]
```

In row-major order, if elements are accessed in the order in which they are stored, the rightmost subscript varies most frequently, and the leftmost subscript varies least frequently. Table 4-4 illustrates the row-major order of the two-dimensional array `matrix[3][5]`. Notice that the `matrix` array has 15 elements, arranged in 3 rows, each row having 5 columns.

Table 4-4. Two-Dimensional Array `matrix[3][5]`

	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`
`matrix[0]`	1st	2nd	3rd	4th	5th
`matrix[1]`	6th	7th	8th	9th	10th
`matrix[2]`	11th	12th	13th	14th	15th

To locate an individual element in a two-dimensional array stored in row-major order, use the following calculation:

```
1 + (NUMBER OF ELEMENTS IN A ROW * ROW SUBSCRIPT) + COLUMN SUBSCRIPT
```

Therefore, the multidimensional expression `matrix[2][3]` evaluates to the value of the 14th element of the array, calculated as follows:

```
1 + (5 * 2) + 3 = 14
```

Array Operations

This section explains the operations that you can perform with arrays. The information on array operations is divided as follows:

- accessing array elements
- using array names
- using pointer arithmetic with multidimensional arrays.

Accessing Array Elements

Normally, you access an array element with a bracketed constant expression, `subscript`, that yields an integer value specifying the element. The syntax for this type of array reference is as follows:

```
array_id [subscript]
```

The array-subscript operator (`[]`) is simply an alternate method of expressing the indirection operation. The preceding expression is identical in all respects to the following:

```
*(array_id + subscript)
```

The bracket and indirection forms are interchangeable whether the operand is an array, a character-string literal, or a pointer variable. For information about the indirection operator, see the "Indirection Operator" section in Chapter 7.

Using Array Names

An array name is a pointer representing the address of the first element of the array. Therefore, `array_id` is identical to the address of `array_id [0]`. An array name is a **pointer constant**, not a pointer variable. A pointer constant is the address in memory of an object (in this case, the address of the first element of an array). In contrast to a pointer variable, a pointer constant is **not** a modifiable lvalue. Thus, the following assignment is illegal:

```
array_id = NULL;
```

The compiler issues an error message in response to an attempt to reference an array name as if it were a pointer variable. However, an array name can be used in any other context where a pointer is allowed. For example, consider the following array.

```
char customer_name[30];
```

The array name `customer_name` can be thought of as a pointer to `char`. A pointer variable can be defined that points to a `char` object, and the address of the array can be assigned to that variable. For example:

```
char *ptr_to_name;

ptr_to_name = customer_name;
```

In this case, both `ptr_to_name` and `customer_name` point to the first element in the `customer_name` array. Notice that the array name, `customer_name`, is a pointer constant, not a modifiable lvalue. In contrast, `ptr_to_name` is a pointer variable containing the address of the array's first element.

In addition to being the right operand in an assignment expression, some of the other operations allowed with array names are explained in the following paragraphs.

When the address-of operator (`&`) has an array name as its operand, the operation yields different results depending on whether or not the array name was declared as a parameter.

- When an array name that is not a parameter is the operand of the `&` operator, the operation yields the address of the first element of specified array.
- When an array name that is a parameter is the operand of the `&` operator, the operation yields the address of a pointer object, which points to the first element of the array.

During compilation, a parameter of the type "array of *`type`*" is always adjusted into an object of the type "pointer to *`type`*." Therefore, when an array name that is a parameter is the operand of the `&` operator, the resulting address is of a pointer object, having the parameter storage class.

The `sizeof` operation can be performed on an array name. For an array that is not a parameter, the `sizeof` operation yields the number of bytes in the array. For an array that is a parameter, the `sizeof` operation yields the size of a pointer (four bytes) because an array name is converted to a pointer when used in this context.

VOS C also allows you to convert an array to a varying-length character string for subsequent use in `char_varying` string operations. You can use the `$substr` built-in function to convert an array to a `char_varying` string. See the "`$substr`" section in Chapter 12 for information about the `$substr` built-in.

Using Pointer Arithmetic with Multidimensional Arrays

Subscripting for multidimensional arrays involves similar conversions of array names to pointers. Multidimensional arrays are arrays of arrays. In most contexts, an array-name expression of type "*`dimension_1`* by *`dimension_2`* by … by *`dimension_n`* array of *`type`*" is converted to type "pointer to *`dimension_2`* by … by *`dimension_n`* array of *`type`*."

Figure 4-5 represents the storage allocated for a 3-by-4 array of int. The figure illustrates how an array name acts as a pointer. It also shows the pointer arithmetic involved in some typical references to the following multidimensional array:

```
int multi[3][4] =
   {
   {10, 11, 12, 13},
   {20, 21, 22, 23},
   {30, 31, 32, 33}
   };
```

```
multi == &multi[0]                                 ======== Increasing
**multi == *multi[0] == multi[0][0]                   10     Addresses
                                                   --------
                                                      11        !
                                                   --------     !
                                                      12        v
                                                   --------
                                                      13
multi + 1 == &multi[1]                             ========
**(multi + 1) == *multi[1] == multi[1][0]             20
                                                   --------
                                                      21
                                                   --------
                                                      22
                                                   --------
                                                      23
                                                   ========
multi + 2 == &multi[2]                                30
                                                   --------
                                                      31
                                                   --------
                                                      32
                                                   --------
*(multi[2] + 3) == multi[2][3]                        33
                                                   ========
```

Figure 4-5. Multidimensional Arrays and Pointer Arithmetic

Figure 4-5 shows that the two-dimensional array `multi[3][4]` is, in fact, three contiguously stored arrays, and that each array contains four `int` elements. The array name `multi` is the address of another array, `multi[0]`. The expression `multi[0]` is the address of the first element of the first array. The indirection operation `*multi[0]` is the equivalent of the expression `multi[0][0]` and yields the value 10. Both pointers `multi` and `multi[0]` are pointer constants.

In a similar way, the expression `(multi + 1)` is the address of another array, `multi[1]`. In the expression `(multi + 1)`, the scale factor that is incremented by the `+ 1` is an array of four objects of the type `int`. The expression `multi[1]` is the address of the first element of the second array. The indirection operation `*multi[1]` is the equivalent of `multi[1][0]` and yields the value 20.

The expression `multi[2]` is the address of the first element of the third array. The expression `(multi[2] + 3)` points to the third element of the third array. In the expression `(multi[2] + 3)`, the scale factor that is incremented by the `+ 3` is a single `int`. The indirection operation `*(multi[2] + 3)` is the equivalent of `multi[2][3]` and yields the value 33.

Structure Types

A *structure* is a sequentially allocated set of objects, grouped under a single name. Within the structure, each object or *member* can have its own name and a distinct type. A structure is similar to a record in some other high-level languages, such as Pascal. Figure 4-6 shows a sample structure declaration.

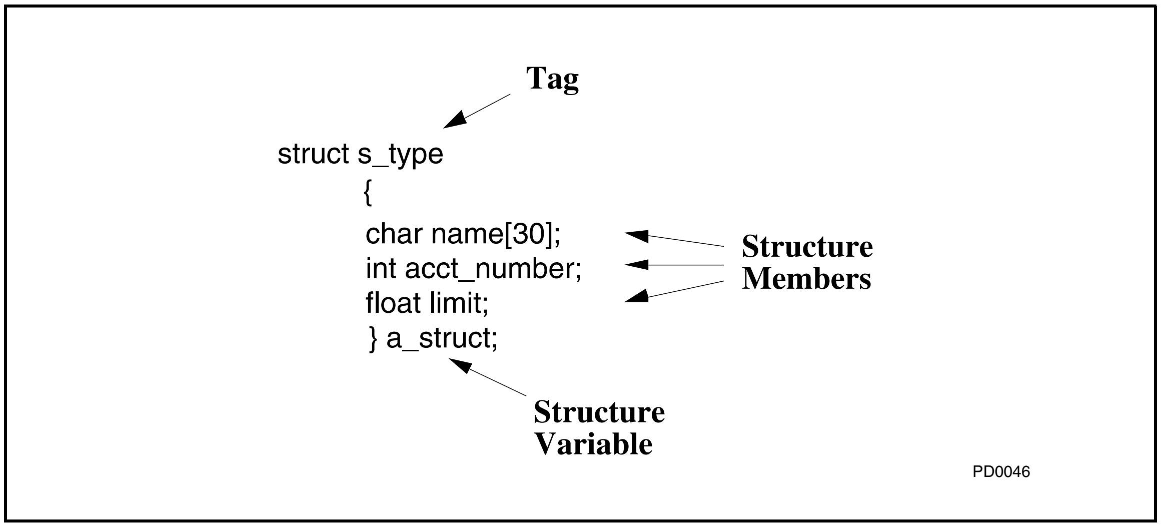


Figure 4-6. Sample Structure Declaration

The syntax for a `struct` declaration that specifies the contents of the structure object is as follows:

```
struct[tag] { member_declaration_list } [declarator_list];
```

Although they are not shown in the preceding syntax, the declaration of a structure can include type qualifiers and alignment specifiers. For information on how to use these elements when declaring a structure object, see the “Type Qualifiers” and “Alignment Specifiers” sections in Chapter 3.

The keyword `struct` introduces a structure declaration. Following the keyword `struct`, the *`tag`* is an optional name that identifies the particular type of structure. In Figure 4-6, `s_type` is the structure tag. You can use this tag preceded by the word `struct` to declare structure variables of this type. For example:

```
struct s_type a_struct2, a_struct3;
```

In the preceding example, `a_struct2` and `a_struct3` are structure variables of the type `s_type`.

The *`member_declaration_list`* is a list of object declarations for the structure members. In Figure 4-6, the structure type `s_type` contains three members: a `char` array, an `int`, and a `float`. In VOS C, the identifiers for each member are optional. However, a member with no identifier cannot be accessed. A structure member can be an object of any type except `void`. A structure member can be a union or another structure. A structure member cannot be a function but can be a pointer to a function.

In a structure declaration, you can include an optional *`declarator_list`*, containing one or more declarators naming variables of this particular structure type. Multiple declarators are separated by commas. In Figure 4-6, `a_struct` is a structure variable of the type `s_type`.

The tag can use the same name as a variable identifier because tags have their own name space distinct from ordinary identifiers. Also, the same member name can appear in more than one structure. Each unique structure has a separate name space for its members. However, when referencing the member, you must qualify the member name with the structure name if two structures have members with identical names. See the "Name Space" section in Chapter 3 for more information on name space.

As an extension, VOS C allows you to declare an anonymous structure. An *anonymous structure* is a structure that is nested within another structure or union and that does not have a name associated with it. The following example shows a structure containing an anonymous structure.

```
struct
   {
   int i;
   struct                /*  Nested anonymous structure  */
      {
      char c;
      short s;
      };
   float f;
   } ex_struct;
```

As shown in the preceding example, no identifier for the structure nested within `ex_struct` is required.

Tags for Structure Types

A structure declaration that is not followed by any declarators does not allocate storage for any structure variables but merely describes the form of a structure. In this case, you must include a tag in the declaration. You can use the tag to declare objects of the specified structure type. In the following example, the structure declaration is not followed by any declarators, but the tag `s1` is declared to identify a specific structure type.

```
struct s1
   {
   char c;
   int i;
   };
```

You can now use the tag to declare structures or pointers to structures of the type `struct s1`. For example:

```
struct s1 a_struct;

struct s1 *s1_ptr;
```

Using a structure tag in a declaration before you define the contents of the associated structure creates an *incomplete type*. You can use an incomplete tagged structure type only in contexts where the incomplete type's size is not required. For example, you can associate an identifier with the incomplete tagged structure type when a `typedef` name is declared to be a specifier for the structure, or when a pointer to or a function returning the structure is declared.

You complete an incomplete tagged structure type by specifying the contents of the structure in a definition in the **same** scope. The incomplete tagged structure type cannot be completed in an enclosed scope, which declares a new type known only within that block.

See the "Incomplete Types" section in Chapter 3 for more information on and an example of using an incomplete tagged structure type.

Self-Referential Structures

When you declare a self-referential structure, you must use a structure tag. A *self-referential structure* is one that contains a pointer to a structure of the same type as is being declared.

The following example is a self-referential structure that contains two pointers to structures of its own type.

```
struct link_list_item
   {
   int data;
   struct link_list_item *prev;
   struct link_list_item *next;
   };
```

The two structure components `prev` and `next` are pointers to structures of the type `struct link_list_item`.

Type Definitions for Structures

Type definitions can also be used with structures. You can use type definitions and structure tags in a similar way to define structure types. In the following example, the `typedef` name `LIST_ITEM` is used to define a structure and a pointer to a structure.

```
typedef struct link_list_item
   {
   int data;
   struct link_list_item *prev;
   struct link_list_item *next;
   } LIST_ITEM;

LIST_ITEM one_item, *head_ptr;
```

In the preceding declaration, `one_item` is a structure of the type `struct link_list_item`, and `head_ptr` is a pointer to a structure of the same type. Notice that the tag, `struct link_list_item`, is needed in the type definition because the structure is self-referential.

Operations on Structures

Structure members can be accessed in two ways: with the structure-member operator or the structure-pointer operator.

The first method for accessing a structure member is with the structure-member operator (`.`). You use the structure-member operator with a structure variable name (not a tag name) and a member name as specified in the structure definition. The syntax for this reference is as follows:

```
expression.member_name
```

The following program fragment illustrates how to use the structure-member operator to modify the `num` member of the `acct` structure.

```
int new_num;

struct acct_tag
   {
   char name[30];
   int num;
   };

struct acct_tag  acct;
   .
   .
   .
acct.num = new_num;
```

The second method for accessing a structure member is with the structure-pointer operator (`->`). After a pointer to a `struct` type is initialized with the structure's address, you use the structure-pointer operator with the pointer and a member name as specified in the structure definition. The syntax for this reference is as follows:

```
structure_pointer
```

The following example, a modified version of the preceding program fragment, illustrates how to use the structure-pointer operator to modify the `num` member of the pointed-to `acct` structure.

```
int new_num;

struct acct_tag
   {
   char name[30];
   int num;
   };

struct acct_tag  acct, *struct_ptr;
   .
   .
   .
struct_ptr = &acct;

struct_ptr->num = new_num;
```

See the "Structure/Union Member and Pointer Operators" section in Chapter 7 for more information on referencing structure members.

You can assign one structure to another structure **of compatible type**. In the following program, the value of `a_struct` is assigned to `another_struct`.

```
struct aaa
   {
   char a;
   int x;
   short y;
   char b;
   };

struct aaa a_struct = {'Z', 20, 10, 'B'};
struct aaa another_struct;
   .
   .
   .

another_struct = a_struct;
```

In VOS C, you can use the `$substr` built-in function to compare two structures. See the "`$substr`" section in Chapter 12 for information on this method of comparison.

A structure can be passed as an argument in a function invocation. In addition, a function can have a structure as its return type.

Memory Allocation for Structures

Structure members are allocated memory starting at the address of the structure itself. In general, members are stored at increasing addresses and in the same order in which they are declared.

Structure members are allocated with additional trailing bytes, or *padding*, whenever necessary to ensure that each structure member is aligned correctly. When you use the shortmap alignment rules with structures that contain anything other than `char` data, all members begin on an even-numbered byte. For structures that contain only `char` data, all members can begin on an odd- or even-numbered byte (that is, all members are byte-aligned).

Consider the following declaration. The two `char` structure member objects are each followed by a trailing byte of padding because the structure also contains non-`char` data.

```
struct
   {
   char a;
   int x;
   char b;
   short y;
   } s1;
```

With the shortmap alignment rules, this structure is allocated 10 bytes of storage. Figure 4-7 represents the memory allocated for structure `s1`. In the figure, each square represents one byte of memory. A square containing an identifier, such as `a`, indicates a byte used for that structure member. A square containing an asterisk (`*`) indicates a byte used for alignment padding.

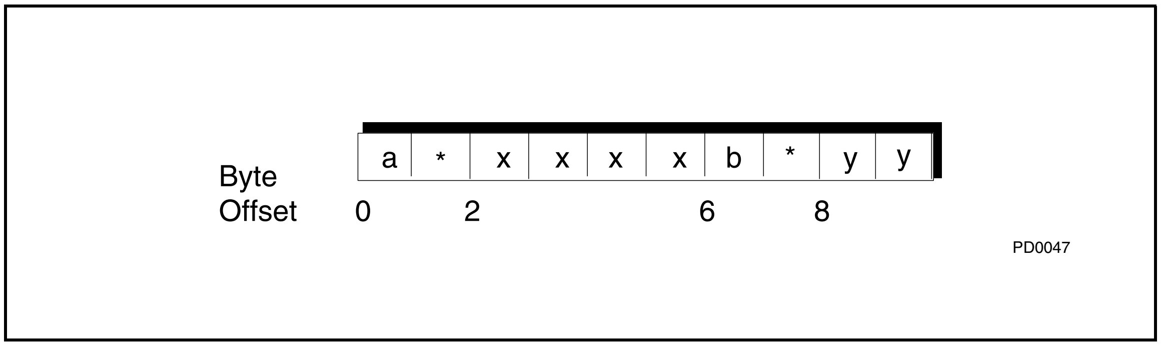


Figure 4-7. Structure Memory Allocation Example

Objects of the type `char` normally require one byte of storage. However, to ensure that `int x` and `short y` are aligned on an even-numbered byte, `char a` and `char b` are each followed by a trailing byte of padding. Reorganizing the structure so that the two `char` variables are stored next to each other would conserve two bytes of storage. In an array containing many structures, this savings can be considerable.

For more information on structure alignment, see the “Data Alignment” section in Chapter 5.

Bit-Field Data

A *bit field* is a structure or union member that consists of a cluster of contiguous bits. Within a structure or union declaration, you define a bit field with a colon and constant expression. For example:

```
struct s_template
   {
   unsigned x: 1;
   unsigned y: 20;
   unsigned:   0;
   unsigned z: 4;
   } s1;
```

The integer constant expression following the colon specifies how many bits to allocate for that field. If the default bit-field size is in effect, a bit field can contain 0 to 32 bits. A bit-field structure member declared to have a width of 0 indicates that no further bit fields are to be packed in the unit of reserved space in which the previous bit field was placed. See the next section, “Bit-Field Allocation,” for information on bit-field size.

You can declare bit fields to be signed or unsigned integers. However, in VOS C, regardless of the type specified, bit fields are always unsigned integers (that is, the high-order bit is not treated as a sign bit). You can declare an unnamed bit field for padding, but you cannot access the field. You cannot declare an array of bit fields.

A bit field can be used as a modifiable lvalue or as an operand in an arithmetic or bitwise operation. However, in VOS C, you cannot use a bit field as the operand for an increment (`++`) or decrement (`--`) operation. Also, you cannot use the address-of operator (`&`) with a bit field. Thus, you cannot access a bit field with a pointer. If a bit field is the initial member of a structure, a pointer to the structure, suitably converted, points to the storage unit that contains the bit field.

Bit-Field Allocation

In VOS C, three factors control how bit fields are allocated within a structure: bit-field size, bit-field alignment, and bit-packing direction. By default, the compiler uses a specific method to allocate bit fields. If you want to change how bit fields are allocated for a source module, three `#pragma` options allow you to control all aspects of bit-field allocation.

The three factors that control bit-field allocation are as follows:

- *Bit-field size* determines the amount of space that the compiler reserves when it needs to allocate storage for bit fields in a structure or union. By default, the compiler reserves four bytes (an `int`-size space).

- *Bit-field alignment* determines the type of boundary where the reserved space can begin. By default, the compiler allocates the reserved space on an even-numbered byte boundary.

- *Bit-packing direction* determines whether the compiler should start allocating bits from the left or right side of the reserved space. By default, the compiler begins allocating bits from the left side of the reserved space.

When the compiler encounters the first (or only) bit field within a structure, it uses the following procedure to reserve space for, align, and store bit fields within the structure.

1. The compiler reserves a unit of space determined by the current value of bit-field size.
2. As the compiler reserves each amount of space, it aligns the space according to the method of bit-field alignment currently in effect. Alignment of space is relative to the beginning of the structure.
3. The compiler allocates as many consecutive bit fields as will fit **completely** in the reserved space. The fields are allocated with the first field occupying either the leftmost or rightmost bits in the space, depending on the bit-packing direction currently in effect.
4. If an additional amount of space is needed, go to step 1.

When a non-bit-field member follows a bit field, the compiler allocates enough trailing bytes to ensure correct alignment. If subsequent bit fields appear after an intervening non-bit-field member, the compiler uses the same four-step procedure to allocate storage for these fields.

As an example, consider the following structure containing a `char` member and three bit fields.

```
struct
   {
   char f;
   unsigned a:  3;
   unsigned b:  4;
   unsigned c:  2;
   } s;
```

Figure 4-8 and the explanation that follows it assume that the compiler uses the default value for each factor that determines how bit fields are allocated in a structure. Figure 4-8 shows how the compiler allocates storage for the structure `s`. In the figure, each square represents one bit of memory. A square containing an identifier, such as `f`, indicates a bit used for that member or bit field. A square containing an asterisk (`*`) indicates a bit used for alignment padding. A square containing a hyphen (`-`) indicates a bit that is not used.

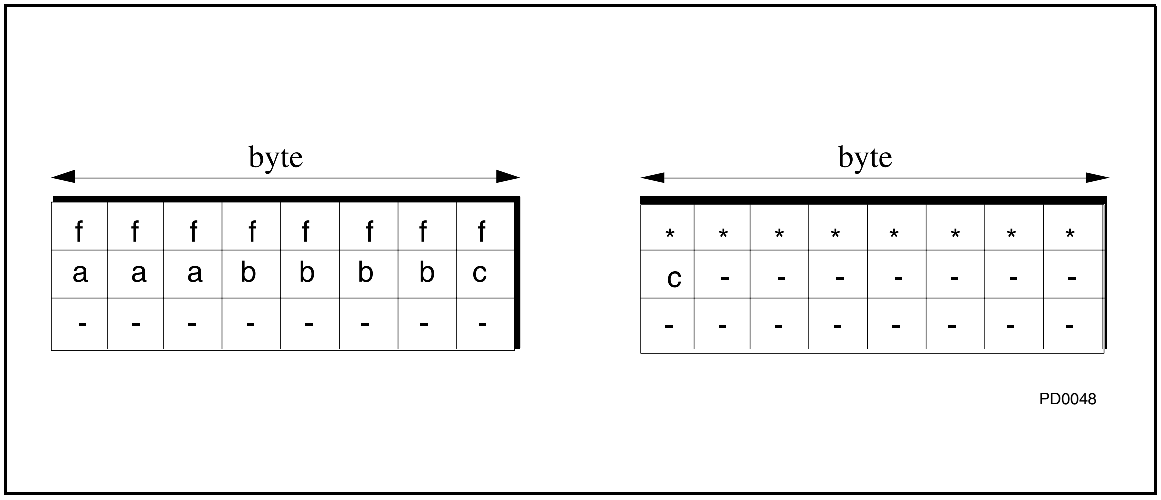


Figure 4-8. Default Bit-Field Allocation Example

The compiler allocates one byte (eight bits) for the `char` structure member, `f`, and one byte for alignment padding. With shortmap alignment, all non-`char` members are aligned on an even-numbered byte, and all `char` members are byte-aligned. Regardless of the size specified in bit-field size, bit-field structure members are always aligned according to the bit-field alignment method that is in effect when the source module is compiled.

When the compiler encounters the first bit field, `a`, it does the following:

- reserves four bytes of space (the default bit-field size).
- aligns the space on an even-numbered byte boundary (the default bit-field alignment method).
- allocates three consecutive bit fields because `a`, `b`, and `c` will fit completely in the four bytes. The fields are allocated with the first field occupying the leftmost bits in the space (the default bit-packing direction).

The remainder of the four bytes of reserved space is part of the structure but remains unfilled. Therefore, the `s` structure is allocated a total of six bytes.

Bit-Field Allocation Using Pragmas

In VOS C, values associated with bit-field size, bit-field alignment, and bit-packing direction control how bit fields are allocated within a structure. The previous section, “Bit-Field Allocation,” explains the method that the compiler uses to allocate bit fields.

Three `#pragma` options allow you to change the default values for these three factors and, therefore, to control all aspects of bit-field allocation. These `#pragma` options are as follows:

```
bit_field_size ( size )

bit_field_align ( type )

bit_packing ( direction )
```

With these options, you can change the default allocation method used by the compiler. You can specify these bit-field allocation values only once at the beginning of the source module. Table 4-5 describes these options and their allowed values. See "The `#pragma` Directive" section in Chapter 9 for more information on using `#pragma` options.

Table 4-5. Pragma Options for Bit-Field Allocation

Option	Purpose	Allowed Values

`bit_field_size`	The *`size`* value determines the amount of space that the compiler reserves when it needs to allocate storage for bit fields. The default value for *`size`* is `int`.	`char`
		`short`
		`int`

`bit_field_align`	The *`type`* value determines the type of boundary where space for bit fields can begin. The default value for *`type`* is `short` (aligned on an even-numbered byte boundary).	`char`
		`short`
		`int`

`bit_packing`	The *`direction`* value determines whether the compiler begins allocating bits from the left or right side of the reserved space. The default value for *`direction`* is `left_to_right`.	`left_to_right` `right_to_left`

The compiler generates an error message if you try to specify an individual bit field that is larger than the size specified in `bit_field_size`.

The following examples illustrate how to use the three `#pragma` options to allocate bit fields.

Example 1

Consider the following program fragment containing a structure with a `char` member and three bit fields.

```
#pragma bit_field_size (char)

struct
   {
   char f;
   unsigned a:    3;
   unsigned b:    4;
   unsigned c:    2;
   } s1;
```

Notice that the `bit_field_size` option changes the bit-field size to `char` or one byte. For bit-field alignment and bit-packing direction, the compiler uses the default values, respectively `short` and `left_to_right`, to allocate bit fields in the structure.

Figure 4-9 shows how the compiler allocates storage for the structure `s1`. In the figure, each square represents one bit of memory. A square containing an identifier, such as `f`, indicates a bit used for that member or bit field. A square containing an asterisk (`*`) indicates a bit used for alignment padding. A square containing a hyphen (`-`) indicates a bit that is not used.

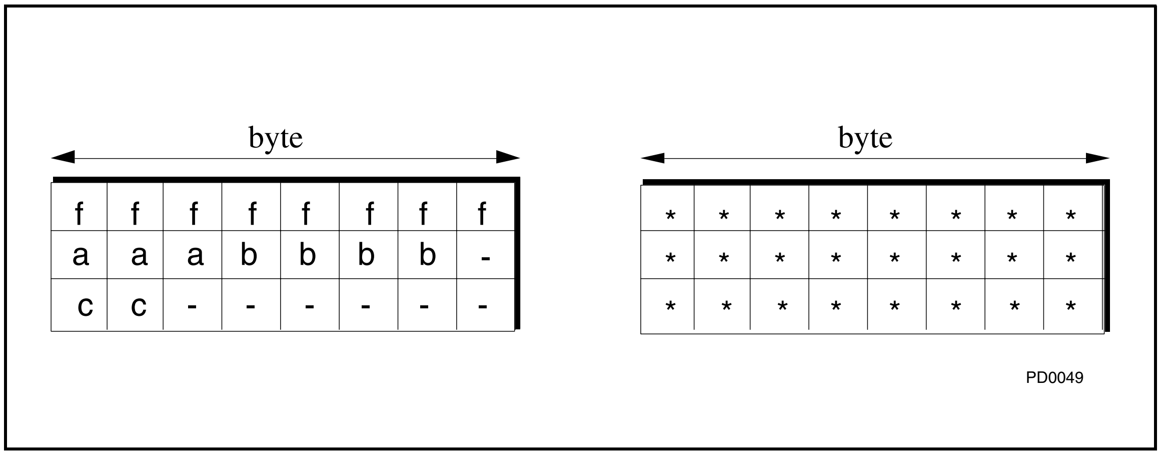


Figure 4-9. Bit-Field Allocation Example 1

The compiler allocates one byte for the `char` structure member, `f`, and one byte for alignment padding. When the compiler encounters the first bit field, `a`, it does the following:

- reserves one byte of space (`bit_field_size` is `char`).
- aligns the space on an even-numbered byte boundary (the default bit-field alignment method).
- allocates two consecutive bit fields because `a` and `b` will fit completely in one byte. The fields are allocated with the first field occupying the leftmost bits in the space (the default bit-packing direction).

- allocates one byte for alignment padding
- reserves one byte of space.
- aligns the space on an even-numbered byte boundary.
- allocates the last bit field, `c`. The bit field is allocated so that it occupies the leftmost bits in the space.

The six unused bits of reserved space are part of the structure but remain unfilled. Finally, the compiler allocates one byte for alignment padding. Therefore, the `s1` structure is allocated a total of six bytes.

Example 2

Consider the following program fragment containing a structure with a `char` member and three bit fields. In this example, the program changes each of the default values for bit-field allocation.

```
#pragma bit_field_size (char)
#pragma bit_field_align (char)
#pragma bit_packing (right_to_left)

struct
   {
   char f;
   unsigned a:    3;
   unsigned b:    4;
   unsigned c:    2;
   } s2;
```

Notice that the `bit_field_size` option changes the size to `char`. The `bit_field_align` option changes the alignment boundary to `char`. Also, the `bit_packing` option changes the direction to `right_to_left`.

Figure 4-10 shows how the compiler allocates storage for the structure `s2`. In the figure, each square represents one bit of memory. A square containing an identifier, such as `f`, indicates a bit used for that member or bit field. A square containing an asterisk (`*`) indicates a bit used for alignment padding. A square containing a hyphen (`-`) indicates a bit that is not used.

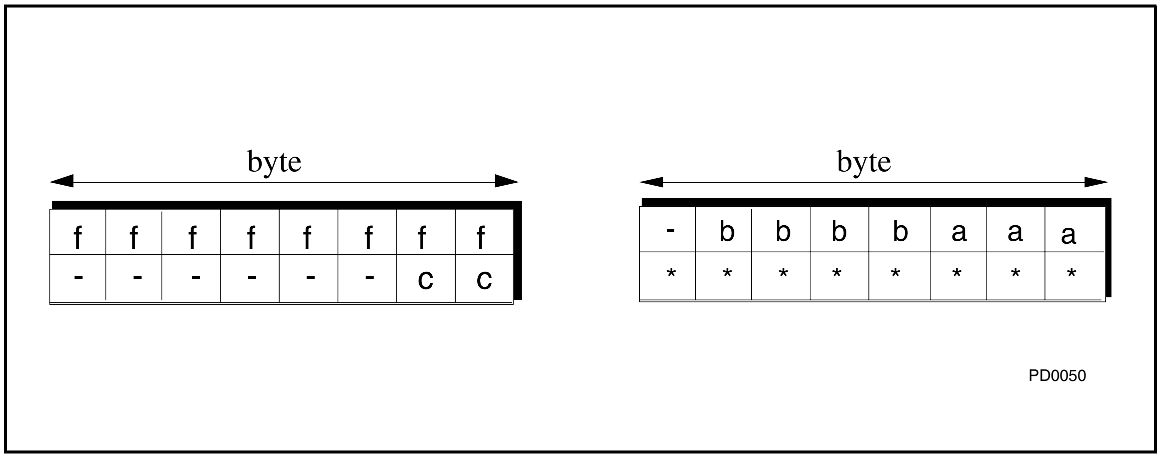


Figure 4-10. Bit-Field Allocation Example 2

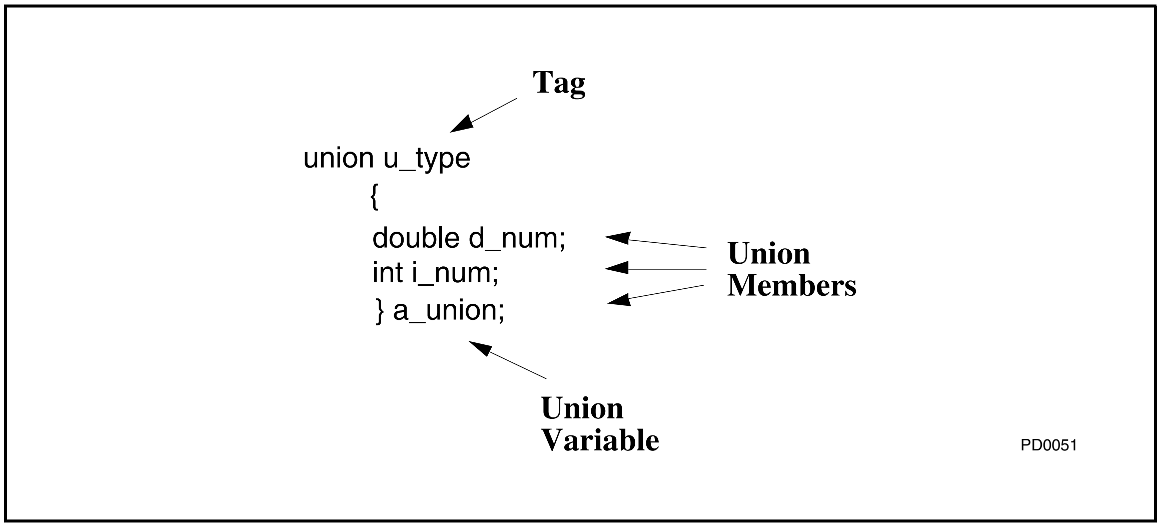
The compiler allocates one byte for the `char` structure member, `f`. Because the structure contains only `char` data, no alignment padding is allocated. When the compiler encounters the first bit field, `a`, it does the following:

- reserves one byte of space (`bit_field_size` is `char`).
- aligns the space on a byte boundary (`bit_field_align` is `char`).
- allocates two consecutive bit fields because `a` and `b` will fit completely in one byte. The fields are allocated from right to left with the first field occupying the rightmost bits in the space (`bit_packing` is `right_to_left`).
- reserves one byte of space.
- aligns the space on a byte boundary.
- allocates the last bit field, `c`. The bit field is allocated so that it occupies the rightmost bits in the space.

The six unused bits of reserved space are part of the structure but remain unfilled. Finally, the compiler allocates one byte for alignment padding. Even if `bit_field_size` is `char`, the structure containing the bit fields is aligned (and must begin and end) on an even-numbered byte boundary. Therefore, the `s2` structure is allocated a total of four bytes.

Union Types

A *union* is an overlapping set of member objects, grouped under a single name. Each object in a union can have its own name and a distinct type. A union is similar to a structure. However, a union can hold **only one** member object at a time. Figure 4-11 shows a sample union declaration.



```
union u_type
    {
    double d_num;
    int i_num;
    } a_union;
```

Figure 4-11. Sample Union Declaration

The syntax for a `union` declaration that specifies the contents of the union object is as follows:

```
union [tag] { member_declaration_list } [declarator_list];
```

Although they are not shown in the preceding syntax, the declaration of a union can include type qualifiers and alignment specifiers. For information on how to use these elements when declaring a union object, see the “Type Qualifiers” and “Alignment Specifiers” sections in Chapter 3.

The keyword `union` introduces a union declaration. Following the keyword `union`, the *`tag`* is an optional name that identifies this particular type of union. In Figure 4-11, `u_type` is the union tag. Similar to a structure tag, you can use this tag preceded by the word `union` to declare unions of this type.

The `member_declaration_list` is a list of object declarations for the union members. In Figure 4-11, the union type `u_type` contains two members: a `double` and an `int`. A union member can be an object of any type, including a structure or another union. A union member cannot be a function type but can be a pointer to a function.

You can include an optional *`declarator_list`*, containing one or more identifiers naming variables of this particular union type. In Figure 4-11, `a_union` is a union variable of the type `u_type`.

A union tag can use the same name as a variable identifier because tags have their own name space distinct from ordinary identifiers. Also, each unique union has a separate name space for its member names. However, when referencing the member, you must qualify the member name with the union name if two unions have members with identical names.

As an extension, VOS C allows you to declare an anonymous union. An *anonymous union* is a union that is nested within another structure or union and that does not have a name associated with it. See the “Structure Types” section earlier in this chapter for an example of an anonymous data item.

Unions and structures are identical in many respects. However, unions are different from structures in the following ways.

- A union can hold **only one** member object at a time. Therefore, the manner in which unions are used is somewhat different from the manner in which structures are used.
- Memory allocation for unions is different from memory allocation for structures. All union members begin at offset 0 from the address of the union.

The sections that follow, “Operations on Unions” and “Memory Allocation for Unions,” focus on the preceding differences between unions and structures.

In most other respects, unions and structures are **identical**. The following table lists sections in this chapter where you can find information on topics that relate to unions.

For This Information on Unions	**See This Section in Chapter 4**
Using tags for union types	“Tags for Structure Types”
Using type definitions with unions	“Type Definitions for Structures”
Creating and using bit fields in a union	“Bit-Field Data”

Operations on Unions

Unions can hold only one member at a time. Usually, unions are used when it is necessary to store different types of data in the same storage location. It is the programmer’s responsibility to keep track of what data type the union currently holds. For example, you should not attempt to read the contents of one union member when the preceding assignment to the union wrote data to another member.

In the following example, the program assigns a value to the `double` member of `union_1`. The second `printf` function attempts to display the value stored in the `int` member of the union. The function generates invalid results because the `int` member was not assigned a value in the earlier assignment statement.

```
union
   {
   int i;
   double d;
   } union_1;

main()
{
   union_1.d = 2.222;

  printf("The double is %lf\n", union_1.d);   /*  Valid results    */

  printf("The integer is %d\n", union_1.i);   /*  Invalid results  */
}
```

As with structures, certain operations are allowed on unions. The operations described in the "Operations on Structures" section earlier in this chapter are performed in an identical manner with unions. Therefore, see the explanation in that section for detailed information on the following:

- accessing a union member with the union-member operator (`.`) and the union-pointer operator (`->`)
- assigning the value of one union (based on its largest member) to another union **of compatible type**.

In VOS C, you can use the `$substr` built-in function to compare two unions. See the "`$substr`" section in Chapter 12 for information on this method of comparison.

You can pass a union as an argument in a function invocation. In addition, a function can have a union as its return type.

Memory Allocation for Unions

Union members are allocated memory starting at the address of the union itself. In contrast to structure members, all union members are stored starting at offset 0 from the address of the union. With a union, different data types are stored in overlapping memory locations.

Consider the following union declaration. The two union members, `d` and `i`, are stored starting at the same address.

```
union
   {
   double d;
   int i;
   } ex_union;
```

This union is allocated eight bytes of storage. The length of a union is the number of bytes occupied by its largest member. In this example, the largest member `d` occupies eight bytes.

Figure 4-12 represents the memory allocated for `ex_union`. The union can never store both members at the same time because overlapping locations are used to hold the union's two members. In the figure, each square represents one byte of memory. A square containing an identifier, such as `d`, indicates a byte used for that union member. A square containing a slant (`/`) indicates a byte that is not used and holds a meaningless value.

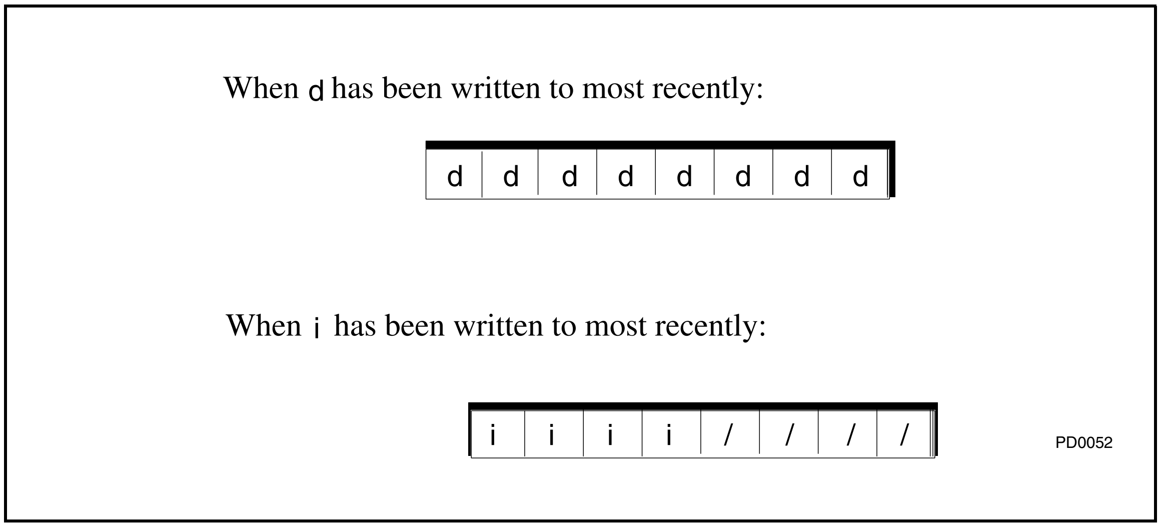

When d has been written to most recently:
d d d d d d d d
When i has been written to most recently:
i i i i / / / /
PD0052

Figure 4-12. Union Memory Allocation Example

In Figure 4-12, when the `i` union member has been written to most recently, the last four bytes of storage are not used and hold meaningless values.

Void Type

The `void` type comprises an empty set of values. The `void` type specifier can be used to specify:

- a void expression
- a `void` function return type
- a pointer to `void`.

Though not related to the `void` type, the `void` keyword can also be used to indicate the absence of function parameters in a function declaration or definition. See the “Function Definitions with Prototypes” section in Chapter 6 for information on this use of the `void` keyword.

Void Expressions

The `void` type specifier can be used in a cast to tell the compiler to discard an expression’s value. For example, the following function invocation is cast to `void`, indicating that the return value is explicitly discarded.

```
int func(int);
int i;
    .
    .
    .
(void)func(i);  /*  Cast to void:  discard any return value  */
```

An expression cast to `void` does not yield a value, and such an expression cannot be assigned to any other expression. However, a void expression is evaluated for its possible side effects.

Explicit or implicit conversions cannot be applied to a void expression. An expression cast to `void` can be used only in contexts where no value is required. For example, a void expression can be used as an expression statement or as the left operand of a comma operator.

If you use an expression of another type in a context where a void expression is required, the expression's value is discarded.

Functions Returning Void

The `void` type specifier can be used to declare or define a function that does not return a value. For example:

```
void func(double d);       /*  Function declaration  */
   .
   .
   .
void func(double d)        /*  Function definition  */
{
   .
   .
   .
}
```

When a function's return type is declared `void`, it is an error to use the function in an expression as if it returned a value. A function that does not return a value exists only to produce side effects (that is, to perform some action).

If one source module defines a function as returning `void`, and another source module declares the same function as returning some data type other than `void`, the binder issues the following warning:

```
bind: Warning: Module calling_function_module_name does not declare
     called_function to be a subroutine.
```

The binder categorizes a function that does not return a value as a subroutine. In other VOS languages such as PL/I, a *subroutine* is a routine that does not return a value.

If one source module defines a function as returning a data type other than `void`, and another source module declares the same function as returning `void`, the binder issues the following warning:

```
bind: Warning: Module calling_function_module_name does not declare
     called_function to be a function.
```

Pointers to Void

A pointer to `void` is a generic pointer that can be used to point to a variety of data types. In VOS C, a pointer to `void` can be converted to or from a pointer to any object type, function type, or incomplete type. In addition, when a pointer to any object, function, or incomplete type is converted into a pointer to `void`, and back again, the result will compare equal to the original pointer. However, a pointer to `void` cannot be dereferenced without an explicit cast. You declare a pointer to `void` as follows:

```
void *ptr;
```

The compiler does not generate an error message when a pointer to `void` is, without a cast, assigned to or compared with a pointer to another data type. For example, the definitions of some library functions specify `void *` as a return type. Consider the following declaration of the `malloc` library function.

```
void *malloc(size_t size);
```

The declaration of `malloc` specifies that the function returns a pointer to `void`. When `malloc` is called, the value returned by the function can be assigned, without an explicit cast, to a pointer to the appropriate data type. For example:

```
char string[30], *string_ptr;

string_ptr = malloc(sizeof(string));
```

Declaring the return type of `malloc` as a pointer to `void` ensures that the function can return a pointer to any valid data type. Though a cast is not required in such an assignment, many programmers prefer to cast `malloc`'s return value so that the conversion from `void *` to `char *` is explicit.

The `void` data type, by definition, is an empty set of values. Therefore, a pointer to `void` is not typed (scaled) for a data type. For this reason, you cannot use the indirection operator (`*`) with a pointer to `void` without including a suitable cast. The compiler generates an error message if you attempt to dereference a pointer to `void` without a cast.

When you cast a pointer to `void` into a pointer to an object of a valid data type, you can then use the indirection operator with the pointer. In the following example, the program dereferences a pointer to `void` without and then with an explicit cast.

```
void *ptr;

main()
{
   short num1, num2 = 10;

   ptr = &num2;

   num1 = *ptr;           /*  Invalid:  ptr is dereferenced without
a cast  */

  num1 = *(short *)ptr;  /*  Valid:  ptr is dereferenced with a cast
*/
}
```

Notice that, when a pointer to `void` is properly cast, you can use the indirection operator to access the pointed-to object. In the last assignment in the preceding example, `ptr` is cast to `short *`. Therefore, the expression `*(short *)ptr` correctly yields the contents of the pointed-to object, `num2`.

Varying-Length Character String Type

A *varying-length character string* is a sequence of characters that can be of different lengths during the program's execution. Varying-length character string data is a VOS C extension. Many VOS operating system subroutines require this data type. Varying-length character string type`char_varying` type

The `char_varying(`*n*`)` data type corresponds exactly to the VOS PL/I data type `char(`*n*`)` varying. When a `char_varying(`*n*`)` string has external linkage, the string can be referenced in VOS PL/I programs in which it is declared `char(`*n*`) varying`.

Additional information on using `char_varying` data can be found in the following sections of this book.

- See the "Type Conversions" section in Chapter 5 for information on varying-length character string conversions.

- See the "String Manipulation" section in Chapter 11 for information on generic library functions that manipulate varying-length character string data.

Varying-length character string data is **very different** from a C string. The length of a C string is the number of characters preceding the first null character (`'\0'`) that appears in the string. In contrast, the current length of a varying-length character string is determined by a 2-byte length that precedes the string. With a C string, the name of the array containing the string is a pointer to the string's initial character. In contrast, a varying-length character string's name is like the name of a structure. The name of a structure identifies an aggregate object: a structure. The name of a `char_varying` variable identifies a `char_varying` object and is not a pointer to the object.

This section provides more information about the varying-length character string data type.

You declare data as a varying-length character string with the keyword `char_varying` followed by an optional parenthesized constant. The following examples are declarations of `char_varying` strings.

```
char_varying(32) index_name;

char_varying(256) input_string;
```

The syntax for a varying-length character string declaration is as follows:

```
char_varying(n) declarator;
```

The *`n`* is an integer constant expression in the range 0 through 32,766. The value of *`n`* is the **maximum** number of characters in the string. You can omit *`n`* only if you are declaring a generic `char_varying` pointer. See the “Generic `char_varying` Pointers” section later in this chapter for information on this type of pointer.

From a conceptual viewpoint, a varying-length character string is implemented as if it were a structure containing two members: a `short` (2-byte) member containing the current length of the string, and a `char` array member containing the number of elements specified in *`n`*. Thus, a `char_varying(`*`n`*`)` object, named `cv_string`, would be allocated identically to the following structure.

```
struct
   {
   short current_length;
   char array[n];
   } cv_string;
```

Figure 4-13 represents how the following varying-length character string, `name`, is stored in memory.

```
char_varying(15) name = "John Brown";
```

In Figure 4-13, hexadecimal format is used to represent both the current length and the string. In the figure, notice that the 2-byte **current** length (000A hexadecimal) precedes the string. After the current length, the `char_varying` object contains the ASCII code for the characters in `John Brown`. Last, the five bytes that follow the ASCII code have unpredictable values (represented by `/` in the figure). In a `char_varying` string, all unused bytes have values that are unpredictable.

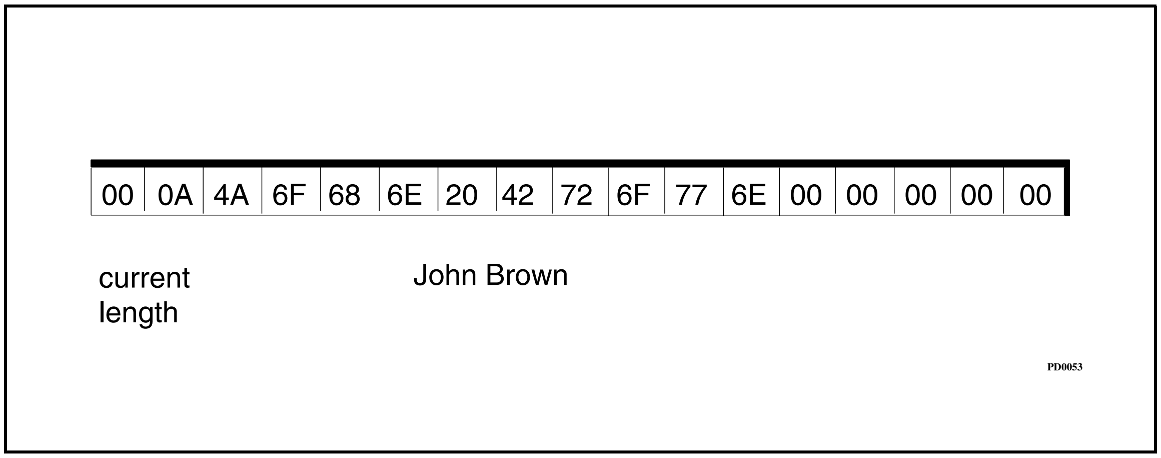


Figure 4-13. Varying-Length Character String Storage

Depending on their storage duration, the compiler initializes `char_varying` strings in the following ways. If a `char_varying` string has static storage duration and no explicit initialization, the compiler sets all bytes in the current-length field and the entire string to binary 0. If a `char_varying` string has automatic storage duration and no explicit initialization, both the current-length field and the string itself have unpredictable values.

Operations on `char_varying` Data

A subset of the VOS C operators can be used with the `char_varying` type. In addition to assignment, the operations allowed with the `char_varying` type include the following:

- all relational operations
- the concatenation operation.

The compiler allows assignments and comparisons when one operand is of the `char_varying` type and the other operand is of the `char_varying` type or is convertible to `char_varying`. All arithmetic types can be converted to `char_varying`. Arithmetic and bitwise operations are **not** allowed with `char_varying` data.

In VOS C, you can use the `$substr` built-in function to convert an array, character-string literal, or structure to the `char_varying` type. Using the `$substr` function, you can assign and compare a `char_varying` operand with an array, character-string literal, or structure operand. See the "`$substr`" section in Chapter 12 for information on this built-in function.

In VOS C, the functions for string handling in the `string.h` header file have been adapted so that they allow both C strings and `char_varying` strings as arguments. See the "String Manipulation" section in Chapter 11 for information on these functions.

You can pass a `char_varying` object as an argument in a function invocation. In addition, a function can have `char_varying(`*`n`*`)` as its return type.

Assignments with the `char_varying` Type

When the right operand in an assignment or the value of an expression is a `char_varying` string, only the number of characters indicated by the value of the current-length field are assigned. When the left operand in an assignment is a `char_varying` string, its 2-byte current length is set automatically to the number of characters assigned to the character array part of the `char_varying` string. In an assignment, the following rules apply when the left and right operands are `char_varying` or have been converted to `char_varying`.

- If the length of the left operand is at least as great as the length of the right operand, the number of characters specified by the right operand's current-length field are assigned.
- If the length of the left operand is shorter than the length of the right operand, the right operand's string is truncated (in concept) from the right side, to the length of the left operand, and then assigned.

In the following program fragment, three `char_varying` strings are used as operands in assignment expressions.

```
char_varying(10) cv_string1;
char_varying(5) cv_string2;
char_varying(10) cv_string3 = "abcdefgh";

cv_string1 = cv_string3;           /*  Assignment 1  */

cv_string2 = cv_string3;           /*  Assignment 2  */

printf("cv_string1 = |%v|\n", &cv_string1);
printf("cv_string2 = |%v|\n", &cv_string2);
```

In the first assignment, the current length of `cv_string1` is set to eight, and the characters `abcdefgh` are assigned to the character array part of `cv_string1`. In the second assignment, the current length of `cv_string2` is set to five (the maximum length of `cv_string2`), and the characters `abcde` are assigned to the character array part of `cv_string2`. Therefore, the output from the preceding program is as follows:

```
cv_string1 = |abcdefgh|
cv_string2 = |abcde|
```

Because the arithmetic types are convertible to `char_varying`, you can use the `char_varying` type in an assignment expression where one of the operands is an arithmetic type. Consider the following valid VOS C assignments involving `char_varying` and arithmetic types. In the explanation following each example, the symbol   represents a space character.

```
char c;
char_varying(5) v = "222";

c = (char)v;
printf("Output = |%d|", c);        /*  Output = |222|  */
```

After the ASCII character representation of `222` stored in `v` is converted into an integer value, the preceding assignment stores that integer value in `c`.

```
char_varying(5) v;
char c = 111;

v = (char_varying(5))c;
printf("Output = |%v|", &v);       /*  Output = |  111|  */
```

After the integer value `111` stored in `c` is converted into an ASCII character representation, the preceding assignment sets `v` to a five-character string containing the characters `  111`. Notice that `char` data is treated as a subclass of integral data. The value stored in the `char` variable is treated as an integer value, not as a character.

```
char_varying(5) v;
int i = 999;

v = (char_varying(5))i;
printf("|%v|", &v);                /*  Output = |  999|  */
```

After the integer value `999` stored in `i` is converted into an ASCII character representation, the preceding assignment sets `v` to a five-character string containing the characters `  999`.

```
char_varying(5) v;
double d = 2.5000;

v = (char_varying(5))d;
printf("|%v|", &v);                 /*  Output = | 2.50|  */
```

After the `double` value `2.5000` stored in `d` is converted into an ASCII character representation, the preceding assignment sets `v` to a five-character string containing the characters ` 2.50`.

See the “Conversions between Varying-Length Character String and Arithmetic Types” section in Chapter 5 for detailed information on the integral and floating-point conversions allowed with `char_varying` data.

Finally, one type of assignment to a `char_varying` string has special significance. The following assignment sets the current length of `v` to 0.

```
char_varying(5) v;

v = "";
printf("Output = |%v|", &v);       /*  Output = ||  */
```

A varying-length character string with a current length of 0 is called the *null string*. After the assignment in the preceding example, `v` is an instance of the null string.

Comparisons with the `char_varying` Type

You can compare a `char_varying` string to another `char_varying` string or to a data type that is convertible to `char_varying`. You perform these comparisons with the relational operators: `==`, `!=`, `>`, `>=`, `<`, and `<=`.

If one of the operands has the `char_varying` type, the other operand must have the `char_varying` type or be convertible to `char_varying`. When two `char_varying` types are the operands in a relational operation, the two strings are compared, character by character, until two characters are found that are different. The string whose differing character has the higher ASCII code compares greater. In the case of unequal length strings, the shorter operand is padded with spaces to the length of the longer operand.

> **Note:** This method of comparing two `char_varying` strings differs from the method used by the `strcmp_vstr_vstr` library function. When `strcmp_vstr_vstr` is used with two `char_varying` strings, the longer string always compares greater than the shorter if both strings contain the same characters up to the length of the shorter string.

Consider the following string comparisons where both operands are of the `char_varying` type.

- The string `"b"` is greater than the string `"a"`.
- The string `"abcd"` is greater than the string `"abca"`.
- The string `"abc "` is greater than the string `"abc\0"`.

Concatenation with the `char_varying` Type

You can append one `char_varying` string to another with the `+` operator. This "inline" operation allows you to concatenate two `char_varying` strings without invoking the `strcat_vstr_vstr` or `strncat_vstr_vstr` function. The following example uses the `+` operator to concatenate `cv_string2` and `cv_string1`.

```
char_varying(10) cv_string1 = "AAAAA";

char_varying(10) cv_string2 = "BBBBBBBBBB";

cv_string1 = cv_string1 + cv_string2;

printf("The resulting string is %v.\n", &cv_string1);
```

In the preceding `+` operation, the number of characters specified by the current-length field of `cv_string2` is concatenated with the number of characters specified by the current-length field of `cv_string1`. The type of the result is `char_varying(15)`.

Notice that because the maximum length of the left operand in the assignment expression is not sufficient to hold the concatenated string, a truncated string is assigned. In the preceding example, the concatenated string is shortened to 10 characters. Therefore, the output from the preceding example is as follows:

```
The resulting string is AAAAABBBBB.
```

Other Operations with the `char_varying` Type

A `char_varying` type or object can be used as the operand in the `sizeof` operation. For a `char_varying` type or object, the total number of bytes equals the 2-byte current length field plus the maximum length, not the current length, of the string. For example, `sizeof(char_varying(10))` yields the value 12: the 10 bytes in the character array part of the string plus 2 bytes. To get the current length of a `char_varying` object, use the `strlen_vstr` function.

A pointer to a `char_varying` object can be used in a pointer arithmetic operation, such as `cv_string + 3`. To calculate the address offset, the compiler multiplies the specified integer value by the size of the pointed-to `char_varying` object. Consider the following declaration of an array of five `char_varying` strings, and a pointer arithmetic expression involving that array.

```
char_varying(10) cv_array[5];

cv_array + 3
```

The pointer expression `cv_array + 3` yields an address that is 36 bytes after the address pointed to by `cv_array`. This address offset is calculated as follows:

```
3 * sizeof(char_varying(10))
```

Passing `char_varying` Data as Function Arguments

A `char_varying` string can be passed as an argument to a function. Such arguments are pushed in their entirety onto the stack. That is, when the calling function allocates storage for the parameter, that `char_varying` object has the same **maximum** length as was specified for the argument. It is usually desirable to pass the address of a `char_varying` string.

All arguments in C are passed by value. When you pass the **name** of a `char_varying` string as an argument to a function, the called function is given a copy of the argument rather than the original. When the name of a `char_varying` string is passed as an argument, the called function cannot modify the original string.

Note: The name of a `char_varying` string is not a pointer to the string.

In contrast, when you pass the **address** of a `char_varying` string as an argument to a function, the called function can access the original string indirectly through a pointer. When the address of a `char_varying` string is passed as an argument, the called function can modify the original string.

In the following example program, when the `func` function is invoked, a `char_varying` string's address is passed as an argument. In addition, a `char_varying` string is returned by `func`.

```
char_varying(15) name = "Michelangelo";
char_varying(15) ret_value;

char_varying(15) func(char_varying(15) *cv_ptr);

main()
{
   printf("Before func, name is %v\n", &name);

   ret_value = func(&name);  /*  Passes the address of name as the
argument  */
```

(Continued on next page)

```
   printf("After func, name is %v, ", &name);
   printf("and ret_value is also %v\n", &ret_value);}

char_varying(15) func(char_varying(15) *cv_ptr)
{
   *cv_ptr = "Raphael";

   return(*cv_ptr);
}
```

Because the address of `name` is passed as the `char_varying` argument, the called function can modify the original contents of `name` in the calling routine. The output from this program is as follows:

```
Before func, name is Michelangelo
After func, name is Raphael, and ret_value is also Raphael
```

Generic `char_varying` Pointers and Casts

VOS C allows you to omit the parenthesized integer constant expression, *n*, when you declare a pointer to a varying-length character string or cast an expression to `char_varying`. This manual uses the term *generic* to describe these `char_varying` pointers and casts. Other than for these two uses, you must specify the length (*n*) when you declare a `char_varying` string.

The next two sections explain some of the uses of generic `char_varying` pointers and casts.

> **Note:** A `char_varying` type with unspecified length **cannot** be used in contexts where storage allocation is required, such as the following: definition of a `char_varying` object, left operand in an assignment expression, operand of the `sizeof` operator, function argument, function parameter, or function return type.

Generic `char_varying` Pointers

Pointers to `char_varying` that are declared **with unspecified length** can be assigned to each other and to `char_varying` pointers that are declared with various specified lengths. A `char_varying` pointer with unspecified length is a generic pointer that is similar, in concept, to a pointer to `void`. You can use a `char_varying` pointer with unspecified length only in those contexts where knowledge of the maximum length of the pointed-to object is not required.

In the following example, the pointers `gen_ptr` and `g_ptr` are generic `char_varying` pointers that are declared with unspecified length.

```
#include <stdio.h>

char_varying *gen_ptr;          /*  Declares generic pointer  */

char_varying(30) name = "Terry", *name_ptr = &name;

void func(char_varying *);

```

(Continued on next page)

```
9    main()
10   {
11      gen_ptr = &name;
12
13      printf("%v\n", gen_ptr);
14
15      func(name_ptr);
16   }
17
18   void func(char_varying *g_ptr)    /*  Declares another generic
pointer  */
19   {
20      char_varying(30) *str_ptr;
21
22      str_ptr = g_ptr;
23
24      printf("%v\n", str_ptr);
25   }
```

Notice that generic `char_varying` pointers and `char_varying` pointers declared with a specified length are compatible in many contexts. On line 11, a generic pointer, `gen_ptr`, is assigned the address of `name`, a `char_varying` string with a specified length. On line 22, `str_ptr`, a `char_varying` pointer declared with specified length, is assigned the address of a generic pointer.

You can dereference a pointer to `char_varying` with unspecified length in all contexts except those in which the maximum length of the pointed-to string is required. Consider the two assignments in the following example containing part of a function definition.

```
void func(char_varying(32) *name_ptr, char_varying *gen_ptr)
{
   *name_ptr = *gen_ptr;       /*  Valid assignment    */

   *gen_ptr = "abcdefg";       /*  Invalid assignment  */
   .
   .
   .
}
```

The first assignment is valid. This assignment requires the compiler to know the value of the current-length field. However, the assignment does not require the compiler to know the maximum length of the object pointed to by `gen_ptr`. The second assignment is invalid because it requires the compiler to know the maximum length of the object pointed to by `gen_ptr`.

Generic `char_varying` Casts

A `char_varying` cast **with unspecified length** uses the implied length of the operand to determine the current length of the result. Consider the following example.

```
(char_varying)"Message"
```

The implied length of the operand, `"Message"`, determines the current length of the result from the cast operation. In the preceding example, the current length of the result is seven characters.

Generic `char_varying` casts are particularly useful if you call a VOS operating system subroutine. Many of these subroutine calls require the address of a `char_varying` object. Thus, the ability to cast the standard C data types is a convenience.

You can use a `char_varying` cast with unspecified length to convert integral or floating-point types, including constants, to `char_varying`. See the "Conversions between Varying-Length Character String and Arithmetic Types" section in Chapter 5 for information on these conversions.

Data Types Summary

This section provides a summary of the VOS C data types. Table 4-6 lists the VOS C arithmetic data types. For each type, the table includes the following information:

- keywords used to declare the type
- internal representation of the type
- range of allowable values for the type.

Table 4-6. Arithmetic Data Types *(Page 1 of 2)*

Data Type Keyword(s)	Internal Representation	Range
`signed char`	Signed fixed-point binary, 7-bit precision, stored in 1 byte	-128 to 127

`char`
`unsigned char`
	Unsigned fixed-point binary, 8-bit precision, stored in 1 byte	0 to 255

`short int`
`short`
`signed short int`
`signed short`
	Signed fixed-point binary, 15-bit precision, stored in 2 bytes	-32,768 to 32,767

`unsigned short int`
`unsigned short`
	Unsigned fixed-point binary, 16-bit precision, stored in 2 bytes	0 to 65,535

`int`
`signed int`
`signed`
	Signed fixed-point binary, 31-bit precision, stored in 4 bytes	-2,147,483,648 to 2,147,483,647

`unsigned int`
`unsigned`
	Unsigned fixed-point binary, 32-bit precision, stored in 4 bytes	0 to 4,294,967,295

`long int`
`long`
`signed long int`
`signed long`
	Signed fixed-point binary, 31-bit precision, stored in 4 bytes	-2,147,483,648 to 2,147,483,647

Table 4-6. Arithmetic Data Types *(Page 2 of 2)*

Data Type Keyword(s)	Internal Representation	Range
`unsigned long int` `unsigned long`	Unsigned fixed-point binary, 32-bit precision, stored in 4 bytes	0 to 4,294,967,295
`float`	Floating-point binary, 24-bit precision, stored in 4 bytes	$\pm 10^{-38}$ to $\pm 10^{38}$
`double`	Floating-point binary, 53-bit precision, stored in 8 bytes	$\pm 10^{-308}$ to $\pm 10^{307}$
`enum` object	Signed fixed-point binary, 31-bit precision, stored in 4 bytes	-2,147,483,648 to 2,147,483,647

In addition to the arithmetic types, the other VOS C types include the following:

- Array. The number of bytes allocated for an array is determined as follows:
- sizeof(*type*) `* number_of_elements`
- Pointer. All pointers are stored in four bytes, have the same format, and contain an unsigned address, `NULL`, `OS_NULL_PTR`, or the value of one of the three signal-handling macros `SIG_IGN`, `SIG_DFL`, or `SIG_ERR`.
- Structure. The storage allocated for a `struct` varies depending on the number of members and the data types and alignment requirements of those members.
- Union. The storage allocated for a `union` is equal to the number of bytes occupied by the largest member.
- Varying-length character string. The VOS C `char_varying` string type is a sequence of characters that can have a maximum declared length ranging from 0 to 32,766 characters. During a program's execution, the current length of the string can range from zero to the maximum length specified in the string's declaration. The storage allocated for a `char_varying` object is equal to the number of characters in the string **plus** two bytes (the current-length field).
- Void. The `void` type comprises an empty set of values.
- Function. Function types are discussed in Chapter 6.

Chapter 5 contains more information on allocation and alignment.

Chapter 5: Data Types: Compatibility, Conversions, and Alignment

This chapter explains three topics related to data types:

- type compatibility
- type conversions
- data alignment.

Type Compatibility

This section describes the compatibility of the VOS C object data types with each other. It also describes the compatibility of the VOS C data types with data types in other VOS languages.

See the "Compatibility with Function Types" section in Chapter 6 for information on function type compatibility.

Compatibility within the VOS C Language

In many situations, the compiler must determine whether two types are compatible. For example, certain operations, such as assigning the value of an object of one data type to an object of another data type, require type compatibility or the operation may yield incorrect results. When the value of an operand is converted to a compatible type, the value does not change.

This section describes the conditions under which two VOS C data types are compatible.

In this discussion of type compatibility, to "have the same alignment" means that two types are aligned on the same type of boundary. Consider the following examples:

```
short int $shortmap si_1;           /*  Example 1  */
short int $longmap si_2;

int $shortmap i_1;                  /*  Example 2  */
int $longmap i_2;
```

In example 1, the two variables, `si_1` and `si_2`, are allocated on an even-numbered byte boundary (mod2 boundary). Thus, the two types have the same alignment, even though one is allocated using the shortmap alignment rules, and the other is allocated using the longmap alignment rules.

In example 2, the `i_1` variable is allocated on an even-numbered byte boundary, but `i_2` is allocated on a boundary that is equal to the size of an `int` (mod4 boundary). The two types do not have the same alignment.

For information on the shortmap and longmap alignment rules, see the "Data Alignment" section later in this chapter.

Compatibility with Basic Types

The basic types consist of the signed and unsigned `char`, signed and unsigned integral, and floating-point types. Two basic types are compatible if the following conditions are true.

- Both types specify the same basic type.
- Both types have the same alignment.
- For two qualified basic types, both are the identically qualified version of a compatible type.

With the `char` and integer types, signedness is significant. Signed and unsigned versions of a type are not the same basic type and are **incompatible**.

Two types need not be identical to be compatible. For example, in VOS C, both an `int` and a `long` are stored in four bytes. A `short` is stored in two bytes. The `int` type is compatible with the `long` type. However, the `int` type is not compatible with the `short` type.

Compatibility with Array Types

Two array types are compatible if the following conditions are true.

- Both array types have compatible element types.
- Both array types have the same number of elements if the array's size is specified.
- Both array types have the same alignment.
- For two qualified array types, both are the identically qualified version of a compatible type.

You can omit the size of an array in some contexts. For example, when you declare an array using the `extern` specifier, you can omit the array's size. In such cases, the size is not considered in determining type compatibility.

Compatibility with Pointer Types

Two pointer types are compatible if the following conditions are true.

- Both pointer types point to compatible types.
- Both pointer types point to objects that have the same alignment.
- Both pointer types are identically qualified.

Furthermore, pointers to arrays, pointers to functions, and pointers to pointers have an additional condition for type compatibility. For pointers to arrays, the elements of the arrays must be identically qualified. For pointers to functions, the function types must be identically qualified. For pointers to pointers, the pointed-to objects must be identically qualified. As an example, for two pointers of the type "pointer to pointer to `int`" to be compatible, the `int` types must be identically qualified.

Consider the following example of array types that are incompatible.

```
const int (*array_ptr1)[10];

int (*array_ptr2)[10];
```

In the preceding example, both pointers have type "pointer to an array containing 10 elements of the type `int`." The elements of the two arrays are not identically qualified. The `int` elements of the array pointed to by `array_ptr1` are `const`-qualified. However, the `int` elements of the array pointed to by `array_ptr2` are not `const`-qualified. Therefore, `array_ptr1` and `array_ptr2` are **incompatible**.

Compatibility with Enumerated Types

In VOS C, two enumerated types are compatible if both types are declared with tags and the tags are identical. Or, two enumerated types are compatible if one or both types are declared without a tag and the following conditions are true.

- Both enumerated types have the same number of enumerators.
- Both enumerated types have enumerators with the same respective values.
- Both enumerated types have enumerators with the same respective names.
- Both enumerated types have the same alignment.
- For two qualified enumerated types, both are the identically qualified version of a compatible type.

In the following example, the enumerations `x` and `y` are compatible because both are declared with the same tagged type, `enum e_type`. Therefore, both have the same enumerated type.

```
enum e_type {a, b, c};

enum e_type  x, y;
```

Compatibility with Structure and Union Types

In VOS C, two structure types or two union types are compatible if both types are declared with tags and the tags are identical. Or, two structure types or two union types are compatible if one or both types are declared without a tag and the following conditions are true.

- Both types have the same number of members, and the respective members have compatible types.
- Both types have the same member names.
- Both types have the same alignment.
- For two qualified types, both types are the identically qualified version of a compatible type.
- For two structures, the members of both types are declared in the same order.

A structure or union tag names a unique user-defined type. In the following example, the structures `x` and `y` are compatible because both are declared with the tagged type `struct s_type`.

```
struct s_type {int a; int b;};

struct s_type  x, y;
```

In VOS C, if one or both of the structure or union types is declared without a tag, the two types are compatible if the conditions listed earlier are true. In the following example, the structures `s1` and `s2` are compatible.

```
struct {int i; char c;} s1;

struct {int i; char c;} s2;
```

The structures `s1` and `s2` are compatible because they contain the same number of members, declare their members in the same order, use the same member names, are identically aligned, and respective members have compatible member types.

Compatibility with Varying-Length Character String Types

In VOS C, two `char_varying` types are compatible if the following conditions are true.

- Both `char_varying` types have the same maximum length.
- Both `char_varying` types have the same alignment.
- For two qualified `char_varying` types, both are the identically qualified version of a compatible type.

You can create a *generic* `char_varying` pointer or cast by omitting the parenthesized integer constant expression (maximum length) when you declare a pointer to `char_varying` or cast an expression to `char_varying`. These generic `char_varying` pointers and casts are compatible with `char_varying` types that are declared with various specified lengths.

See the "Generic `char_varying` Pointers and Casts" section in Chapter 4 for detailed information on how to use these pointers and casts.

Compatibility with Other VOS Languages

Data can be accessed by and passed between a VOS C program and programs written in other high-level languages to the extent that both languages define the particular data type. Except for `unsigned` data, the VOS C data types, in general, are compatible with the data types defined in each of the other VOS languages. The `unsigned` types are unique to C.

When it is declared to have external linkage, data can be accessed by two programs written in different VOS languages. For example, a data structure declared to have external linkage (`extern`) in a C program can be accessed in a PL/I program where the data structure is declared with the `external` attribute.

Data can be passed to other programs through arguments. However, a C program **cannot** call a program that declares a parameter using a descriptor for additional length or bounds information. For example, a C program cannot call a PL/I procedure that declares a parameter using asterisk (`*`) extents.

See the *VOS C User's Guide (R141)* for detailed information on how to call, from a VOS C program, a program written in another VOS language.

Table 5-1 summarizes the type compatibility of the VOS C data types with the data types in other VOS languages.

Table 5-1. Cross-Language Compatibility of Data Types

BASIC	**C**	**COBOL**	**FORTRAN**	**Pascal**	**PL/I**
`name=7`	`float`	`comp-1`	`real*4`	N/A	`float binary(24)`
`name=15`	`double`	`comp-2`	`real*8`	`real`	`float binary(53)`
`name%=15`	`short`	`comp-4`	`integer*2`, logical*2	`-32768..32767`	`fixed binary(15)`
`name%=31`	`long`, int, enum	`comp-5`	`integer*4`, logical*4	`integer`	`fixed binary(31)`
`name# =(i+f,f)`	N/A	`comp-6`, binary, picture s9(`i`)(`f`)	N/A	N/A	`fixed decimal(i+f, f)`
`name$=1`	`char`	`picture x display`	`character*1`, logical*1	`char`	`character(1)`
`name$=n`	`char id [n]`	`picture x(n) display`	`character*n`	`array [1..n] of char`	`character(n)`
`name$<=n`	`char_varying(n)`	`picture x(n) display-2`	`string*n`	`string (n)`	`character(n) varying`
N/A	`(*id )()`	`entry`	N/A	N/A	`entry variable`
N/A	N/A	`label`	N/A	N/A	`label`
N/A	`type *id`	`pointer`	N/A	`^ type_name`	`pointer`

Notes:

1. In all VOS high-level languages, data types must have the **same alignment** to be compatible.
2. Though the C data type `char id[n]` is allocated similarly to the data types shown in its row, it is essentially different from these data types. For example, unlike the other data types listed in this row, an array type in C, such as `char id[n]`, is treated as a pointer in almost all contexts.

3. In VOS PL/I, the `fixed binary(15)` type does **not** support the value -32,768.

Type Conversions

In C, there are two general categories of data type conversions. An *explicit conversion* occurs when you use a cast to convert one data type to another type. An *implicit conversion* occurs when the data type of an object or function is changed during an operation other than the cast operation. For example, implicit conversions can occur in the following situations: Type conversionsConversions

- when the operands in an assignment operation are of different types
- when the operands in an arithmetic or relational operation are of different data types
- when a function is invoked
- when the value specified in a `return` statement is different from the function's declared return type.

See Chapter 6 for information on conversions involving the function type, including argument and parameter conversions and return value conversions.

See Chapter 7 for information on the conversions that occur with specific operators.

Promotions and Demotions

This section explains data type conversions where one type is promoted to another type and data type conversions where one type is demoted to another type.

Promotions

In data type conversions, a *promotion* occurs when one data type is converted to another data type of greater size. For example, if a `float` and a `double` are the operands of the addition operator (`+`), the `float` is promoted to a `double` before the expression is evaluated.

In VOS C, you can specify how some promotions will occur by using the `promotion_rules` option in a `#pragma` preprocessor control line or by using the corresponding command-line argument. Table 5-2 summarizes the effects of the three values usually used with the `promotion_rules` option: `ansi`, `traditional`, and `ansi, check`. See "The `promotion_rules` Option" section in Chapter 9 for more information on each value.

Table 5-2. Values for the `promotion_rules` Option

Value	Effect on Data Type Conversions
`ansi`	In integral promotions, `unsigned char` and `unsigned short` values are promoted to `signed int`. The variable's value is preserved. In addition, `float`-to-`double` promotion does **not** always occur before `float` data participates in an expression. The `ansi` value is the default.
`traditional`	In integral promotions, `unsigned char` and `unsigned short` values are promoted to `unsigned int`. The variable's unsignedness is preserved. In addition, `float` data is always promoted to `double` before participating in an expression.
`ansi, check`	Same as `ansi` except the compiler also issues diagnostics for certain situations where specifying `traditional` rather than `ansi` could cause differences in program behavior.

Note: For all explanations in the table, the integral promotions described for the `unsigned char` and `unsigned short` types also apply to bit fields of less than 32 bits.

The `ansi` value for `promotion_rules` is the default. Unless stated otherwise, the explanations in this section on data type conversions assume that the default ANSI C promotion rules are in effect.

Demotions

Most data type conversions do not affect the validity of the result. However, some conversions can produce unexpected and unwanted results. Consider an assignment operation in which an operand of one data type is assigned to an operand of another data type. Before a value is assigned, the value of the operand on the right side of the assignment is always converted to the type of the operand on the left side. If the type of the left operand is a data type of smaller size than the data type of the right operand, inaccurate results are possible.

This conversion from one data type to another data type of smaller size is called *demotion*. The following program fragment illustrates how demotion can produce inaccurate results. In the example, the type of the left operand is `unsigned short int`. The type of the right operand is `unsigned int`.

```
unsigned short int un_s = 0;
unsigned int un_i = 2000000000;

un_s = un_i;
printf("un_s = %d\n", un_s);        /*  Output:  un_s = 37888  */
```

The assignment `un_s = un_i;` produces a result that is probably unwanted. An `unsigned short int` operand, which is stored in two bytes, cannot accurately hold the value stored in the `unsigned int` operand, which is stored in four bytes. Therefore, the leftmost two bytes of the `unsigned int` are discarded, or truncated, and the remaining two bytes are stored in the `unsigned short int`.

If you specify the appropriate `type_checking` value in a `#pragma` preprocessor control line or the corresponding command-line argument, all demotions that involve implicit conversions where loss of precision is possible are detected at compile time. See the “The `type_checking` Option” section in Chapter 9 for information on this option.

Usual Arithmetic Conversions

Many binary operators that expect operands of arithmetic types cause conversions and yield result types in a similar way. These conversions occur in a predefined order known as the *usual arithmetic conversions*. In VOS C, both the `int` and `long` types are four bytes in length. In the usual arithmetic conversions, because `int` and `long` are equivalent types, there are certain differences between the VOS C implementation and a C implementation in which there is a distinction between `int` and `long`.

The usual arithmetic conversions, as they apply in VOS C, are as follows:

1. If either operand has the type `double`, the other operand is converted to `double`.

2. Otherwise, if either operand has the type `float`, the other operand is converted to `float`.

   Note: All floating-point constants are of the type `double`. VOS C does not currently support single-precision (`float`) floating-point constants. You can use a cast to convert a floating-point constant to a `float`.

3. Otherwise, the integral promotions are performed on both operands. Then the following rules are applied.

4. If either operand is of the type `unsigned int` or `unsigned long int`, the other operand is converted to `unsigned int`.

5. Otherwise, both operands have the type `int` (or `long`) and the type of the result is `int`.

The following integral types can be used in an expression wherever an `int` or `unsigned int` can be used:

- `signed char` and `unsigned char`
- `signed short int` and `unsigned short int`
- bit fields of less than 32 bits.

When you use one of the preceding types as an operand in an arithmetic expression, the type is promoted, by default, to a `signed int`. These promotions are called the *integral promotions*. If you use the default ANSI C promotion rules, the integral promotions preserve value including sign. The integral promotions do not affect the other arithmetic types.

On XA2000-series modules, values of the type `long double` are produced as intermediate results in arithmetic operations involving floating-point types, although VOS C does not support declarations of the type `long double`. The type `long double` is equivalent to IEEE-defined *extended long data*. See the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985) for more information on extended long data.

Conversions between Integral Types

When one integral type is converted to another integral type, the value of the original is preserved if the value can be represented by the new type. However, if a signed integral type that holds a negative value is converted to an unsigned type, the conversion **does change** the value because an unsigned integral type cannot represent negative values. In addition, some demotions from a longer integral type to a shorter integral type change the original value.

The following rules apply to conversions between integral types.

- When a shorter **signed** integral type is promoted to a longer integral type, the bit pattern of the low-order bits does not change, but the high-order bits are sign-extended.
- When a shorter **unsigned** integral type is promoted to a longer integral type, the bit pattern of the low-order bits does not change, but the high-order bits are filled with zeroes.
- When a longer integral type, signed or unsigned, is demoted to a shorter integral type, the longer type is truncated on the left to derive the converted value.

Sign extension occurs when a shorter signed integral type is promoted to a longer integral type. In sign extension, the added high-order bits in the longer integral type are filled with copies of the sign bit of the shorter signed integral type.

As an example of sign extension, consider the following conversion. A `signed char` variable with the value -4 is converted to an `unsigned short int` type. The `signed char` type stores negative numbers using two's complement format. In two's complement format, the value -4 in a `signed char` is stored as `11111100` binary. With two's complement representation, the most significant bit of a negative number is always set equal to 1.

When the `signed char` value is converted to an `unsigned short int`, the additional high-order bits (bits 15 through 8) are filled with copies of the sign bit from the `signed char` variable (that is, filled with 1's). After the conversion, the `unsigned short int` has the following bit pattern: `11111111 11111100`. The `unsigned short int` type stores numbers using binary notation. Therefore, the `unsigned short int` variable holds the value 65,532, the decimal equivalent of `11111111 11111100` binary.

Truncation occurs when a longer integral type is demoted to a shorter integral type. For example, when an `int` is converted to an `unsigned short int`, the two high-order bytes in the `int` are discarded, and the two low-order bytes are assigned to the `unsigned short int`.

Conversions between Floating-Point Types

In both `double` and `float` floating-point arithmetic, extended long data is used for intermediate results on XA2000-series modules. After the intermediate arithmetic calculations are performed, the final result is converted to the appropriate type, either `float` or `double`.

When a value of the type `float` is promoted to `double`, the exponent and the fractional part of the internal floating-point representation are extended with zeroes. The value of the `float` is not changed.

When a value of the type `double` is demoted to `float`, if the value's magnitude cannot be represented by a `float`, a run-time error occurs and a default handler is invoked. If the value being converted is in the range of values that can be represented by a `float` but cannot be represented exactly, the result is rounded to the nearest higher or lower value and then converted to `float`.

Conversions between Floating-Point and Integral Types

When a value of the type `float` or `double` is converted to an integral type, the floating-point value is truncated (not rounded) to an integer. For example, the value -1.1 is truncated to -1. A run-time error occurs, and a default handler is invoked if one of the following conditions occurs:

- if the integer part of the floating-point value is too large to store in the particular integral type
- if a negative floating-point value is converted to an unsigned integral type.

When a value in an integral type is converted to a floating-point type, the value is preserved **if** the floating-point type can represent the value exactly. In cases where the floating-point type cannot represent the integer value exactly, some loss of precision occurs. That is, the floating-point type represents the integer value with a close approximation but not the exact integer value.

Specifically, variables of the type `float` cannot exactly represent the full range of `unsigned int`, `signed int`, and `signed long int` values. The `unsigned int` type has 32-bit precision, and the `int` type has 31-bit precision. However, the `float` type has only 24-bit precision.

Conversions with Pointer Types

The `int` and `long` types, suitably cast, can be converted to pointers. Similarly, pointers can be converted to the `int` and `long` types without loss of information. In the following examples, two integer constants, a hexadecimal integer constant representing an address and the decimal integer constant 0, are cast to pointers before being assigned to `flt_ptr`.

```
#define NULL  ( (void *) 0)   /*  Macro defining the null pointer
constant   */

float *flt_ptr;

flt_ptr = (float *)0x00E0C18C;  /*  Pointer assigned the address of
a float  */

flt_ptr = NULL;              /*  Pointer assigned the null pointer
constant  */
```

In concept, a VOS C pointer can be considered an unsigned 4-byte integral object. An integral type undergoes the standard integer conversions before being converted to the pointer type.

An array name, in most contexts, is implicitly converted to a pointer type. An expression "array of *`type`*" is converted into a "pointer to *`type`*" except when such an expression is the operand of the `sizeof` operator, or the address-of operator (`&`), or is a character string literal used to initialize an array of `char`. The pointer resulting from the conversion is a pointer constant, storing the address of the array's first element. If `array_name` is an array name, the expressions `&array_name` and `array_name` yield the same address.

Similarly, a function name, in most contexts, is implicitly converted to a pointer type. An expression "function returning *`type`*" is converted into a "pointer to a function returning *`type`*" except when such an expression is the operand of the `sizeof` operator or the address-of operator (`&`). The pointer resulting from the conversion is a pointer constant, storing the address of a VOS C entry value. If `func_name` is a function name, the expressions `&func_name` and `func_name` are equivalent.

In VOS C, a pointer to any object type or function type can be converted into a pointer to a different object type or function type by using an explicit cast. Pointers to different objects or functions are guaranteed to be compatible only if they are converted by explicit cast.

For any type qualifier *`q`*, a pointer to a non-*`q`*-qualified type can be converted into a pointer to the *`q`*-qualified version of the type. The values stored in the original and converted pointers will compare equal.

In VOS C, a pointer to `void` can be converted to or from a pointer to any object type, function type, or incomplete type. In addition, when a pointer to any object, function, or incomplete type is converted into a pointer to `void`, and back again, the result will compare equal to the original pointer.

See the "Pointer Types" section in Chapter 4 for more information on and examples of pointers and pointer conversions.

See the "Generic `char_varying` Pointers and Casts" section in Chapter 4 for information on that pointer type.

Conversions between Varying-Length Character String Types

A `char_varying` value (source) can be converted to a `char_varying` type (target type) with a different maximum length.

- If the current length of the source is greater than or equal to the target type's maximum length, the converted value contains the number of characters indicated by the target type's maximum length, and the converted value's current length equals the target type's maximum length.

- If the current length of the source is less than the target type's maximum length, the converted value contains the number of characters indicated by the source's current length, and the converted value's current length equals the source's current length.

Table 5-3 shows some sample `char_varying` to `char_varying` conversions where the two types have different maximum lengths. In the table, the symbol   indicates a space character. The source string in the table is declared and initialized as follows:

```
char_varying (5) cv_string = "abcd ";
```

Table 5-3. `char_varying` to `char_varying` Conversion Examples

`char_varying` **String Value**	`char_varying` **Target Length**	**Resulting String**
`abcd`	`char_varying(3)`	`abc`
`abcd`	`char_varying(5)`	`abcd`
`abcd`	`char_varying(7)`	`abcd`

Conversions between Varying-Length Character String and Arithmetic Types

The compiler also allows conversions between the `char_varying` types and the integral and floating-point data types. The following sections explain these conversions.

Integral to Varying-Length Character String Conversions

All integral types, including signed and unsigned `char` data, can be converted to `char_varying` data. The integer is converted into a `char_varying` string containing the ASCII character representation of the integer value.

The integer is converted to a `char_varying` string as follows:

- Any leading zeroes in the integer value are discarded. The value 0 has one integral digit 0.
- If the integer value is negative, a minus sign immediately precedes the leftmost digit.
- If needed, spaces fill the string on the left.

The resulting string's length is equal to the maximum length of the `char_varying` type or 14 characters, whichever is less. If the maximum length of the `char_varying` type is less than the number of characters used to represent the value, characters are truncated from the left.

Table 5-4 shows some sample integral to `char_varying` conversions. In the table, the symbol   indicates a space character in the resulting string.

Table 5-4. Integral to `char_varying` Conversion Examples

Integer Value	**`char_varying`** **Target Length**	**Resulting String**
123	`char_varying(6)`	`   123`
-123	`char_varying(6)`	`  -123`
-123	`char_varying(2)`	`23`
-123	`char_varying(40)`	`          -123`

Floating-Point to Varying-Length Character String Conversions

All floating-point types can be converted to `char_varying` data. The `float` or `double` value is converted into a `char_varying` string containing the ASCII character representation of the floating-point value.

The floating-point value is converted into a `char_varying` string having the following form:

```
prefix mantissa E exponent
```

The *`prefix`* is a space for positive values or minus sign for negative values. The *`mantissa`* has the form:

```
integer_part.fractional_part
```

In the *`mantissa`*, the *`integer_part`* has 1 digit. If the exponent has 2 digits, the *`fractional_part`* has 14 digits. If the exponent has 3 digits, the *`fractional_part`* has 13 digits.

The mantissa is followed by the letter `E` and the *`exponent`*. The *`exponent`* contains 2 digits if the exponent value is in the range 0 through 99, or it contains 3 digits if the exponent value exceeds 99.

The resulting string's length is equal to the maximum length of the `char_varying` type or 21 characters, whichever is less. If the maximum length of the `char_varying` type is less than the number of characters used to represent the value, characters are truncated from the right.

Table 5-5 shows some sample floating-point to `char_varying` conversions. In the table, the symbol   indicates a space character in the resulting string.

Table 5-5. Floating-Point to `char_varying` Conversion Examples

Floating-Point Value	`char_varying` Target Length	Resulting String
25e0	`char_varying(32)`	` 2.50000000000000E+01`
-25e0	`char_varying(32)`	`-2.50000000000000E+01`
-25e123	`char_varying(32)`	`-2.5000000000000E+123`
-25e0	`char_varying(6)`	`-2.500`
positive infinity	`char_varying(15)`	` infinity`
negative infinity	`char_varying(15)`	` infinity`

If the floating-point value is positive or negative infinity, the resulting string's length is nine characters: the word `infinity` preceded by a space for positive infinity or a minus sign for negative infinity.

Varying-Length Character String to Integral Conversions

A varying-length character string can be converted to an integral type. The value represented by the string's ASCII characters is converted into an integer and stored in the integral variable.

The `char_varying` string can be the ASCII representation of any valid decimal integer constant or floating-point constant. The constant can have any number of leading or trailing spaces. An optional minus sign can precede the first nonspace character in the string. When the string contains a sequence of characters that does not conform to the preceding specifications, the conversion error is reported as follows:

- If the `char_varying` string is a constant expression, the compiler issues an error message.
- If the `char_varying` string is not a constant expression, a run-time error occurs.

Table 5-6 shows some sample `char_varying` to integral conversions. In the table, the symbol   indicates a space character in the string to be converted.

Table 5-6. char_varying to Integral Conversion Examples

`char_varying` **String Value**	**Integral Target Type**	**Resulting Integer Value or Conversion Error**
`  123`	`int`	123 r
`-123.987`	`int`	-123
`-123`	`int`	-123
`123-`	`int`	run-time error
`300`	`char`	run-time error

To avoid run-time errors, make sure that the integer value in the `char_varying` string can be converted to the integral type. In the last example in Table 5-6, a `char_varying` string containing the ASCII representation of the value 300 cannot be converted to the `unsigned char` type. This conversion causes a run-time error to occur. An `unsigned char` variable can store values in the range 0 through 255.

Varying-Length Character String to Floating-Point Conversions

A varying-length character string can be converted to a floating-point type. The value represented by the string's ASCII characters is converted into a floating-point value and stored in the `float` or `double` variable.

The `char_varying` string can be the ASCII representation of any valid floating-point constant or decimal integer constant. The constant can have any number of leading or trailing spaces. An optional minus sign can precede the first nonspace character in the string.

When the string contains a sequence of characters that does not conform to the preceding specifications, the conversion error is reported as follows:

- If the `char_varying` string is a constant expression, the compiler issues an error message.
- If the `char_varying` string is not a constant expression, a run-time error occurs.

Table 5-7 shows some sample `char_varying` to floating-point conversions. In the table, the symbol   indicates a space character in the string to be converted.

Table 5-7. `char_varying` to Floating-Point Conversion Examples

`char_varying` String Value	**Floating-Point Target Type**	**Resulting Floating-Point Value or Conversion Error**
`  12.0`	`float`	12.0
`-2e10`	`double`	-20000000000.0
`-123`	`float`	-123.0
`123-`	`float`	run-time error
`9.9E45`	`float`	run-time error †

† The operating system invokes whatever handler is active for the `error` condition.

Data Alignment

VOS C allows you to specify one of two data alignment methods: shortmap alignment or longmap alignment. If you do not specify either alignment method, the alignment method is the system-wide default, which is site-settable by the system administrator.

The shortmap alignment method has traditionally been used for programs designed to run on Stratus modules that contain processors from the MC68000® family. On both XA2000-series modules that use processors from the MC68000 family and XA/R-series modules that use processors from the i860™ family, longmap alignment is available and is **far more efficient** than shortmap alignment.

On programs designed to run on XA2000-series modules or XA/R-series modules, there are some situations where you must specify shortmap alignment for a data structure and where the use of longmap alignment produces **inaccurate results**. Specifically, if an existing data structure is defined using shortmap alignment rules, you will have to use shortmap alignment either implicitly or explicitly when accessing the data structure. For example, when certain structures are passed as arguments to some VOS subroutines, shortmap alignment may be required for the structures. See the descriptions of specific subroutines in the *VOS C Subroutines Manual (R068)* for information on structures that require shortmap alignment.

See the *Migrating VOS Applications to the Stratus RISC Architecture (R288)* manual for detailed information on alignment requirements.

You can direct the C compiler to allocate data for an entire source module according to a specific set of alignment rules by specifying an alignment method with the `mapping_rules` option in a `#pragma` preprocessor control line, or with the `-mapping_rules` argument to the c command. Also, you can indicate the shortmap or longmap alignment rules in a specific declaration by using the $shortmap or $longmap alignment specifier.

The compiler uses the following procedure to determine the alignment method to use for a function return type or an object type **other than** a structure, union, or enumerated type.

1. If you specify the $shortmap or $longmap alignment specifier in a declaration, the alignment rules indicated by the specifier are used.
2. If you do **not** specify alignment rules by the preceding method, the alignment rules indicated in a #pragma preprocessor control line are used.
3. If you do **not** specify alignment rules by either of the preceding methods, the alignment rules indicated in the c command's -mapping_rules argument are used.
4. If you do **not** specify alignment rules by any of the preceding methods, the system-wide default alignment rules are used. The system-wide default alignment rules are site-settable by the system administrator.

Notice that there is an order of precedence in the procedure used by the compiler to determine which alignment method applies. For example, the alignment indicated by the method in step 1 always overrides any alignment specifications indicated by the methods in steps 2 through 4.

For a structure, union, or enumerated type, the alignment rules associated with the type as specified in the type's definition determine which alignment method applies. **Once a structure, union, or enumerated type is defined, the alignment rules for that type cannot be changed, even if the default alignment rules applied when the type was defined.**

See the "Alignment Specifiers" section in Chapter 3 for information on using alignment specifiers in the declaration of an object or function.

The next four sections describe the following topics:

- #pragma options for specifying shortmap or longmap alignment
- shortmap alignment rules
- longmap alignment rules
- alignment for nested structures and unions.

For information on bit-field alignment, see the "Bit-Field Allocation" section in Chapter 4.

Pragma Options for Specifying Shortmap and Longmap Alignment

You can indicate the alignment rules for a source module by using the mapping_rules option in a #pragma preprocessor control line. If you do not specify an alignment method in a #pragma or with the -mapping_rules command-line argument, the compiler uses the system-wide default alignment method.

If you use the mapping_rules option in a #pragma, the compiler uses the specified data alignment method when laying out storage for the source module. In addition, when you specify one of the check values, the compiler checks for alignment padding in structures. The values that can be used in the mapping_rules option are shown Table 5-8. By default, the compiler uses the system-wide default alignment method. The default method is site-settable.

Table 5-8. Values for the `mapping_rules` Pragma Option

Value	Description
`shortmap`	Specifies the shortmap alignment rules.
`shortmap, check`	Same as `shortmap` except, in addition, the compiler diagnoses alignment padding within structures.
`longmap`	Specifies the longmap alignment rules.
`longmap, check`	Same as `longmap` except, in addition, the compiler diagnoses alignment padding within structures.

Using an alignment specifier, such as `$longmap`, to indicate an alignment method for an object or function return type overrides the alignment method given in the `mapping_rules` option, but the compiler still diagnoses alignment padding within the structure if you have specified one of the `check` values in the `#pragma`.

In addition to the `mapping_rules` option, you can use the `system_programming` option in a `#pragma` to diagnose alignment padding that appears in structures aligned according to the longmap alignment rules.

You must place the `#pragma` specifying `mapping_rules` at the very beginning of the source module **before** any data declarations or function definitions. If you specify the `mapping_rules` option in more than one `#pragma`, the compiler uses the last `#pragma` given to determine the alignment method for the source module.

To specify shortmap alignment for a source module, use either of the following `#pragma` options:

```
#pragma mapping_rules (shortmap)
```

or

```
#pragma mapping_rules (shortmap, check)
```

To specify longmap alignment for a source module, use either of the following `#pragma` options:

```
#pragma mapping_rules (longmap)
```

or

```
#pragma mapping_rules (longmap, check)
```

In the preceding examples, the `shortmap, check` and `longmap, check` values specify an alignment method and cause the compiler to diagnose alignment padding (gaps) in structures.

Shortmap Alignment

When you use shortmap alignment, most non-`char` data types are allocated so that they begin on an even-numbered byte boundary. With shortmap alignment, `char` data items are allocated so that they begin on a byte boundary. Table 5-9 shows the shortmap alignment rules.

Alignment boundaries are determined with the modulo operation, represented in the table as mod*n*. The *n* specifies the number of bytes by which the boundary is divisible with no remainder. For example, mod1 alignment indicates that the compiler allocates a data item so that it begins on a byte boundary. As another example, mod2 alignment indicates that the compiler allocates a data item so that it begins on one even-numbered byte boundary.

Table 5-9. Shortmap Alignment Rules

Data Type	Alignment	Size (in bytes)

`signed char`
`unsigned char`
	mod1	1

`signed short int`
`unsigned short int`
	mod2	2

`signed int`
`unsigned int`
`enum`
`signed long int`
`unsigned long int`
	mod2	4

`float`	mod2	4
`double`	mod2	8
`char_varying(`*`n`*`)`	mod2	If *n* is even, *n*+2 If *n* is odd, *n*+3
pointer	mod2	4
structure	max. of members	sum of members
`union`	max. of members	`sizeof(`*`largest_member`*`)`

For a structure or union, the shortmap alignment rules are as follows:

- The structure or union **itself** (and thus the first member) is aligned according to the maximum alignment required by the structure or union's members.
- If a structure or union contains only `char` data, the data **within** the structure or union is aligned on a mod1 boundary.
- If a structure or union contains data other than `char` data, the data **within** the structure or union is aligned according to the rules shown in Table 5-9.

The size of a structure is equal to the sum of bytes allocated for all of the structure's members plus any additional bytes needed for alignment padding. In VOS C, the size of a structure is **rounded up** so that the structure ends on a boundary that is a multiple of its alignment. For example, if a structure begins on a mod2 boundary, additional alignment padding is, if necessary, added to the end of the structure so that it also ends on a mod2 boundary.

The size of a union is equal to the number of bytes allocated for the union's largest member.

Consider, as an example of shortmap alignment, the following structure.

```
struct $shortmap struct_tag
   {
   char c;
   double d;
   int i;
   } ex_struct;
```

When the shortmap alignment rules apply, the `ex_struct` structure is allocated 14 bytes of storage. The `ex_struct` structure itself (and thus its first member) is aligned on a mod2 boundary because that alignment is the maximum alignment required by the structure's members. In this example, the `ex_struct.d` and `ex_struct.i` members require mod2 alignment. Within the structure, the members are aligned as shown in the following table.

Structure Member (or padding)	Alignment	Byte Offset	Size (in bytes)
`ex_struct.c`	mod2	0	1
alignment padding	–	1	1
`ex_struct.d`	mod2	2	8
`ex_struct.i`	mod2	10	4

Figure 5-1 illustrates shortmap alignment within the `ex_struct` structure. In the figure, each square represents one byte of memory. A square containing an identifier, such as `c`, indicates a byte used for that member. A square containing an asterisk (`*`) indicates a byte used for alignment padding.

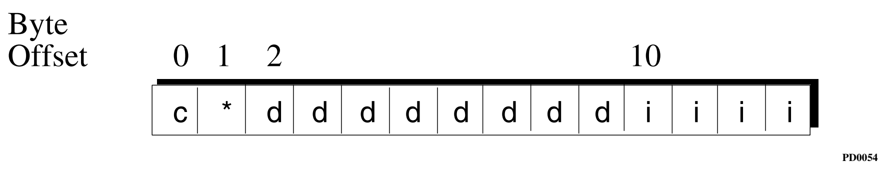


Figure 5-1. Shortmap Alignment in a Structure

The “Alignment for Nested Structures and Unions” section later in this chapter provides additional information on memory alignment with structures and unions.

Longmap Alignment

When you use longmap alignment, most scalar data types, except `char_varying`, are allocated so that they begin on an alignment boundary that is equal to the type’s size. Table 5-10 shows the longmap alignment rules.

Alignment boundaries are determined with the modulo operation, represented in the table as mod*n*. The *n* specifies the number of bytes by which the boundary is divisible with no remainder. For example, mod4 alignment indicates that the compiler allocates a data item so that it begins on a boundary that is divisible by 4.

Table 5-10. Longmap Alignment Rules

Data Type	Alignment	Size (in bytes)

`signed char`
`unsigned char`
	mod1	1

`signed short int`
`unsigned short int`
	mod2	2

`signed int`
`unsigned int`
`enum`
`signed long int`
`unsigned long int`
	mod4	4

`float`	mod4	4
`double`	mod8	8
`char_varying(`*n*`)`	mod2	If *n* is even, *n*+2 If *n* is odd, *n*+3
pointer	mod4	4
structure	max. of members	sum of members
`union`	max. of members	`sizeof(`*largest_member*`)`

For a structure or union, the longmap alignment rules are as follows:

- The structure or union **itself** (and thus the first member) is aligned according to the maximum alignment required by the structure or union’s members.
- The data **within** the structure or union is aligned according to the rules shown in Table 5-10.

The size of a structure is equal to the sum of bytes allocated for all of the structure’s members plus any additional bytes needed for alignment padding. In VOS C, the size of a structure is **rounded up** so that the structure ends on a boundary that is a multiple of its alignment. For example, if a structure begins on a mod8 boundary, additional alignment padding is, if necessary, added to the end of the structure so that it also ends on a mod8 boundary.

The size of a union is equal to the number of bytes allocated for the union's largest member.

Consider, as an example of longmap alignment, the following structure.

```
struct $longmap struct_tag
   {
   char c;
   double d;
   int i;
   } ex_struct;
```

When the longmap alignment rules apply, the `ex_struct` structure is allocated 24 bytes of storage. The `ex_struct` structure itself (and thus its first member) is aligned on a mod8 boundary because that alignment is the maximum alignment required by the structure's members. In this example, the `ex_struct.d` member requires mod8 alignment. Within the structure, the members are aligned as shown in the following table.

Structure Member (or padding)	Alignment	Byte Offset	Size (in bytes)
`ex_struct.c`	mod8	0	1
alignment padding	–	1	7
`ex_struct.d`	mod8	8	8
`ex_struct.i`	mod4	16	4
alignment padding	–	20	4

Notice that the compiler adds 4 bytes of padding to the end of `ex_struct` so that the structure ends on a boundary that is a multiple of its alignment. That is, `ex_struct` must begin **and** end on a mod8 boundary. The structure is 24 bytes long. This length provides a mod8 boundary, one that is divisible by 8 with no remainder.

Figure 5-2 illustrates longmap alignment within the `ex_struct` structure. In the figure, each square represents one byte of memory. A square containing an identifier, such as `c`, indicates a byte used for that member. A square containing an asterisk (`*`) indicates a byte used for alignment padding.

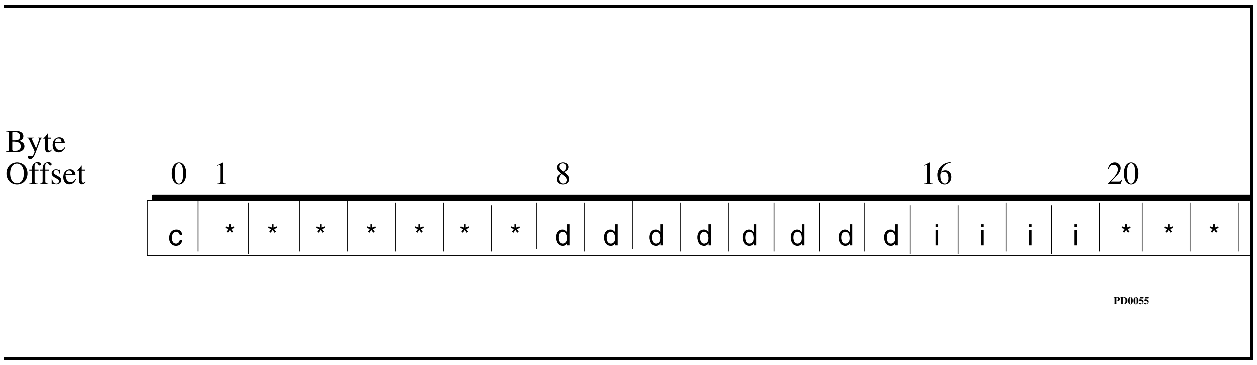


Figure 5-2. Longmap Alignment in a Structure

The “Alignment for Nested Structures and Unions” section that follows provides additional information on memory alignment with structures and unions.

Alignment for Nested Structures and Unions

A structure or union can have a member that is itself a structure or union. This individual member is a *nested* structure or union. In this discussion, a *containing* structure or union contains a nested structure or union. As an example, in the following declaration, the structure `struct_2` is a nested structure, and the structure `struct_1` is a containing structure.

```
struct type_1
   {
   char c;
   struct type_2
      {
      int i;
      double d;
      } struct_2;
   float f;
   } struct_1;
```

For nested structures and unions, the compiler uses the following method to determine whether to use the shortmap or longmap alignment rules.

1. If a nested structure or union type **does** have a `$shortmap` or `$longmap` specifier associated with its type, the alignment rules indicated by the specifier are used for that nested structure or union only. The alignment specified for a nested structure or union does not affect the alignment of the individual members of the containing structure or union.

2. If a nested structure or union type **does not** have a `$shortmap` or `$longmap` specifier associated with its type, the data within the nested structure or union is aligned according to the rules that are in effect for the containing structure or union.

See the “Alignment Specifiers” section in Chapter 3 for information on alignment specifiers and tags.

Example 1

Consider, as an example of nested structure alignment, the following structure.

```
struct $shortmap s1
   {
   char c;
   double d;
   struct $longmap s2
      {
      int i;
      double d_num;
      } inner;
   } outer;
```

The `outer` structure is allocated 32 bytes of storage. The `outer` structure itself (and thus its first member) is aligned on a mod8 boundary because that alignment is the maximum alignment required by the `outer` structure's members. In this example, the nested structure, `inner`, requires mod8 alignment. Within the `outer` structure, the members are aligned as shown in the following table.

Structure Member (or padding)	**Alignment**	**Byte Offset**	**Size (in bytes)**
`outer.c`	mod8	0	1
alignment padding	–	1	1
`outer.d`	mod2	2	8
alignment padding	–	10	6
`inner.i`	mod8	16	4
alignment padding	–	20	4
`inner.d_num`	mod8	24	8

Notice that the `inner` structure itself (and its first member, `inner.i`) is aligned on a mod8 boundary because that alignment is the maximum alignment required by the `inner` structure's members.

Example 2

Consider, as a second example of nested structure alignment, the following structure, where a tagged structure type is defined outside the containing structure and then used as a nested member of that structure.

```
struct $shortmap s2
   {
   int i;
   double d_num;
   };

struct $longmap s1
   {
   char c;
   double d;
   struct s2 inner;
   } outer;
```

The `outer` structure is allocated 32 bytes of storage. The `outer` structure itself (and thus its first member) is aligned on a mod8 boundary because that alignment is the maximum alignment required by the `outer` structure's members. In this example, the `outer.d` member requires mod8 alignment. The `inner` structure is allocated according to the `$shortmap` alignment rules, the alignment method specified when the `struct s2` type was defined. Within the `outer` structure, the members are aligned as shown in the following table.

Structure Member (or padding)	Alignment	Byte Offset	Size (in bytes)
`outer.c`	mod8	0	1
alignment padding	–	1	7
`outer.d`	mod8	8	8
`inner.i`	mod2	16	4
`inner.d_num`	mod2	20	8
alignment padding	–	28	4

Notice that the compiler adds 4 bytes of padding to the end of the `outer` structure so that it ends on a boundary that is a multiple of its alignment. That is, the `outer` structure must begin **and** end on a mod8 boundary. The structure is 32 bytes long. This length provides a mod8 boundary, one that is divisible by 8 with no remainder.

Chapter 6: Functions

A *function* is a subprogram invoked during the evaluation of a function-call expression that designates the function. A function takes zero or more arguments as input values, and returns a single value to the point of invocation. In addition to returning a value, functions often produce side effects. *Side effects* are operations that a function performs in addition to establishing a return value. Side effects can alter the execution of other functions in some way, such as by modifying data defined outside the called function.

A C program consists of one or more functions. Each function describes an algorithm used to perform a task.

This chapter explains how to declare, define, and use functions.

For information on function calls, see the “Function-Call Operator” section in Chapter 7.

Function-Prototype Information. You can declare and define a function with or without a prototype. A *function prototype* is a function declaration or definition that serves as a model specifying the number and types of a function’s parameters. In the source module where the prototype is specified, the compiler uses the prototype’s parameter information to check all subsequent references to the function. The “Advantages of Function Prototypes” section explains how this information is used. PrototypesFunction prototypes

For completeness, this chapter explains how to declare and define a function with or without a prototype. In most cases, the explanations that affect your program are in either of two sections.

- Use the information in the “Functions with Prototypes” section if your program **does use** prototypes.
- Use the information in the “Functions without Prototypes” section if your program **does not use** prototypes.

The ANSI C Standard states that the use of function definitions and declarations that do not include prototypes is an **obsolescent** feature of the C language. In new C programs, the use of function prototypes is recommended.

Function-related Terms. To understand how to declare, define, and use functions, you need to know the following terms.

A *function type* describes a function with a specified return type. A function type is characterized by its return type and the number and type of its arguments. A function type is derived from its return type. For example, if a function's return type is `type`, the function type is "function returning `type`."

A *function definition* reserves storage for the function. A function definition fully describes the function, including its storage class, return type, name, parameters, local variables, and the statements that define the function's algorithm.

A *function declaration* specifies the attributes associated with a function: storage class, return type, and name. A function declaration that specifies a prototype also includes the number and types of any parameters. A function declaration provides access to a function defined elsewhere, in another part of the source module or in another source module.

A *parameter* is an object declared as part of a function definition that acquires a value when the function is called. Parameter types are specified in a function declaration that includes a prototype. A parameter is sometimes called a formal argument or formal parameter.

An *argument* is an expression that appears between parentheses (`()`) in a function-call expression. Multiple arguments are separated by commas. An argument is sometimes called an actual argument or actual parameter.

Functions with Prototypes

This section explains how to define and declare a function in prototype form. The information on functions with prototypes is divided as follows:

- function prototypes, including the advantages of using prototypes
- function definitions with prototypes
- function declarations with prototypes.

Function Prototypes: An Overview

The function prototype mechanism is one of the more useful additions of the ANSI C Standard. A function prototype allows you to specify the number and types of the parameters that each function takes. A function prototype can be established in a function declaration and in a function definition. Figure 6-1shows where the prototyping information appears in a function declaration and function definition.

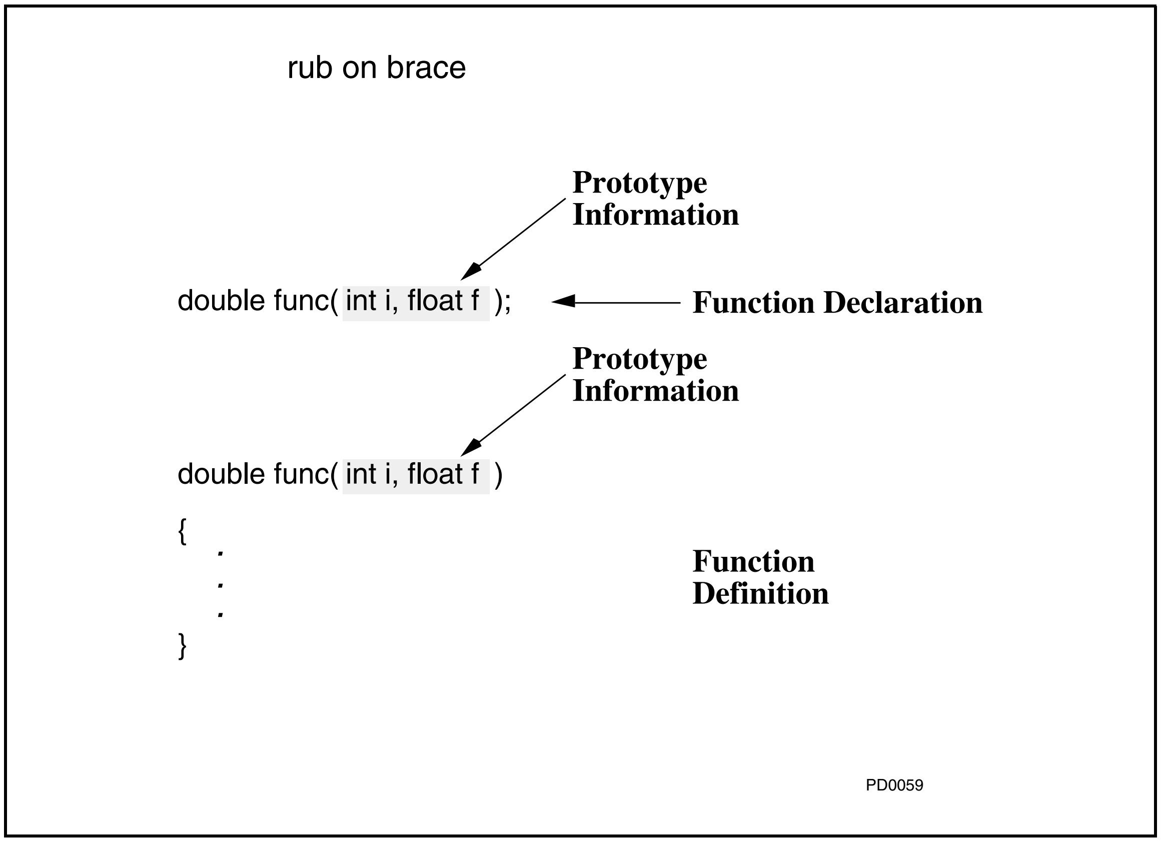


Figure 6-1. Sample Function Prototype

As shown in Figure 6-1, the prototyping information in a function definition and function declaration use almost identical syntax. To establish a function prototype, a function must be explicitly declared or defined prior to being referenced.

In VOS C, when checking for function type information, such as the function's return type and parameter information, the compiler uses one of the following:

- If the function definition is visible, the compiler uses the information in the definition of the function.
- If the function definition is not visible, the compiler uses the information in the first visible declaration of the function.

The two sections that follow explain the advantages of using functions prototypes, and describe how you can specify function prototypes within existing programs.

Advantages of Function Prototypes

Whether or not a function prototype is visible, the VOS C compiler checks that a function's return type is the same in the function's declaration and definition. The compiler also checks that the storage-class specifiers for the function type (but not the parameter types) in the function's declaration and definition are compatible.

One advantage to using a function prototype is that once a prototype is established, the compiler performs two additional checks.

First, the compiler checks that the number and types of the parameters in a function declaration, if present, match the number and types of the parameters in the function definition. The compiler also checks that the use of ellipsis notation (`...`) is consistent in the function declaration and definition. If the prototype information in the function declaration and definition does not match, the compiler issues the following error message:

```
ERROR 1904 SEVERITY 2 BEGINNING ON LINE NUM
The function "function" has conflicting prototype specifications.
```

Second, when a prototype is visible, the compiler checks that the number and types of the arguments in each function invocation match the number and types of the parameters specified in the prototype.

- If the **number** of arguments in the function invocation does not match the function prototype, the compiler issues an error message.
- If the **type** of an argument in the function invocation does not match the corresponding parameter in the function prototype, the compiler may issue a warning for an implicit conversion **depending** on the level of data-type checking that you select.

When a function is invoked and a prototype for the function is visible, the compiler implicitly converts, as if by assignment, the value of the argument to the type specified in the prototype for the corresponding parameter. If the implicit conversion is invalid, the compiler always issues a diagnostic. If the implicit conversion may result in a loss of precision, the compiler may or may not issue a diagnostic depending the level of data-type checking that you specify in a `#pragma` preprocessor control line specifying the `type_checking` option, or in the `-type_checking` command-line argument.

For example, regardless of the level of data-type checking that you select, the compiler always issues a diagnostic when you attempt to pass an argument declared as a “pointer to a function” to a parameter declared as a structure. This is an invalid conversion. As another example, the compiler issues a diagnostic when you pass an argument declared as a `float` to a parameter declared as an `int` only if the level of data-type checking is at least `normal`. The latter conversion may yield an inaccurate result. See “The `type_checking` Option” section in Chapter 9 for more information on that option.

In a prototype, you can use ellipsis notation (`...`) to tell the compiler that the function takes a varying number of arguments or arguments of varying types. The ellipsis in a function prototype causes the compiler to stop implicitly converting arguments after the last declared parameter.

Another advantage to using a function prototype is that you can explicitly control the argument conversions that occur when a function is invoked. With nonprototype form, the compiler performs the default argument promotions when a function is called. For example, with nonprototype form, the compiler always promotes arguments of the type `float` to `double`. In contrast, with function-prototype form, the compiler implicitly converts the type of an argument to the type of the corresponding parameter. No inflexible set of default argument promotions occurs.

See the “Argument and Parameter Conversions” section later in this chapter for detailed information on the conversions that occur when a function is invoked.

Existing Programs and Function Prototypes

To incorporate prototypes into an existing program, you can do either of the following:

- modify all function declarations and function definitions so that they contain the required parameter information
- modify only the function declarations so that they contain the required parameter information.

The task of modifying both function declarations and definitions is simplified by the fact that the syntax for both is nearly identical. You can copy the part of the function definition that precedes the function body and use it as the function’s declaration by preceding that part with an `extern` or `static`, as needed, and a semicolon after the closing parentheses.

A function prototype must be visible for the compiler to check the number and types of the arguments in a function call. If no function prototype is visible, the compiler performs the default argument promotions on arguments in a function invocation. Therefore, if you intend to use function prototypes, you should modify function declarations **in all source modules** so that the declarations contain the parameter information required for a prototype. Also, function-prototype form requires that each function be explicitly declared or defined before it is used.

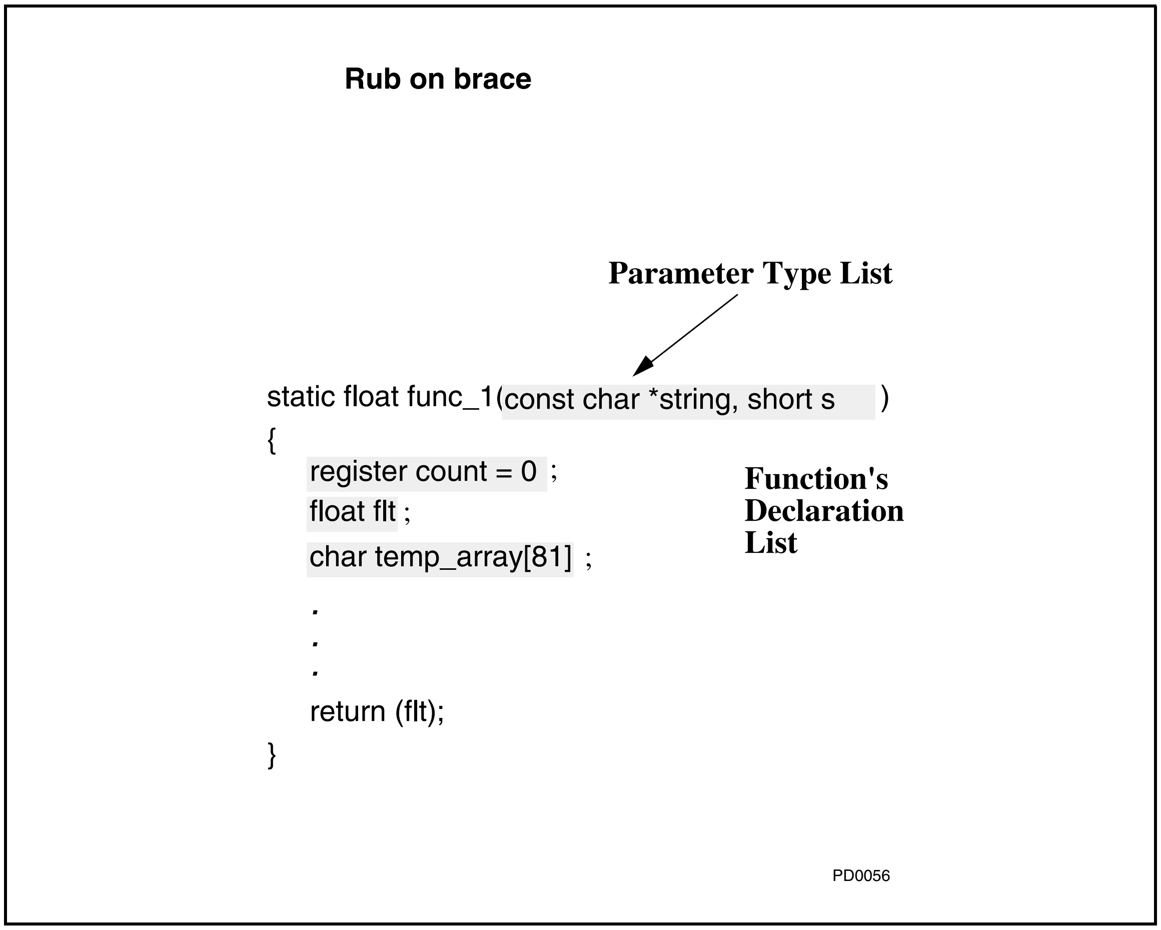
If you modify only the declaration of a function so that it contains the required parameter information, the corresponding function definition can contain a (possibly empty) identifier list. In this case, the function declaration and definition must have the same number of parameters, and the type of each prototype parameter must be compatible with the type that results from the application of the default argument promotions to the type of the corresponding parameter. For each parameter declared with function or array type, its type for these comparisons is the one that results from conversion to a pointer type as described in the “Argument and Parameter Conversions” section later in this chapter.

See the “Compatibility with Function Types” section later in this chapter for more information on the rules used to determine whether a function declaration and definition are compatible.

Function Definitions with Prototypes

In function-prototype form, a function definition includes a declaration for each parameter in the parameter type list. These parameter declarations provide the compiler with the information needed to create a prototype for the function.

Figure 6-2 shows a sample function definition with a prototype.



```
static float func_1(const char *string, short s    )
{
    register count = 0 ;
    float flt ;
    char temp_array[81] ;
    .
    .
    .
    return (flt);
}
```

Figure 6-2. Sample Function Definition with a Prototype

Each function can be defined only once. A function definition cannot appear within another function (either within the parameter type list or function body) or in a type definition.

The syntax for a function definition with a prototype is as follows:

```
[extern ] [volatile] [type_specifier] identifier (parameter_type_list)
[statis ]
      compound_statement
```

Although it is not shown in the preceding syntax, the return type given in *`type_specifier`* can have an alignment specifier, either `$shortmap` or `$longmap`, associated with it. The alignment specifier can appear anywhere before the *`identifier`*.

In a function definition, the `extern` storage-class specifier tells the compiler that the function definition has file scope and external linkage. An `extern` function definition can be accessed in other source modules where it is declared `extern`. If no storage-class specifier is given, `extern` is assumed. The `extern` specifier is typically not used on a function definition. If you indicate the `ext_shared` storage-class specifier for a function definition, the compiler interprets it as `extern`.

The `static` storage-class specifier tells the compiler that the function definition has file scope and internal linkage. A `static` function definition cannot be accessed in other source modules.

The optional `volatile` type qualifier is associated with the function itself and **not** with the function's return type. In VOS C, the `volatile` type qualifier tells the compiler that this function can return program control to some point in the calling routine other than the point of the function invocation. Also, with a `volatile` function, the state of the calling routine is unknown (and possibly changed) when program control returns to the calling routine. For example, the `setjmp` library function is declared with the `volatile` type qualifier.

> **Note:** The `const` type qualifier cannot be specified for a function type.

The *`type_specifier`* indicates the function's return type. The function's return type can be `void` or any valid object type other than an array. The return type cannot be a function type. If no type specifier is given, a return type of `int` is assumed.

When a function definition uses function-prototype form, all parameter declarations are enclosed in one set of parentheses. The *`identifier`*, parentheses, and *`parameter_type_list`* make up the function's declarator. The *`identifier`* gives the function's name.

The *`parameter_type_list`* and *`compound_statement`* are explained in the next two sections.

Parameter Type List

In VOS C, you can declare up to 127 parameters in *`parameter_type_list`*. The syntax for each parameter declaration in *`parameter_type_list`* is as follows:

```
    [register] [type_qualifier] [type_specifier] declarator
```

Although it is not shown in the preceding syntax, each parameter can have an alignment specifier, either `$shortmap` or `$longmap`, declared with it. The alignment specifier can appear anywhere before the *`declarator`*.

For each parameter, the *`parameter_type_list`* can specify an optional `register` storage-class specifier. The optional *`type_qualifier`* can be `const` or `volatile` or both. The optional *`type_specifier`* can be any valid VOS C object type. The *`declarator`* is any valid declarator and contains the parameter's identifier. The identifier for each parameter is an lvalue.

Some other rules concerning *`parameter_type_list`* are as follows:

- Only the `register` storage-class specifier is allowed for parameters in the list.
- If no *`type_specifier`* is given, the `int` type is assumed.
- If *`parameter_type_list`* contains more than one parameter, they are separated by commas.
- The *`parameter_type_list`* cannot contain initializers.

- If the `void` keyword appears instead of a list of parameters, it indicates that a function has no parameters.
- If an ellipsis (`...`) appears at the end of *`parameter_type_list`*, the function takes a varying number of arguments or arguments of varying types.

The `register` storage-class specifier tells the compiler that the parameter will be frequently accessed and should be kept in a machine register if possible.

If `...` is specified, the use of the ellipsis notation must be consistent between the function definition and the function declaration. In addition, the *`parameter_type_list`* must consist of at least one parameter if you want to access the parameters in the varying part of the parameter list using the library functions found in the `stdarg.h` header file. In the varying part of the list, the **programmer is responsible** for ensuring that the promoted types of the arguments are compatible with the types of the parameters after the parameters are promoted. If a function that accepts a varying number of arguments is defined without a parameter type list that ends with an ellipsis notation, the behavior is unpredictable.

See the “Variable Arguments” section in Chapter 11 for information on how to access a parameter in a parameter type list that uses the `...` notation.

See Chapter 3 for more information on the `register` specifier, type qualifiers, alignment specifiers, and type specifiers used in a parameter type list.

Compound Statement

In each function definition, the *function body* consists of a compound statement, which begins with an opening brace (`{`) and ends with a closing brace (`}`). The syntax for the compound statement that comprises the function body is as follows:

```
{
    [declaration ;]...
    [statement]...
}
```

Inside the compound statement, *`declaration`* can be a function or object declaration. Each *`declaration`* can include an initializer except for array, structure, or union types that have automatic storage duration. All declarations within the compound statement must immediately follow the compound statement’s opening brace.

The *`statement`* can be any single or compound statement. One or more statements perform the task or produce the side effects for which the function was designed.

If an object declaration with no storage-class specifier or with the `auto` or `register` specifier appears inside a compound statement, the object has automatic storage duration, and the object’s identifier has block scope. See the “Compound Statements” section in Chapter 8 for more information on compound statements.

Examples of Function Definitions with Prototypes

Three examples of function definitions containing prototypes and a brief explanation of each example follow.

Example 1

The following function definition includes a prototype.

```
static float func_1(const char *string, short s)
{
   register count = 0;
   float flt;
   char temp_array[81];
   .
   .
   .
   return (flt);
}
```

The declarator, `func_1(const char *string, short s)`, contains the declarations for the two parameters that `func_1` takes. After the compound statement's opening brace, `func_1` declares three local variables that have automatic storage duration and block scope.

The `static` storage-class specifier indicates that `func_1` has file scope and internal linkage. That is, the function will not be accessible in other source modules bound into the program module.

Example 2

The following function definition includes a prototype that specifies that `func_2` has no parameters because the `void` keyword appears between the parentheses instead of a list of parameters.

```
void func_2(void)
{
      .
      .
      .
   return;
}
```

The `void` return type indicates that the function returns no value.

Example 3

The following function definition includes a prototype that specifies `func_3` has one fixed parameter, `i`, which has the `register` storage-class specifier associated with it. The ellipsis notation (`...`) at the end of the parameter type list tells the compiler that the prototype gives no information about the number or types of parameters other than `i`.

```
volatile int func_3(register int i, ...)
{
   .
   .
   .
   return(i);
}
```

The function definition specifies that `func_3` is a `volatile` function that returns an `int` value. With the preceding prototype, the ellipsis notation causes the compiler to stop implicitly converting the arguments after the last declared parameter. The compiler performs the default argument promotions on the argument values that correspond to the varying part of the parameter type list.

Function Declarations with Prototypes

The syntax for a function declaration with a prototype is almost identical to the syntax for a function definition with a prototype. In prototype form, the syntax for a function declaration is as follows:

```
⌈extern ⌉
⌊statis ⌋ [volatile] [type_specifier]  identifier
          (parameter_type_list) ;
```

Although it is not shown in the preceding syntax, the return type given in *`type_specifier`* can have an alignment specifier, either `$shortmap` or `$longmap`, associated with it. The alignment specifier can appear anywhere before the *`identifier`*.

When both the declaration and definition use prototype form, the syntax for a function declaration is different in only three respects from the syntax for a function definition.

The first difference is that the identifiers for parameters within the *`parameter_type_list`* are optional. In a function declaration with a prototype, an identifier, if included, does **not** declare an lvalue of the specified name. In the declaration of a function prototype, parameter identifiers are included for documentation purposes only. The identifiers allow you to associate a meaningful name with each argument position. The parameter identifiers have function prototype scope: each identifier's scope ends when the function declarator ends.

The second difference is that a function declaration often includes the `extern` storage-class specifier to indicate that the function is defined elsewhere in the current source module or in another program source module. Typically, function definitions do not use this specifier.

The third difference is that, for each parameter declared to have the `register` storage class in the function definition, the function declaration can include or omit the `register` storage-class specifier. If `register` is specified for a parameter in the function declaration, the compiler ignores the specifier.

The following three function declarations contain prototypes for the three functions that are defined in Examples 1 through 3 of the previous section, "Examples of Function Definitions with Prototypes." See that section for an explanation of the meaning of each prototype.

```
static float func_1(const char *, short);          /*  Example 1  */

void func_2(void);                                 /*  Example 2  */

volatile int func_3(register int i, ...);          /*  Example 3  */
```

Notice that, in the first example, the two parameters do not include identifiers. Identifiers in a function declaration with a prototype are optional. The following function declaration, which includes identifiers for each parameter, is identical to the first example.

```
static float func_1(const char *string, short s);
```

In VOS C, the `static` storage-class specifier can be used for a function declaration that appears inside a function definition. In this case, the `static` specifier indicates that the function is defined within the source module. The corresponding function definition must also use the `static` storage-class specifier and must appear in the same source module. After the function is defined, the function's identifier has file scope and internal linkage.

Functions without Prototypes

A function can be defined and declared without a prototype. The nonprototype form of a function declaration and definition is allowed so that an existing program that does not use prototypes can still be compiled without modification. However, when a prototype is not visible, the compiler does **not** check that the number and types of the arguments in a function call match the number and types of the parameters specified in the function definition.

Note: The ANSI C Standard states that the use of function definitions and declarations that do not include prototypes are an **obsolescent** feature of the C language. The use of nonprototype form in new C programs is discouraged.

This section explains how to define and declare a function in nonprototype form. The information on functions without prototypes is divided as follows:

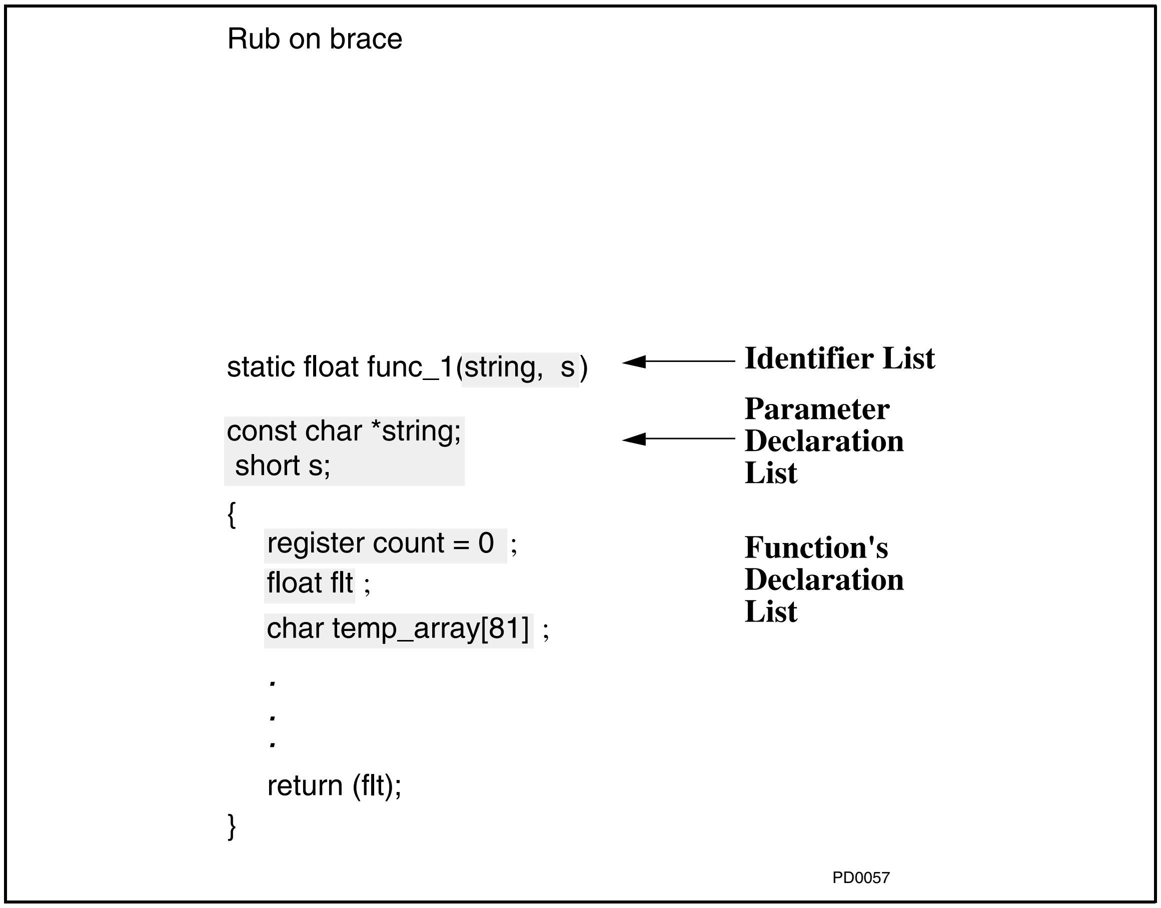
- function definitions without prototypes
- function declarations without prototypes.

The "Compatibility with Function Types" section later in this chapter provides a detailed explanation of the conditions under which two function types are compatible. These conditions apply when matching a function declaration and function definition where prototypes are not present.

Function Definitions without Prototypes

The "old-style" form for a function definition does not include a function prototype. For a function definition without a prototype, each parameter's identifier is specified in an identifier list enclosed in parentheses following the function's name. These parameters are then declared in a parameter declaration list that appears immediately before the function body's opening brace (`{`).

Figure 6-3 shows a sample function definition without a prototype.



```
static float func_1(string,  s)

const char *string;
 short s;
{
    register count = 0  ;
    float flt ;
    char temp_array[81] ;
    .
    .
    .
    return (flt);
}
```

Figure 6-3. Sample Function Definition without a Prototype

Each function can be defined only once. A function definition cannot appear within another function (either within the parameter declaration list or function body) or in a type definition.

The syntax for a function definition without a prototype is as follows:

```
[extern]  [volatile] [type_specifier]  identifier ([identifier_list])
[statis]
          [parameter_declaration_list]
          compound_statement
```

Although it is not shown in the preceding syntax, the return type given in *`type_specifier`* can have an alignment specifier, either `$shortmap` or `$longmap`, associated with it. The alignment specifier can appear anywhere before the *`identifier`*.

Most elements of the preceding syntax are identical to the elements used to define functions with prototypes. See the "Function Definitions with Prototypes" section earlier in this chapter for a description of the following elements of a function definition:

- the optional `extern` storage-class specifier
- the optional `static` storage-class specifier
- the optional `volatile` type qualifier
- the optional *`type_specifier`*
- the required *`compound_statement`*.

When a function definition uses nonprototype form, the *`identifier_list`* contains the names, but not the types, of all parameters. If a function takes no parameters, the parentheses are empty. In VOS C, you can list up to 127 parameters in *`identifier_list`*. Identifiers for parameters are separated by commas.

> **Note:** An identifier previously used as a `typedef` name in the source module **cannot** be reused as an identifier in the *`identifier_list`*.

The *`identifier`*, parentheses, and *`identifier_list`* make up the function's declarator. The *`identifier`* gives the function's name.

Parameter Declaration List

The *`parameter_declaration_list`* contains declarations for each of the parameters named in *`identifier_list`*. The syntax for each parameter declaration in the *`parameter_declaration_list`* is as follows:

```
[register] [type_qualifier] [type_specifier] declarator ;
```

Although it is not shown in the preceding syntax, each parameter can have an alignment specifier, either `$shortmap` or `$longmap`, declared with it. The alignment specifier can appear anywhere before the *`declarator`*.

For each parameter, the *`parameter_declaration_list`* can specify an optional `register` storage-class specifier. The optional *`type_qualifier`* can be `const` or `volatile` or both. The optional *`type_specifier`* can be any valid VOS C object type. The *`declarator`* is any valid declarator. It is illegal to declare a *`declarator`* in the *`parameter_declaration_list`* that is not named in the function definition's *`identifier_list`*. The identifier for each parameter is an lvalue.

Some other rules concerning *`parameter_declaration_list`* are as follows:

- Only the `register` storage-class specifier is allowed for parameters in the list.
- If no *`type_specifier`* is given, the `int` type is assumed.
- The *`parameter_declaration_list`* cannot contain initializers.

For a function definition without a prototype, you do not use the `void` keyword inside the declarator's parentheses to indicate that the function has no parameters. In nonprototype form, the empty parentheses in the declarator specify that the function takes no parameters.

For a function definition without a prototype, you do not use the ellipsis notation (`. . .`) to indicate a varying number of parameters or parameters of varying types. When you use nonprototype form, the compiler does not check the number or types of parameters.

Examples of Function Definitions without Prototypes

Three examples of function definitions that do not contain prototypes and a brief explanation of each example follow.

Example 1

The following function definition uses nonprototype form to define a function that takes two parameters and returns a value of the type `float`.

```
static float func_1(string, s)

const char *string;
short s;

{
   register count = 0;
   float flt;
   char temp_array[81];
   .
   .
   .
   return (flt);
}
```

The declarator, `func_1(string, s)`, specifies the identifiers for the two parameters, `string` and `s`, that `func_1` takes. Each parameter is declared in the parameter declaration list that precedes the function's body. After the compound statement's opening brace, `func_1` declares three local variables that have automatic storage duration and block scope.

The `static` storage-class specifier indicates that `func_1` has file scope and internal linkage. That is, the function will not be accessible in other source modules bound into the program module.

Example 2

The following function definition uses nonprototype form to define a function that takes no parameters.

```
void func_2()
{
      .
      .
      .
   return;
}
```

The `void` return type indicates that the function returns no value.

Example 3

The following function definition uses nonprototype form to define a function that takes one parameter, `i`, which has the `register` storage-class specifier associated with it.

```
volatile int func_3(i)

register int i;

{
   .
   .
   .
   return(i);
}
```

The function definition specifies that `func_3` is a `volatile` function that returns an `int` value. See the “Function Definitions with Prototypes” section earlier in this chapter for information on `volatile` functions.

Function Declarations without Prototypes

The syntax for a function declaration without a prototype is almost identical to the syntax for a function definition without a prototype. In nonprototype form, the syntax for a function declaration is as follows:

```
┌extern ┐
│statis │ [volatile] [type_specifier] identifier () ;
└       ┘
```

Although it is not shown in the preceding syntax, the return type given in *`type_specifier`* can have an alignment specifier, either `$shortmap` or `$longmap`, associated with it. The alignment specifier can appear anywhere before the *`identifier`*.

When both the declaration and definition use nonprototype form, the syntax for a function declaration is **different** in only two respects from the syntax for a function definition.

- The parentheses following *`identifier`* are always empty. The empty parentheses tell the compiler that the declaration provides no information about the number or types of parameters. The empty parentheses do **not** mean that the function takes no parameters.
- A function declaration often uses the `extern` storage-class specifier to indicate that the function is defined elsewhere in the current source module or in another program source module. Typically, function definitions do not use this specifier.

The following three function declarations declare the functions that are defined in Examples 1 through 3 of the previous section, “Examples of Function Definitions without Prototypes.” See that section for an explanation of the attributes of each function.

```
static float func_1();             /*  Example 1  */

void func_2();                     /*  Example 2  */

volatile int func_3();             /*  Example 3  */
```

In VOS C, the `static` storage-class specifier can be used for a function declaration that appears inside a function definition. In this case, the `static` specifier indicates that the function is defined within the source module. The corresponding function definition must also use the `static` storage-class specifier and must appear in the same source module. After the function is defined, the function’s identifier has file scope and internal linkage.

Compatibility with Function Types

Compatibility between two function types is required in a number of contexts. For example, the declaration and definition of a function must specify compatible function types. As another example, for two pointers to functions to be compatible, the two pointers must point to compatible function types.

For two function types to be compatible, the following conditions must be true regardless of whether the function types are specified in prototype or nonprototype form.

- Both function types specify compatible return types.
- For two qualified function types, both are the identically qualified version of a compatible type.

In addition to these conditions, when one or both of the function types are in prototype form, **additional conditions** must be met for two function types to be compatible. These additional conditions are explained in the sections that follow. Based on the function types being compared, Table 6-1 lists the function-type compatibility topics explained in each section.

Table 6-1. Function Type-Compatibility Topics

Function Types Being Compared	**See This Section in Chapter 6**
Both function types are in prototype form.	“Matching Prototyped Function Types”
A function definition is in nonprototype form, and another function type is in prototype form.	“Matching Nonprototyped Function Definitions”
One function type is in prototype form, and another function type is in nonprototype form.	“Matching Other Nonprototyped Function Types”
One or both function types have a parameter declared with a qualified type.	“Matching Qualified Parameters”

When both function types are in nonprototype form, the compiler does not compare the parameter types when checking for type compatibility because no parameter type information is available outside of the function definition.

Some other factors that can affect function type compatibility are as follows:

- When a parameter is declared to have function or array type, its type for the type-compatibility comparisons is the one that results from conversion to the corresponding pointer type.
- When a parameter is declared to have a qualified type, the parameter's type for the type-compatibility comparisons is the unqualified version of its declared type. See the "Matching Qualified Parameters" section, later in this chapter, for information on matching qualified parameters.

Two function types naming the same function are required to be compatible. When two function types naming the same function are not compatible, the compiler always issues a diagnostic. When two function types naming the same function do not have identical (as opposed to compatible) return types, the compiler issues a diagnostic if you specify at least the `minimum` value for the `type_checking` option in a `#pragma` preprocessor control line or in the corresponding command-line argument.

Matching Prototyped Function Types

For two function types that have parameter type lists (that is, have prototypes) to be compatible, the lists must agree in the number of parameters, in the use of the ellipsis terminator, and corresponding parameters must have compatible types. In addition, both function types must have compatible return types and similarly qualified function types.

The following example illustrates the rules for function type compatibility when two function types have prototypes. The example contains a function declaration and a function definition that are specified in prototype form.

```
int func(int x);                  /*  Function declaration  */

int func(register int y)          /*  Function definition   */
{
   .
   .
   .
}
```

In the preceding example, the declaration and definition of `func` specify compatible function types. The `int` return type given in the declaration and definition are identical. The two parameter type lists specify the same number of parameters, and the parameters have compatible types.

Notice that, in the preceding example, the identifiers given with the parameters do not need to be identical. In a function declaration, identifiers for parameters are optional and do not affect function type compatibility. Also, whether or not the `register` storage-class specifier is used with a parameter is irrelevant in determining compatibility. However, the alignment of a parameter, whether specified explicitly or implicitly, must be identical or corresponding parameters will not have compatible types.

Matching Nonprototyped Function Definitions

The function-type compatibility rule explained in this section applies when a function type that uses prototype form is compared with the function type specified in a function definition that uses nonprototype form. For example, the rule applies when a function declaration is specified in prototype form, but a function definition is specified in nonprototype form. In such a case, the following rule determines if two function types are compatible.

If one function type has a parameter type list (that is, has a prototype) and the other function type is specified by a function definition that contains a, possibly empty, identifier list (that is, does not have a prototype), both must agree in the number of parameters, and the type of each prototype parameter must be compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier. In addition, both function types must also have compatible return types and similarly qualified function types.

The following example illustrates the preceding rule. The example contains a function declaration that is specified in prototype form and a function definition that is specified in nonprototype form.

```
void func_a(int i, double d);      /*  Function declaration  */

void func_a(s, f)                  /*  Function definition   */
short s;
float f;
{
    .
    .
    .
}
```

In the preceding example, the function declaration and definition specify compatible types because all conditions for function type compatibility are met.

- Both function types specify compatible return types.
- Both function types have the same number of parameters.
- The type of each prototype parameter is compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier in the function definition.

In the default argument promotions, arguments of the type `float` are promoted to `double`. Arguments of type signed and unsigned `char`, and signed and unsigned `short int` are promoted to `int`. In the preceding example, the types of the prototype parameters, an `int` and a `double`, in the function declaration are compatible with the promoted types that result from the application of the default argument promotions to the corresponding identifiers in the function definition. In the preceding example, the type of `i` (`int`) is compatible with the promoted type of `s`. After the default argument promotions are applied to `s`, the type that results is `int`. Also, the type of `d` (`double`) is compatible with the promoted type of `f`. After the default argument promotions are applied to `f`, the type that results is `double`.

Consider another example that uses the same function definition for `func_a` but applies the preceding rules to pointer types.

```
void func_a(s, f)
short s;
float f;
{
   .
   .
   .
}

void (*f_ptr1)(int, double);     /*  Compatible with a pointer to
func_a      */

void (*f_ptr2)(short, float);   /*  Not compatible with a pointer to
func_a  */
```

The function type-compatibility rules also affect whether two function pointer types are compatible. For two function pointer types to be compatible, both pointers must point to compatible function types.

In the preceding example, the address of `func_a` can be assigned to `f_ptr1`. However, if you attempt to assign the address of `func_a` to `f_ptr2`, the compiler issues a diagnostic for an implicit conversion from one pointer type to another pointer type that locates a different function. The function type-compatibility rules affect whether the function types pointed to by `f_ptr1` and `f_ptr2` are compatible.

- In the declaration of `f_ptr1`, the types of the pointed-to function type's parameters, an `int` and a `double`, **are** compatible with the promoted types that result from the application of the default argument promotions to the corresponding identifiers in the definition of `func_a`.
- In the declaration of `f_ptr2`, the types of the pointed-to function type's parameters, a `short` and a `float`, **are not** compatible with the promoted types that result from the application of the default argument promotions to the corresponding identifiers in the definition of `func_a`.

Matching Other Nonprototyped Function Types

The function-type compatibility rule explained in this section applies when one function type that uses prototype form is compared with another function type that is not part of a function definition and that uses nonprototype form. For example, this rule applies when a function declaration or definition is specified in prototype form, but another declaration for the function is specified in nonprototype form. In such a case, the following rule determines if the two function types are compatible.

If one function type has a parameter type list (that is, has a prototype) and the other function type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list (that is, does not have a prototype), the parameter type list cannot have an ellipsis terminator, and the type of each parameter must be compatible with the type that results from the application of the default argument promotions. In addition, both function types must also have compatible return types and similarly qualified function types.

The following example illustrates the preceding rule. The example contains two sets of function declarations that mix prototype and nonprototype form.

```
void func_b(int i);           /*  Example 1:  compatible function
types    */
void func_b();


void func_c(short s);         /*  Example 2:  incompatible function
types  */
void func_c();
```

In example 1, the two function declarations for `func_b` specify compatible types because all conditions for function type compatibility are met.

- Both function types specify compatible return types.
- The parameter type list does not have an ellipsis terminator.
- The type of the parameter, an `int`, is compatible with the type that results from the application of the default argument promotions.

In example 2, the two function declarations for `func_c` do not specify compatible types because the type of the parameter, a `short`, is incompatible with the type that results from the application of the default argument promotions. In the default argument promotions, `short` values are promoted to `int`.

Matching Qualified Parameters

When a parameter is declared with qualified type, the parameter's type for the type-compatibility comparisons is the unqualified version of its declared type.

To determine function type compatibility, the compiler converts array types and function types to pointer types. A declaration of a parameter "array of *`type`*" is converted to "pointer to *`type`*." A declaration of a parameter "function returning *`type`*" is converted to "pointer to function returning *`type`*." If a parameter is a pointer type or is a parameter that is converted to a pointer type, the compiler uses, for the function type-compatibility comparisons, the unqualified version of the pointer type. Furthermore, if a parameter is a pointer type, type qualifiers for types pointed to by the pointer type must match exactly.

The following examples illustrate compatible and incompatible function types that specify a qualified parameter.

```
int func_1(int x);            /*  Example 1:  compatible function
types    */

int func_1(const int x)
{
   .
   .
   .
}

int func_2(int *y);           /*  Example 2:  incompatible function
types  */

int func_2(const int *y)
{
   .
   .
   .
}

int func_3(int *const z);     /*  Example 3:  compatible function
types    */

int func_3(int *z)
{
   .
   .
   .
}
```

In all of the preceding examples, both function types specify compatible return types and have the same number of parameters. The following explanation focuses only on the type qualification of the parameters.

In example 1, the function types specified in the declaration and definition of `func_1` are compatible. In this case, to determine type compatibility, the compiler compares the **unqualified** versions of the parameter's declared types. In both the function declaration and definition, the unqualified version of the parameter's type is `int`.

In example 2, the function types specified in the declaration and definition of `func_2` are incompatible. Because the parameter `y` is a pointer type, the compiler compares the **unqualified** versions of the pointer types. In addition, the compiler checks that type qualifiers for types pointed to by the pointer types match exactly. In the function declaration, the type pointed to by `y` is an unqualified `int`. In the function definition, the type pointed to by `y` is a `const`-qualified `int`. The two function types are incompatible because the pointed-to types are **not** identically qualified.

In example 3, the function types specified in the declaration and definition of `func_3` are compatible. Because the parameter `z` is a pointer type, the compiler compares the

unqualified versions of the pointer types. In both the function declaration and definition, the unqualified version is “pointer to `int`.” In addition, the compiler checks that type qualifiers for types pointed to by the pointer types match exactly. In both the function declaration and definition, the type (`int`) pointed to by `z` is similarly unqualified.

For information on how type qualifiers are used in the declarations of pointer types, see the “Type Qualifiers and Pointer Types” section in Chapter 3.

Argument and Parameter Conversions

This section describes the argument and parameter conversions that occur when a function is invoked. In general, the conversions that occur differ significantly depending on whether a function prototype is visible when the function is called.

Two argument conversions **always** occur regardless of whether a function prototype is visible. The compiler converts array expressions and function designators, when used as arguments, to pointers before the call. A *function designator* is an expression, such as a function name, that has function type. The compiler converts an argument of the type “array of *`type`*” to “pointer to *`type`*.” The compiler also converts an argument of the type “function returning *`type`*” into “pointer to function returning *`type`*.”

In a similar manner, two parameter adjustments **always** occur in a function definition. The compiler adjusts a parameter of the type “array of *`type`*” to “pointer to *`type`*.” The compiler also adjusts a parameter of the type “function returning *`type`*” into “pointer to function returning *`type`*.” In both cases, the resulting parameter type is an object type.

Argument Conversions with Prototypes

When a function is invoked and a function prototype for the called function’s type **is** visible, the compiler performs certain conversions. The compiler converts, as if by assignment, the type of each argument to the type of the parameter specified in the prototype. If a function’s prototype uses ellipsis notation (`. . .`) at the end of the parameter type list, the compiler stops this type of conversion after the last declared parameter. However, the default argument promotions are performed on any arguments in the varying part of the parameter type list. See the next section, “Argument and Parameter Conversions without Prototypes,” for a description of the default argument promotions.

The following example illustrates argument conversions where a function prototype is visible.

```
int func(int i, int i1);          /*  Function prototype  */

main()
{
   int num, i_num = 0;
   double d_num = 9.9;
```

(Continued on next page)

```
   num = func(i_num, d_num);      /*  Arguments are converted to the
types  */
     .                       /*  of the parameters in the prototype   */
       .
       .
}

int func(int i, int i1)           /*  Function definition  */
{
       .
       .
       .
}
```

In the preceding example, the function prototype for `func` specifies two `int` arguments. The first argument, `i_num`, is passed without conversion because `i_num` is already an `int`. However, because the second argument, `d_num`, is a `double`, its value is converted to an `int` type. Thus, the value 9.9 is truncated to 9 before it is passed to the `i1` parameter in `func`.

Conversion with Constant Operands of the Address-of Operator. In VOS C, certain constant values can be used as the operand of the address-of operator (`&`) if the expression is part of an argument list. If you pass an argument that uses such a construct and a prototype for the called function type is visible, the compiler generates a constant having the type of the corresponding parameter in the prototype. The compiler allocates storage for the constant value for the duration of the call. The address that is passed to the called function is the address of this temporary object. Consider the following example:

```
int func_1(short *);               /*  Function prototype  */
   .
   .
   .
func_1(&123);                      /*  Function call       */
```

In the preceding example, the constant `123` has the `int` type. However, the compiler uses the prototype information to generate a `short` constant for the expression `&123`. The compiler allocates temporary storage for this `short` constant and assigns its address to the corresponding parameter.

For more information on the use of the `&` operator with a constant that appears as part of an argument list, see the “Using Constant Operands with the Address-of Operator” section in Chapter 7.

Argument and Parameter Conversions without Prototypes

When a function is invoked and a function prototype for the called function’s type is **not** visible, the compiler performs the default argument promotions. In the *default argument promotions*, arguments of the type `float` are promoted to `double`, and the integral promotions are performed on signed and unsigned `char`, signed and unsigned `short int`, and bit-field arguments of less than 32 bits. In VOS C, these integer types are normally promoted to `int`. See “The `promotion_rules` Option” section in Chapter 9 for information on the `#pragma` options that you can use to specify promotion rules.

In addition to the default argument promotions, the VOS C compiler also adjusts parameters of the type `float` to `double`. If a parameter is a signed or unsigned `char`, or is a signed or unsigned `short int`, the compiler uses only the appropriate part of the value that is passed by the calling function. For example, if a `short` is specified as an argument in a function call, the compiler converts the value of the `short` argument to an `int` before the value is passed to the called function. If the corresponding parameter is declared as a `short`, the compiler assigns only the `short` part (least significant two bytes) of the value to the `short` parameter.

When a function prototype is not visible, the compiler performs no other argument conversions, and does not issue diagnostics if the number or types of the arguments are incompatible with the number or types of the parameters. The **programmer is responsible** for ensuring the following:

- that the number of arguments agrees with the number of parameters
- that the promoted types of the arguments are compatible with the types of the parameters after the parameters are promoted.

The following example illustrates argument conversions where a function prototype is not visible.

```
int func();                    /*  Function declaration without a
prototype  */

main()

{
   int num = 0;
   short s_num = 0;
   float f_num = 0.0;

   num = func(s_num, f_num);   /*  Arguments undergo the default   */
      .                        /*  argument promotions             */
      .
      .
}

int func(i, f)                 /*  Function definition without a
prototype   */
int i;
float f;
{
      .
      .
      .
}
```

In the preceding example, no function prototype is visible when `func` is invoked. Therefore, the compiler performs the default argument promotions on the arguments. The value of the first argument, `s_num`, is converted to an `int`, and the value of the second argument, `f_num`, is converted to a `double`. In addition, the type of the formal parameter, `f`, is adjusted from `float` to `double`.

Argument Passing

In C, all arguments are passed by value. In preparing for a call to a function, each argument is evaluated, and each parameter is assigned the value of the corresponding argument. On XA2000-series modules, the calling function pushes the value of each argument onto the stack. On XA/R-series modules, the calling function typically stores the value of each argument in a machine register if a machine register is available and is capable of storing the value. The called function can change the values of its parameters, but the changes to the parameters are local and do not affect the original values of the arguments.

Though C does not provide by-reference argument passing, it is possible to modify a variable in the calling function by passing a pointer to the object. In the called function, you can use the indirection operation to change the value of the pointed-to object. Thus, the called function can access and change variables in the calling function.

Treatment of an array name within the called function is unique. When an array name, "array of `type`," is used as an argument, it is converted to "pointer to `type`" before it is assigned to the corresponding parameter. Because the array name argument is passed as a pointer to the array's first element, the address is copied into the corresponding parameter. In the called function, you can access and change any element in the original array in the usual manner through subscripting or pointer arithmetic and indirection.

The following program illustrates the effects of passing three types of arguments to a called function:

- an object
- an address of an object
- an array name.

```
void func(int, int *, char *);              /*  func prototype  */

main()
{
   int int_arg, int_arg1;
   char array_arg[81];

   int_arg = 100;
   int_arg1 = 100;
   strcpy(array_arg, "Last modified in main");

   func(int_arg, &int_arg1, array_arg);    /*  func invocation  */

   printf("BEFORE CALL: int_arg = 100\n");
   printf("AFTER CALL: int_arg = %d\n\n", int_arg);

   printf("BEFORE CALL: int_arg1 = 100\n");
   printf("AFTER CALL: int_arg1 = %d\n\n", int_arg1);

   printf("BEFORE CALL: array_arg = Last modified in main\n");
   printf("AFTER CALL:  array_arg = %s\n", array_arg);
}
```

(Continued on next page)

```
void func(int int_param, int *int_ptr, char *array_ptr) /*  func
definition  */
{
   int_param = 999;

   *int_ptr = 999;

   strcpy(array_ptr, "Last modified in func");
}
```

The output from the preceding program is as follows:

```
BEFORE CALL: int_arg = 100
AFTER CALL: int_arg = 100

BEFORE CALL: int_arg1 = 100
AFTER CALL: int_arg1 = 999

BEFORE CALL: array_arg = Last modified in main
AFTER CALL:  array_arg = Last modified in func
```

In the preceding program and output, notice the following points about the arguments that are passed to `func` when it is invoked, and the original values of the arguments when program control returns to `main`.

- The first argument, `int_arg`, is an object. The called function's `int_param` is passed the value of `int_arg`. The called function **cannot change** the original value of this argument.
- The second argument, `&int_arg1`, is the address of an object. In the called function, this address is passed to a pointer, `int_ptr`. Using the indirection operation (`*int_ptr`) to access the pointed-to object, the called function **can change** the original value of `int_arg1`.
- The third argument, `array_arg`, is an array name. Before it is passed to `func`, the array name is converted to a pointer to the first element of `array_arg`. That address is then assigned to the corresponding parameter, `array_ptr`. Using array subscripts, or pointer arithmetic and indirection, the called function **can change** the original values of the array's elements.

See the "Array Types" section in Chapter 4 for more information on arrays.

Return Values

A function's return value corresponds to its return type. The return type can be `void` or any valid object type other than an array. A function's return type cannot be a function type. A function's return type can, however, be a pointer to an array or a pointer to a function.

The called function can return a value to the calling function through the use of the `return` statement. If the returned value does not have the same data type as is specified in the function's definition, the returned value is converted, as if by assignment, to the return type indicated in the function's definition. That converted return value becomes the value of the function-call expression when program control returns to the calling function. A function-call expression is **not** a modifiable lvalue and cannot appear as the left operand in an assignment statement.

Depending on the level of data-type checking you select in the `type_checking` option in a `#pragma` preprocessor control line or in the corresponding command-line argument, the compiler can issue an error message in the following situations:

- if you use a `return` statement specifying a return value in a function defined as returning `void`
- if you use a `return` statement with no return value in a function defined as returning some value other than `void`.

When program control reaches a function's closing brace (`}`), the equivalent of a `return` statement with no return value is executed. See "The `return` Statement" section in Chapter 8 for more information on the `return` statement.

The `main` Function

Most C programs define a function named `main` as the entry point of the program module. The `main` function specifies where program execution begins. In VOS C, a system function, named `s$start_c_program`, is used to initiate most C programs. In `s$start_c_program`, the `main` function is invoked as follows:

```
return_code = main(argc, argv, envp);
```

A program can use the `argc` and `argv` parameters to read arguments that are entered on the command line with the program's name. When the `main` function terminates, it can return an `int` value to the `s$start_c_program` function that called it. The `s$start_c_program` function declares no function prototype for `main`. The `main` function is usually defined with no parameters or with two parameters, `argc` and `argv`, though any names can be used for these parameters. By convention, the names for the two parameters to `main` are `argc` and `argv`.

Note: In VOS C, `envp` is passed as an argument when `main` is called but, otherwise, is not used. The `envp` pointer always equals the `NULL` pointer constant. The ANSI C Standard does not include the `envp` parameter to `main` in its definition of that function.

This explanation of the `main` function is divided as follows:

- program startup and the program entry point
- command-line arguments
- program termination.

Program Startup and the Program Entry Point

In VOS C, a function defined with the name `main` is, by default, the main entry point of an executable program when the following conditions are met.

- An object module resulting from a source module contains a function named `main` that has external linkage.
- This object module is the first (or only) one named when the `bind` command is invoked.
- No main entry point is specifically named in an `entry` directive in a binder control file.

When these conditions are met, a VOS-supplied function is bound into the program module and used as the program's main entry point. This program-startup function, named `s$start_c_program`, is invoked when a typical C program begins execution. The `s$start_c_program` function invokes `main`.

The `s$start_c_program` function sets up command-line arguments in the format described in the next section, "Command-Line Arguments." This function also enables the VOS C signals (for example, `SIGFPE`) and sets up the default handlers that will be invoked if the operating system raises these error-condition signals during the program's execution. In addition, `s$start_c_program` calls the `clock` function to initialize the variable that stores the amount of CPU time used by the program. Then, `s$start_c_program` invokes the function named `main`.

If the `main` function is **not** the main entry point of the executable program module, the command-line arguments are not formatted in the conventional manner. In this case, command-line arguments conform to the standard VOS conventions and can be processed using the operating system service subroutines: `s$parse_command`, or `s$get_arg_count` and `s$get_arg`. See the *VOS C Subroutines Manual (R068)* for information about these subroutines.

Also, if the `main` function is **not** the main entry point of the executable program module, the `clock` function's processor-time variable is not initialized. In addition, the VOS C signals, such as `SIGFPE`, are not enabled.

The name `main` has no special significance other than as the object modules' default entry point. The name `main` can be used legitimately in any other context in the source module (for example, to define data or labels).

Command-Line Arguments

When you are at command level, you can run a program by entering the name of an executable program module. VOS C allows you to enter command-line arguments on the command line with the program's name. The `main` function is passed these command-line arguments when it is invoked. Typically, a program uses command-line arguments as input values for tasks that the program performs. For example, the command-line argument `addr_file` could be the name of a file that the program opens and updates.

A C program can read these command-line arguments by using the `argc` and `argv` parameters to `main`. The `argc` and `argv` parameters are sometimes called *program parameters*.

The `argc` parameter is an `int` variable containing the argument count: the number of strings entered on the command line. The value in `argc` is also the number of pointers in the `argv` array. The minimum value of `argc` is 0, and the number of pointers in the array equals `argc` + 1.

The `argv` parameter is an array of pointers to `char`. Each pointer contains the address of a string argument value, entered on the command line. Each string is delimited by one or more space or horizontal tab characters. The first pointer, `argv[0]`, points to the name of the first string entered on the command line (that is, the program module's name). If the value of `argc` is greater than 1, the strings pointed to by `argv[1]` through `argv[argc - 1]` represent command-line arguments. The last pointer, `argv[argc]`, is the `NULL` pointer constant.

The `argc` and `argv` parameters and the strings pointed to by the `argv` array can be modified by the program and retain their last-stored values between program startup and program termination.

The following program illustrates how to read command-line arguments using the `argc` and `argv` parameters.

```
main(int argc, char *argv[])
{
   int i;

   printf("argc = %d\n", argc);

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

If `display_args addr_file Thomas Jones 'John Smith' 1000` were entered on the command line, the output from the preceding program, `display_args`, would be as follows:

```
argc = 6
argv[0] = display_args
argv[1] = addr_file
argv[2] = Thomas
argv[3] = Jones
argv[4] = John Smith
argv[5] = 1000
```

Notice that, if a command-line argument (for example, `John Smith`) consists of more than one word separated by spaces or tabs, enclosing the words in apostrophes causes the words to be read as one argument.

Program Termination

A C program ends when one of the following occurs.

- Program control reaches the `main` function's closing brace.
- A `return` statement in the `main` function executes.
- An `exit` function in the program executes.

If a `return` statement in `main` executes or program control reaches the `main` function's closing brace (`}`), the program-startup function, `s$start_c_program`, disables the handlers for the signals that it previously enabled. The `s$start_c_program` function then calls `exit` with a `status` argument equal to the value returned from `main`.

If `main` returns a nonzero return value other than -1, it usually signifies that the program has abnormally terminated. The `main` function's return value determines the value of the `command_status` variable. The value of `command_status` is 0 unless the program returns a nonzero value from `main` other than -1, calls the `exit` function, or calls the `s$error` or `s$stop_program` subroutine. If the `main` function returns the value -1, the value of `command_status` is 0 unless the program called `s$error` and passed a nonzero value as the `error` code argument.

In the following example, the `return` statement terminates the `main` function and returns program control to `s$start_c_program`.

```
main()
{
   .
   .
   .
   return (999);
}
```

When the `s$start_c_program` function ends, the program's process is placed at command level. The value of `(command_status)` will be 999.

See the *VOS Commands Reference Manual (R098)* for more information on the `(command_status)` command function.

Function Pointers

A function address can be specified in two forms:

- as a function name
- as a function pointer.

For this discussion, a function name is considered a function pointer.

Except when a function name is the operand of the address-of operator (`&`), a function name is implicitly converted to "pointer to function returning *`type`*." As an example, if the `func` function has the type "function returning `int`," the function name, `func`, is converted to "pointer to function returning `int`." When a function name is the operand of the address-of operator (`&`), a function name is explicitly converted to "pointer to function returning *`type`*."

The address-of operator indicates that the expression `&`*`function_name`* should yield the function's address. Therefore, the address-of operation is the only operation in which a function name is not implicitly converted to a pointer because the operation itself explicitly converts the function name to a pointer.

Like an array name, a function name is an address value. Like an array name, it is a **pointer constant**, not a pointer variable. A function name is not a modifiable lvalue and cannot be used as the left operand of an assignment operator. A function name can appear anywhere a pointer constant can appear.

In contrast to a function name, you can define a **pointer variable** of the type "pointer to function returning *`type`*" that points to a function address. This function pointer is a modifiable lvalue and can appear as the left operand of an assignment statement.

The following example contains both a function name and a function pointer.

```
int func(void);             /*  Declare the function       */

int (*func_ptr)(void);      /*  Define a function pointer  */

func_ptr = func;            /*  Assign the function's address to the
pointer  */
```

After the function pointer, `func_ptr`, is assigned the address of `func`, the pointer can be used similarly to the function name, `func`. However, unlike a function name, `func_ptr` is a pointer variable containing the address of the `func` function. See the "Function Pointer Operations" section later in this chapter for information on the operations that you can perform with function pointers.

A pointer variable of the type "pointer to function returning *`type`*" can contain values other than a function address. It can contain the `NULL` or `OS_NULL_PTR` constant. It can also contain one of the three signal-handling macros `SIG_IGN`, `SIG_DFL`, and `SIG_ERR`, as defined in the `signal.h` header file. See the "Signal Handling" section in Chapter 11 for a description of these macros.

VOS C Entry Values

In VOS C, the address of a function is **not** the memory address where that function's code is stored, but rather it is the address of a three-pointer array that holds certain information about the function. This three-pointer array is called an *entry value* following the VOS PL/I terminology. Figure 6-4 shows the format of a VOS C entry value.

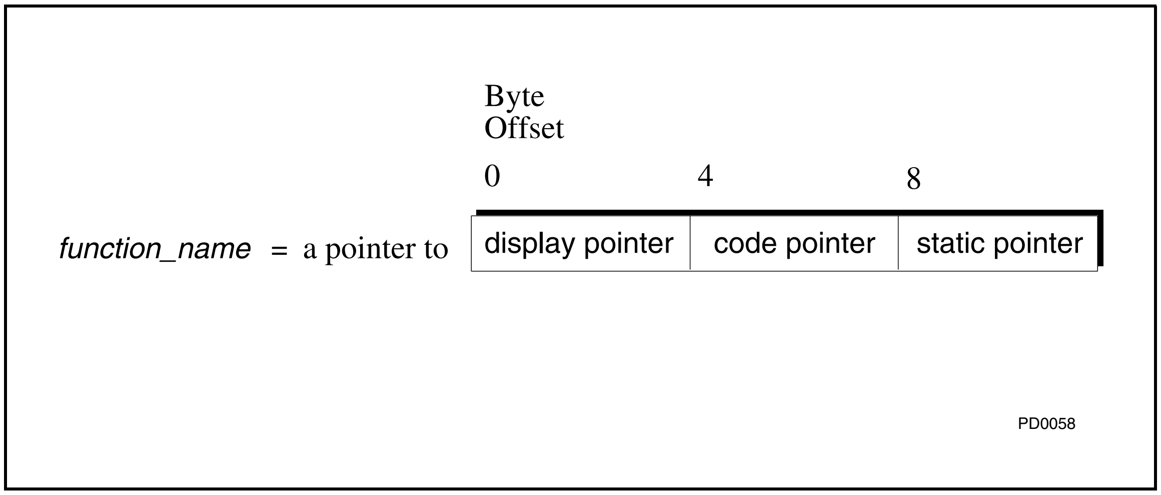


Figure 6-4. VOS C Entry Value

When a C program defines a function, the compiler allocates a 12-byte entry value containing a three-pointer array. If a C program defines a function pointer, the compiler allocates a 4-byte variable that can be assigned the address of a function (entry value). When an object module is bound, the binder initializes the three pointers of the entry value so that the function name and function pointer can be used to reference the function.

When a VOS C function pointer contains the address of a function, it points to an entry value consisting of three pointers. The values held by the three pointers can vary depending on the VOS high-level language used by the program.

- In VOS C, the *display pointer* always contains the value of the `OS_NULL_PTR` constant. In block-structured languages like PL/I and Pascal, the display pointer can contain the address of block-activation information, or the `OS_NULL_PTR` constant.
- The *code pointer* contains the address of the first executable instruction of the function. This location **cannot** be modified.
- The *static pointer* contains the address where the function's static data (data having static storage duration) is stored.

Function Pointer Operations

With the exception of the address-of operation, all operations on functions involve function pointers: either a function name or a function pointer. The set of operations allowed on function names and function pointers is similar.

A function name or a function pointer can be used with the function-call operator (`()`) to invoke a function. For example:

```
int func(char arg1, short arg2);       /*  Function declaration
*/

int (*func_ptr)(char arg1, short arg2);   /*  Function pointer
declaration   */

return_value = func(arg1, arg2);          /*  Call using a function
name     */

func_ptr = func;

return_value = (*func_ptr)(arg1, arg2);   /*  Call using a function
pointer  */
```

Notice that, in the preceding example, `func` is called twice. The second invocation uses the indirection operator (`*`) to dereference the function pointer, and then invoke the function pointed to by `func_ptr`.

A function name, which is a pointer constant, can be used as the right operand in an assignment statement. In contrast, a function pointer, which is a pointer variable, can be used as the left operand **or** the right operand in an assignment statement. For example:

```
int func1(void);

int (*func_ptr1)(void), (*func_ptr2)(void);

func_ptr1 = func1;

func_ptr1 = func_ptr2;
```

In the preceding example, the function name, `func1`, can be used only as the right operand in the first assignment. However, the function pointers, `func_ptr1` and `func_ptr2`, are used as the left or right operand in the second assignment.

Pointers to functions that have different parameter type information have different types. In VOS C, a pointer to `void` can be converted to or from a pointer to any function type. A pointer to a function type can be converted to a pointer to `void` and back again, and the result will compare equal to the original pointer.

Two function pointers, either a function name or a function pointer variable, can be compared for equality. Two function pointers are equal if one of the following conditions is true.

- Both contain `NULL` or `OS_NULL_PTR`.
- Both contain `SIG_IGN`, `SIG_DFL`, or `SIG_ERR`.
- Both locate the same function.

Two function pointers locate the same function if each points to an entry value that contains identical values for the display pointer and code pointer elements of the entry-value array. In a multitasking program, the static pointer, which can be different for the same entry executing in different tasks, does not need to be the same.

If it is necessary to access the static pointer, you can use the following expression:

```
*((int **)func_ptr + 2)          /*  Accesses the static pointer   */
```

In the preceding example, the expression `*((int **)func_ptr + 2)` increments the address of `func_ptr` by the length of two pointers (eight bytes), casts the result to a "pointer to a pointer to an `int`," and references the contents of the specified address.

Function names and function pointers can be passed as arguments in a function invocation.

The `ext_shared` storage-class specifier is not allowed for function pointers. In a tasking program, a function pointer **cannot** be shared among tasks because the static pointer, within the pointed-to entry value, is different for each task.

In VOS C, the `sizeof` operator can be used with a function name or a function pointer. With a function name, the result is the size of an entry value: 12 bytes. With a function pointer, the result is the size of a pointer: 4 bytes.

Chapter 7:
Expressions and Operators

An *expression* is a sequence of operators and operands that does one or a combination of the following:

- specifies the computation of a value
- designates an object or function
- generates side effects.

An *operator* performs an operation (evaluation) on one or more operands. An *operand* is a subexpression on which an operator acts.

This chapter explains expressions and operators, including the syntax and use of each operator.

In this chapter, many explanations of the C operators use the phrase *compatible* type or types. For information on what constitutes a compatible type, see the "Type Compatibility" section in Chapter 5.

Lvalues and the Value of an Expression

An *lvalue* is an expression that designates an object. An *object* is a region of storage, the contents of which can represent values. For example, the name of a variable is an lvalue designating a particular region of storage. An expression that is an lvalue can have an object or incomplete type other than `void`.

The term "lvalue" was originally derived from the assignment expression `E1 = E2`, where the left operand must be a modifiable lvalue. Rather than being associated with the left operand in an assignment expression, an lvalue is now perhaps better considered as an object "locator value."

Certain operators, such as the assignment operator (`=`), require an operand that is a modifiable lvalue. Other operators yield lvalues.

A *modifiable lvalue* is an expression that designates an object whose contents the program can change.

A *nonmodifiable lvalue* is an expression that designates an object whose contents the program cannot change. The following types of objects are nonmodifiable lvalues:

- an array type (in many contexts, a pointer constant)
- an incomplete type
- a `const`-qualified type
- a structure or union containing a member with a `const`-qualified type.

The right operand (`E2`) in an assignment expression is sometimes called an *rvalue*. This document refers to an rvalue as “the value of an expression.”

An lvalue always has an associated data type, the type of the corresponding object. Similarly, the value of an expression always has an associated data type. Its type is determined either by the definition of the object from which the value is derived or by the type conversion rules associated with the operator that yields the particular value.

When an lvalue is an operand, the compiler performs certain conversions. The description of each operator, later in this chapter, specifies what those conversions are. In addition to the conversions specified in the description of each operator, two other conversions occur as described in the following paragraphs.

An lvalue that does not have array type is converted to the value stored in the designated object and, therefore, is no longer an lvalue. However, this conversion does **not** occur if the nonarray lvalue is the left operand in an assignment, the left operand of the structure/union member operator (`.`), or the operand of one of the following operators: `sizeof`, address-of (`&`), or any of the increment (`++`) or decrement (`--`) operators. If the nonarray lvalue has a qualified type, the value has the unqualified version of the type of the lvalue.

An lvalue that has type “array of *`type`*” is converted to an expression that has type “pointer to *`type`*.” This pointer is the address of the first element of the array and is not a modifiable lvalue. However, this conversion does **not** occur if the array lvalue is the operand of the `sizeof` operator or the address-of operator, or is a character-string literal used to initialize an array of `char` or a `char_varying` string.

Operator Precedence and Associativity

The rules of operator precedence and associativity determine the order for executing the operations within an expression.

Operator precedence determines the order in which operations are performed. Operators with higher precedence are processed before operators with lower precedence.

Associativity determines which operation is performed first if two operators of the same precedence level appear in an expression. Operators of the same level of precedence have the same associativity.

Table 7-1 shows the VOS C operators arranged in descending order of precedence. The table also shows how operators associate within a given level of precedence.

Table 7-1. Precedence and Associativity of Operators *(Page 1 of 2)*

Operator	**Name**	**Associativity**

`[]`
`()`
`.`
`->`
`++`
`--`
	Array subscript
	Function call
	Structure/union member
	Structure/union pointer
	Postincrement
	Postdecrement
		Left to right

`&`
`*`
`sizeof`
`()`
`-`
`~`
`!`
`++`
`--`
	Address of
	Indirection
	Size of
	Cast
	Unary minus
	Bitwise complement
	Logical negation
	Preincrement
	Predecrement
		Right to left

`*`
`/`
`%`
	Multiplication
	Division
	Remainder
		Left to right

`+` `-`	Addition Subtraction	Left to right

`<<`
`>>`
	Left shift
	Right shift
		Left to right

`<`
`<=`
`>`
`>=`
	Less than
	Less than or equal to
	Greater than
	Greater than or equal to
		Left to right

`==`
`!=`
	Equality
	Inequality
		Left to right

`&`	Bitwise AND	Left to right
`^`	Bitwise exclusive OR	Left to right
|	Bitwise inclusive OR	Left to right
`&&`	Logical AND	Left to right
||	Logical OR	Left to right
`? :`	Conditional	Right to left

`=`
 	
`*=`
	Assignment
	
	Multiplication assignment
		Right to left

Table 7-1. Precedence and Associativity of Operators *(Page 2 of 2)*

Operator	Name	Associativity
`/=`	Division assignment	
`%=`	Remainder assignment	
`+=`	Addition assignment	
`-=`	Subtraction assignment	
`<<=`	Left-shift assignment	
`>>=`	Right-shift assignment	
`&=`	Bitwise-AND assignment	
`^=`	Bitwise exclusive-OR assignment	
|=	Bitwise inclusive-OR assignment	
,	Comma	Left to right

The order of evaluation of subexpressions and the order in which side effects take place can vary **except as indicated** in Table 7-1 or as specified in the description of the following operators: logical AND (`&&`), logical OR (`||`), function-call (`()`), conditional (`? :`), and comma (`,`).

You can use parentheses within an expression to indicate explicitly the order of evaluation, overriding the normal precedence of operators. Operations within the parentheses are performed first. See the "Parenthesized Expressions as Primary Expressions" section later in this chapter for information on parenthesized expressions.

The following example and explanation illustrate how the compiler uses the rules of operator precedence and associativity to determine the order in which operations within an expression are performed.

```
int code, num;

code = num = 50 * -3 / (8 * 9 + (4 - 1));
```

The preceding expression is evaluated as follows:

1. Parenthesized expressions are evaluated first. The nested parenthesized expression `(4 - 1)` yields the value 3. The expression becomes:

   ```
   code = num = 50 * -3 / (8 * 9 + 3);
   ```

2. Within the second parenthesized expression `(8 * 9 + 3)`, the `*` operator has a higher level of precedence than the `+` operator. Therefore, the multiplication is performed before the addition, yielding the value 72. The expression becomes:

   ```
   code = num = 50 * -3 / (72 + 3);
   ```

3. The parenthesized expression `(72 + 3)` yields the value 75. The expression becomes:

```
code = num = 50 * -3 / 75;
```

4. Within the expression `50 * -3 / 75`, the `-` operator is applied to the value 3, yielding a negative value. The `*` and `/` operators have the same level of precedence. Because these operators associate from left to right, the multiplication is performed first, yielding the value -150. The expression becomes:

```
code = num = -150 / 75;
```

5. The division operation `-150 / 75` yields the value -2. The expression becomes:

```
code = num = -2;
```

6. The assignment operations are performed last because the `=` operator has a lower level of precedence than all of the preceding operations. The `=` operator associates from right to left. Therefore, the rightmost assignment `num = -2` yields the value -2 and, as a side effect, assigns the value -2 to `num`. The expression becomes:

```
code = -2;
```

7. The second assignment expression `code = -2` assigns, as a side effect, the value -2 to `code`.

Primary Expressions

Primary expressions include identifiers for functions and objects, constants, character-string literals, and parenthesized expressions. The following examples are primary expressions.

```
int_num                 /*  Object name               */

func                    /*  Function name             */

char_array              /*  Array name                */

123                     /*  Integer constant          */

222.999                 /*  Floating-point constant   */

'a'                     /*  Character constant        */

"a string"              /*  Character-string literal  */

(a + 999)               /*  Parenthesized expression  */
```

Identifiers as Primary Expressions

An identifier that is declared as designating an object or function is a primary expression. The following identifiers are primary expressions.

- An identifier declared to be of an arithmetic, enumerated, structure, union, pointer, or varying-length character string type is a primary expression and an lvalue. Depending on how the type is qualified, the lvalue can be modifiable or nonmodifiable.
- An identifier declared to be an array name or function name is a primary expression and a nonmodifiable lvalue.
- An identifier declared to be an enumeration constant is a primary expression that yields an integer value and, in general, is not an lvalue.

An identifier for a label, type definition, or tag is **not** a primary expression. See the "Identifiers" section in Chapter 2 for more information on identifiers.

Constants as Primary Expressions

A constant is a primary expression. Constants are grouped into the following categories: integer constants, floating-point constants, character constants, and enumeration constants. The type associated with a constant depends on the form and value of the constant. For example, a character constant such as `'a'` has, by default, the type `unsigned char`. See the "Constants" section in Chapter 2 for more information on constants.

In VOS C, a constant is not an lvalue except when it appears in an argument list as the operand of the `&` operator. In this case, the constant is an lvalue while the called procedure is active. See the "Address-of Operator" section later in this chapter for information on using a constant with the `&` operator.

Character-String Literals as Primary Expressions

A character-string literal, sometimes called a string literal, is a primary expression. A character-string literal has the type "array of `char`." When used in an expression, a character-string literal is converted into a pointer to the first character in the array. Like an array name, a character-string literal is a nonmodifiable lvalue. See the "Character-String Literals" section in Chapter 2 for more information on string literals.

Parenthesized Expressions as Primary Expressions

A parenthesized expression is considered a primary expression because the compiler evaluates the contents of a parenthesized expression before evaluating any containing expression outside the parentheses. The type and value of a parenthesized expression are identical to those of the unparenthesized expression. A parenthesized expression is an lvalue if the enclosed expression is an lvalue.

Using parentheses around an expression causes operations within the parentheses to be performed first, overriding the normal precedence of operators.

Postfix Operators

A *postfix operator* is an operator that appears immediately after its operand. Table 7-2 shows the postfix operators.

Table 7-2. Postfix Operators

Operator	Name	Example	Operation
`[]`	Array subscript	`array_name[0]`	Accesses an array element
`()`	Function call	`func()`	Invokes a function
`.`	Structure/union member	`s.mem1`	Accesses a structure member or union member
`->`	Structure/union pointer	`s_ptr->mem1`	Accesses a structure member or union member
`++`	Postincrement	`num++`	Increments its operand's value after the result is obtained
`--`	Postdecrement	`num--`	Decrements its operand's value after the result is obtained

Array-Subscript Operator

You use the array-subscript operator (`[]`) to reference an element of an array. The syntax for an expression using the array-subscript operator is as follows:

```
array_name [ subscript ]
```

The *`array_name`* has the type "pointer to object *`type`*" and is typically an array name. When an array name is the operand of the array-subscript operator, the compiler converts the array name into a pointer to the first element of the array. The *`subscript`* is an arbitrary expression that evaluates to an integer value, specifying the array element to access. An expression using the array-subscript operator has the type *`type`* and can be a modifiable lvalue depending on how the type is qualified.

Arrays in C are zero-based. The first element of an array named `array_id` is referenced by `array_id[0]` or, equivalently, `*array_id`. The expression `array_id[`*`n`*`]` is identical to the following indirection operation:

```
*(array_id + (n))
```

Because of the conversion rules that apply to the `+` operator, if `array_id` is an array object and *`n`* is an integer, `array_id[`*`n`*`]` specifies the *`n`*-th element of the array, counting from 0.

See the "Array Types" section in Chapter 4 for more information on and examples of how the array-subscript operator is used with arrays, including multidimensional arrays.

Function-Call Operator

You use the function-call operator (`()`) to invoke a function. The syntax for an expression using the function-call operator is as follows:

```
function_designator ([ argument_expression_list ] )
```

The *`function_designator`* is an expression that denotes the called function and is typically a function name. The compiler converts *`function_designator`* into a "pointer to function returning *`type`*," where *`type`* can be the `void` type or any object type other than an array. The operand of the function-call operator can itself be a pointer to a function. In this case, no conversion to a pointer is performed.

If the expression that precedes the parenthesized *`argument_expression_list`* consists solely of an identifier that has not been declared or defined prior to the function call, the identifier is implicitly declared as if the following declaration appeared in the innermost block containing the function call.

```
extern int identifier();
```

The optional *`argument_expression_list`* specifies arguments to the function. Each argument in the list must be an expression of an object type. Multiple arguments are separated by commas. In VOS C, you can specify up to 127 arguments. An empty set of parentheses indicates no arguments. In preparing for the call to the function, each argument is evaluated, and each parameter is assigned the value of the corresponding argument. In C, all arguments are passed by value. On XA2000-series modules, the calling function pushes the value of each argument onto the stack. On XA/R-series modules, the calling function typically stores the value of each argument in a machine register if a register is available and capable of storing the value.

The function-call expression yields a value of the type specified in *`type`*. A function-call expression is not an lvalue. Recursive function calls are allowed.

The following program fragment illustrates use of the function-call operator in an assignment statement.

```
int ex_func(double arg1, char *arg2);      /*  Function prototype  */

double arg1;
char arg2[20];
int return_value;
   .
   .
   .
return_value = ex_func(arg1, arg2);        /*  Function call       */
```

In the preceding example, the function call invokes the `ex_func` function and passes two arguments, `arg1` and `arg2`, to the called function. See Chapter 6 for detailed information on and examples of function calls, including argument conversions and argument passing.

Structure/Union Member and Pointer Operators

You can use either of two operators to access structure and union members:

- the structure/union member operator (`.`)
- the structure/union pointer operator (`->`).

The syntax for an expression using the structure/union member operator is as follows:

```
expression.member_name
```

The *`expression`* is an expression that has qualified or unqualified structure or union type. In VOS C, *`expression`* must be an lvalue. Typically, *`expression`* is the identifier for a structure or union variable. The *`member_name`* specifies a member contained in the structure or union. The value of an expression using the `.` operator is that of the named member and is an lvalue. The result of the expression has the same data type as was specified for the member. If *`expression`* is a qualified type, the result is the similarly qualified version of the specified member.

The syntax for an expression using the structure/union pointer operator is as follows:

```
pointer
```

The *`pointer`* is an expression that has the type "pointer to qualified or unqualified structure" or "pointer to qualified or unqualified union." The *`member_name`* specifies a member in the pointed-to structure or union. The value of an expression using the `->` operator is that of the named member of the structure or union object to which *`pointer`* points. An expression using the `->` operator is an lvalue. The result of the expression has the same data type as was specified for the member. If *`pointer`* points to a qualified type, the result is the similarly qualified version of the specified member.

You use the structure/union member operator (`.`) if the left operand is a structure or union variable. You use the structure/union pointer operator (`->`) if the left operand is a pointer to a structure or union. With either operator, the expression's result is the value of the member.

Consider the following declaration and assignment:

```
struct
   {
   int num;
   char letter;
   } s, *s_ptr;

s_ptr = &s;       /* Assigns the structure's address to the pointer */
```

Figure 7-1 shows how the structure/union member and structure/union pointer operators can be used to reference the `num` member of the structure declared in the preceding example.

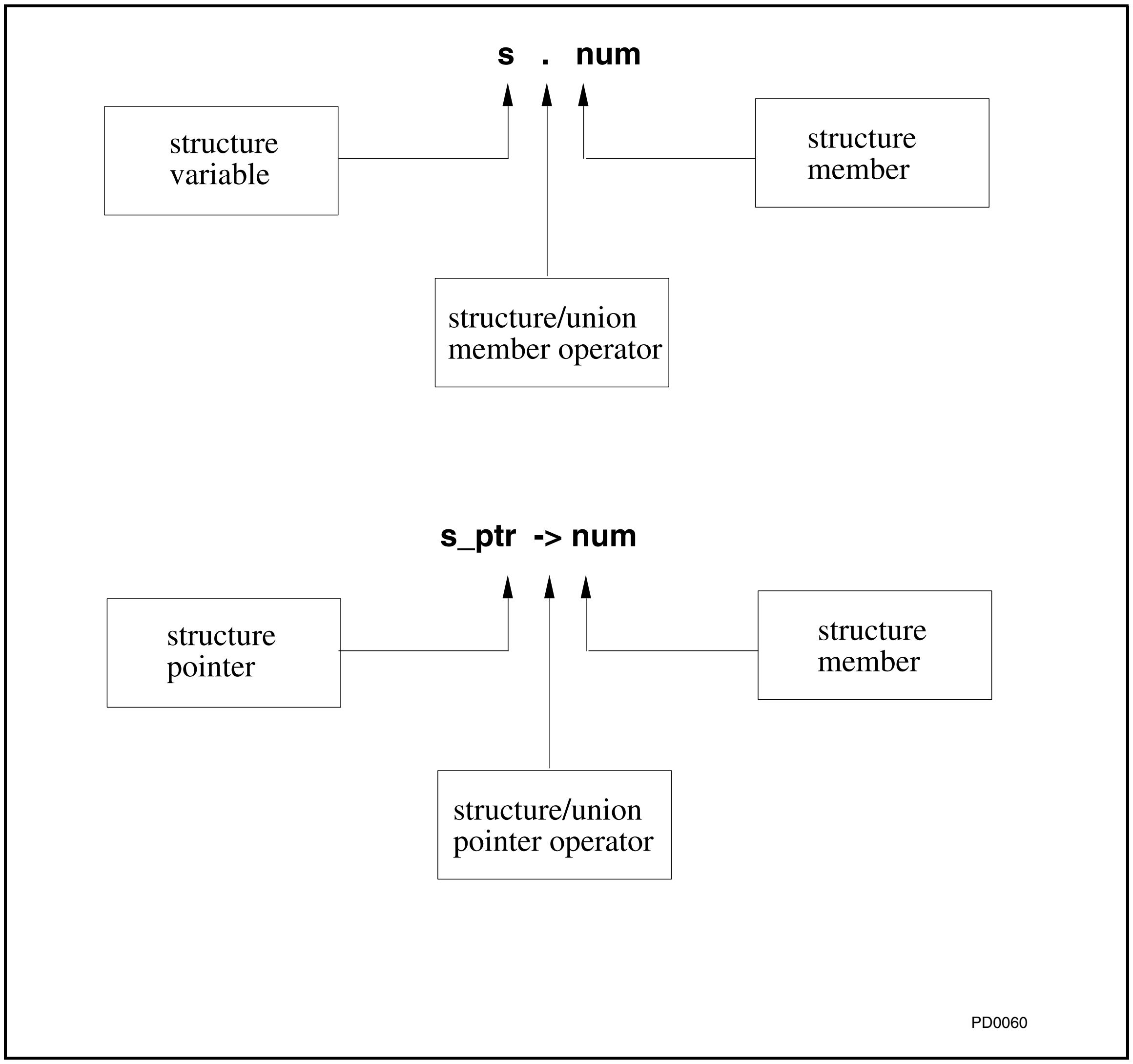


Figure 7-1. Member References

Partial Qualification of Member Names

In VOS C, when you reference a structure or union member, you can use the member name alone without preceding the member name with `expression` or `pointer` if the member name is unambiguous. VOS C allows this *partial qualification* when the expression unambiguously identifies the structure or union variable and the member.

Consider the following definitions:

```
struct
   {
   union
      {
      char alpha[4];
      int  numeric;
      } funds_id;
   int account_type;
   } account_id, *account_ptr = &account_id;
```

An example of a fully qualified reference to the `account_id.funds_id.numeric` member is as follows:

```
account_ptr->funds_id.numeric
```

Some examples of partially qualified references to the same member are as follows:

```
account_ptr->numeric

account_id.numeric

funds_id.numeric

numeric
```

If you partially qualify a reference to an array of structures and do not provide a subscript specifying an element of the array, the VOS C compiler assumes that the subscript is 0. Consider the following array of structures and partially qualified references.

```
struct
   {
   int i;
   char c;
   } s[5];

i = 999;            /*  Example 1  */

s.c = 'Z';          /*  Example 2  */
```

In both example 1 and example 2, the compiler assumes a subscript value of 0 for the array of structures `s`. Thus, for the two preceding partially qualified references, `i` and `s.c`, the compiler supplies the following fully qualified references in the assignment statements:

```
s[0].i = 999;

s[0].c = 'Z';
```

If you select at least the `minor` value for the `extension_checking` option in a `#pragma` preprocessor control line, or in the corresponding command-line argument, the compiler diagnoses a partially qualified reference to a structure or union member.

Common Initial Sequence of Union Members

Accessing a member of a union after a value has been stored in a different member, in general, yields invalid results. However, if a union contains several structures that contain a common initial sequence and one of these structures has been written to most recently, accessing the common initial part of any of them is guaranteed to yield correct results. In VOS C, two structures contain a *common initial sequence* if the corresponding members contain compatible types for a sequence of one or more initial members and if the structures have the same alignment.

In the following example, the `u` union contains two nested structures having a common initial sequence of two members.

```
union
   {
   struct
      {
      char letter;
      int code;
      double amount;
      } s1;

   struct
      {
      char letter;
      int code;
      long amount;
      } s2;
   } u;

u.s1.letter = 'C';
u.s1.code = 99;
u.s1.amount = 333.33;

printf("u.s2.letter = %c\n", u.s2.letter);
printf("u.s2.code = %d\n", u.s2.code);
printf("u.s2.amount = %d\n", u.s2.amount);
```

In the preceding example, even though the members of the nested structure `s1` were written to most recently, accessing the first two members of the nested structure `s2` in the `printf` calls yields correct results. The first two members of these two structures constitute a common initial sequence. The output from the preceding program is as follows:

```
u.s2.letter = C
u.s2.code = 99
u.s2.amount = 1081398599
```

As shown in the output, accessing the third member of the `s2` structure in the third `printf` call yields **incorrect** results because the `amount` member is not part of the common initial sequence.

Postincrement Operator

You can use the postincrement operator (`++`) to add the value 1 to an object **after** the value of the object is obtained. The syntax for an expression using the postincrement operator is as follows:

```
scalar_variable ++
```

In VOS C, *`scalar_variable`* is a modifiable lvalue that has a scalar type, but it cannot specify a bit field. The *`scalar_variable`* can be qualified with `volatile` but not with `const`.

An expression containing the postincrement operator yields the value of *`scalar_variable`* before the operand is incremented. The result has the type of the operand and is not a modifiable lvalue. After the expression is evaluated, the constant value 1 is added to *`scalar_variable`*, modifying the object's stored value. The following example shows how the postincrement operator works.

```
int num = 99;

printf("The expression num++ yields %d\n", num++);
printf("After the expression is evaluated, num = %d\n", num);
```

The output from the preceding program is as follows:

```
The expression num++ yields 99
After the expression is evaluated, num = 100
```

As shown in the preceding example, this postincrement operation has the side effect of assigning the value `num + 1` to the `num` variable.

You can use the postincrement operator with a pointer operand (for example, `pointer++`). In this case, after the value of the pointer is obtained, an address offset is added to the address stored in the pointer. See the "Incrementing and Decrementing Pointers" section in Chapter 4 for information on using the postincrement operator with a pointer.

See the "Addition Operator" section later in this chapter for information on the types that are allowed and the conversions that are performed with an addition operation.

Postdecrement Operator

You can use the postdecrement operator (`--`) to subtract the value 1 from an object **after** the value of the object is obtained. The syntax for an expression using the postdecrement operator is as follows:

```
scalar_variable --
```

In VOS C, *`scalar_variable`* is a modifiable lvalue that has a scalar type, but it cannot specify a bit field. The *`scalar_variable`* can be qualified with `volatile` but not with `const`.

An expression containing the postdecrement operator yields the value of *`scalar_variable`* before the operand is decremented. The result has the type of the operand and is not a modifiable lvalue. After the expression is evaluated, the constant value 1 is subtracted from *`scalar_variable`*, modifying the object's stored value. The following example shows how the postdecrement operator works.

```
int num = 99;

printf("The expression num-- yields %d\n", num--);
printf("After the expression is evaluated, num = %d\n", num);
```

The output from the preceding program is as follows:

```
The expression num-- yields 99
After the expression is evaluated, num = 98
```

As shown in the preceding example, this postdecrement operation has the side effect of assigning the value `num - 1` to the `num` variable.

You can use the postdecrement operator with a pointer operand (for example, `pointer--`). In this case, after the value of the pointer is obtained, an address offset is subtracted from the address stored in the pointer. See the "Incrementing and Decrementing Pointers" section in Chapter 4 for information on using the postdecrement operator with a pointer.

See the "Subtraction Operator" section later in this chapter for information on the types that are allowed and the conversions that are performed with a subtraction operation.

Unary Operators

A *unary operator* is an operator that takes one operand. Table 7-3 shows the unary operators.

Table 7-3. Unary Operators *(Page 1 of 2)*

Operator	Name	Example	Operation
`&`	Address-of	`&num`	Gets the address of an object or function
`*`	Indirection	`*num`	Dereferences a pointer
`~`	Bitwise complement	`~setting`	Forms the bitwise complement of an integral operand
`!`	Logical negation	`!okay`	Computes the logical negation of an expression
`-`	Unary minus	`-amount`	Negates the value of its operand
`()`	Cast	`(char)num`	Converts an operand value to a specified type
`sizeof`	Sizeof	`sizeof(int)`	Determines the size, in bytes, of its operand
`++`	Preincrement	`++num`	Increments its operand's value before the result is obtained

Table 7-3. Unary Operators *(Page 2 of 2)*

Operator	Name	Example	Operation
`--`	Predecrement	`--num`	Decrements its operand's value before the result is obtained

Address-of Operator

You use the address-of operator (`&`) to get the address of an object or function. The syntax for an expression using the address-of operator is as follows:

```
& operand
```

The *`operand`* can be a function designator or an lvalue that designates an object other than a bit field. The object cannot be declared with the `register` storage-class specifier. In VOS C, the operand of the address-of operator can be a constant value **if** the expression is part of an argument list in a function call. See the "Using Constant Operands with the Address-of Operator" section later in this chapter for information on using a constant value as an operand.

In VOS C, the operand of the `&` operator can be the `$substr` built-in function. See the "Using `$substr` Operands with the Address-of Operator" section later in this chapter for information on this use of the `&` operator.

An expression containing the `&` operator yields a pointer to the specified function or object. If the operand is of type *`type`*, the result has the type "pointer to *`type`*." When the operand is a `char_varying` string, the result is a pointer to the first byte allocated for the object. The first byte of a `char_varying` object is part of the 2-byte current length.

In pointer operations, the address-of operator and indirection operator are frequently used, respectively, to assign an address to a pointer and then to dereference the pointer. See the "Pointer Operations" section in Chapter 4 for information on and an example of how the address-of operator and indirection operator are used in pointer operations.

Using Constant Operands with the Address-of Operator

In VOS C, certain constant values can be used as the operand of the address-of operator **if** the expression is part of an argument list. This VOS C extension makes it possible to pass constant values to procedures, such as VOS operating system subroutines, that expect arguments to be passed by reference. By-reference argument passing is the standard method used by the VOS high-level languages other than C.

> **Note:** The constant values whose addresses are passed using the `&` operator are allocated storage only for the duration of the call and can be used only while the called procedure is active.

If a function prototype is visible, the address that is passed for the constant value is that of an object of the type given in the prototype. If a function prototype is not visible, the constant values that are allowed as the operand of the `&` operator include the following:

- If the operand is an integer constant, character constant, enumeration constant, or a constant expression yielding an integer value, the result is the address of a temporary `int` object containing the specified value.
- If the operand is a floating-point constant or a constant expression yielding a floating-point value, the result is the address of a temporary `double` object containing the specified value.
- If the operand is a cast expression, the result is the address of a temporary object containing the converted value. The type of the object is the type specified in the cast.

The compiler issues an error message when you try to use the `&` operator with a constant value if the resulting expression is not part of an argument list. In addition, the compiler issues an error message if you try to use the `&` operator with an argument that is an arbitrary expression. An arbitrary expression contains operands whose values are determined at run time.

The following examples show how you can use the `&` operator with a constant value operand when the resulting expression is part of an argument list in a call to an operating system subroutine, which is a PL/I procedure.

```
s$subroutine(&(1024 * 10), &(999.99), &(char_varying(32))"test_file");
```

In the preceding example, the three arguments use the address-of operator with an integer constant expression, a floating-point constant, and a cast expression.

Using `$substr` Operands with the Address-of Operator

When an expression containing the `$substr` built-in function is the operand of the `&` operator, the result is the address of a location within the string where the `$substr` operation will begin. When a `$substr` expression is used as the operand of the `&` operator, `$substr`'s first argument must be an lvalue. The following example shows an expression that uses the `$substr` built-in as the operand for the address-of operator. In this example, `$substr` has a `char` array argument.

```
char array[20], *c_ptr;
int i = 1, l = 5;

c_ptr = &($substr(array, i, l));
```

In the preceding example, the expression `&($substr(array, i, l))` yields the address of `array[0]` because arrays in C begin with element 0. Notice that, in the `$substr` expression, a starting location of `array[0]` is specified by setting the second argument, `i`, equal to 1. Index counting with `$substr` begins with 1. The third argument to the `$substr` built-in has no relevance to the address-of operation.

When the operand of the `&` operation is a `$substr` expression and when the first argument to `$substr` has the `char_varying` type, the result is the address of the starting location within the string where the `$substr` operation will begin. In a `char_varying` object, the string is preceded by a 2-byte current length field. Therefore, the first character in the string is offset two bytes beyond the beginning address of the `char_varying` object. The following example shows an expression that uses a `$substr` built-in as the operand of the `&` operator. In this example, `$substr` has a `char_varying` argument.

```
char *c_ptr;
char_varying(10) cv_string = "abcdefghij";

c_ptr = &($substr(cv_string, 1));
```

In the preceding example, the expression `&($substr(cv_string, 1))` yields the address of the first character (`a`) within the string part of `cv_string`, which is located at the address of `cv_string` plus two bytes. The two added bytes contain the 2-byte current length field.

See the "Varying-Length Character String Type" section in Chapter 4 for information on that data type.

Indirection Operator

You use the indirection operator (`*`) to dereference a pointer. *Dereferencing* a pointer means accessing the object to which the pointer points. The syntax for an expression using the indirection operator is as follows:

```
* pointer
```

The *`pointer`* is a pointer type. If *`pointer`* points to a function, an expression using the indirection operator yields an expression that has function type. If *`pointer`* points to an object, an expression using the `*` operator yields an lvalue designating the object. If *`pointer`* is of type "pointer to *`type`*," the result has type *`type`*.

If an invalid value has been assigned to the pointer prior to the indirection operation, one of the following occurs.

- If a pointer has been assigned a null pointer constant, the indirection operation often causes a run-time error to occur.
- If a pointer to one type has been assigned the address of an object of another type, the indirection operation may produce incorrect results.
- If the pointer has been assigned the address of an object with automatic storage duration, the indirection operation produces unpredictable results when execution of the block in which the object is declared has terminated.

The following program fragment illustrates this last point.

```
#include <stdio.h>

int num, *i_ptr;
void func(void);
```

(Continued on next page)

```
main()
{
   func();
       .
       .
       .
  num = *i_ptr;        /*  Indirection operation yields            */
}                      /*  unpredictable results                   */


void func(void)
{
  int i_num;           /*  Object with automatic storage duration   */

   i_ptr = &i_num;
  *i_ptr = 999;        /*  Using indirection, assigns 999 to i_num  */
}
```

In the preceding example, the results of the indirection operation `*i_ptr` in the `main` function are unpredictable because `i_ptr` was assigned the address of `i_num` and the block that declares `i_num` has terminated.

In pointer operations, the address-of operator and indirection operator are frequently used, respectively, to assign an address to a pointer and then to dereference the pointer. See the "Pointer Operations" section in Chapter 4 for information on and an example of how the address-of operator and indirection operator are used in pointer operations.

For information on how the indirection operator is used with function pointers, see the "Function Pointers" section in Chapter 6.

Sizeof Operator

You use the `sizeof` operator to determine the size, in bytes, of its operand. The operand of the `sizeof` operator can be either a type name or an expression that designates an object type other than a bit-field. The syntax for an expression using the `sizeof` operator with a parenthesized type name operand is as follows:

```
sizeof ( type_name )
```

The *`type_name`* is the name of a VOS C data type other than a function type, incomplete type, or the `void` type. An expression `sizeof (`*`type_name`*`)` yields the amount of memory, in bytes, that would be occupied by an object of the specified data type.

The syntax for an expression using the `sizeof` operator with an expression operand is as follows:

```
sizeof expression
```

The *`expression`* is an expression that designates an object type, but it cannot specify a bit field. An incomplete type or the `void` type is not an object type. An expression `sizeof` *`expression`* yields the amount of memory, in bytes, that is occupied by the specified object.

However, *`expression`* itself is not evaluated. If the *`expression`* operand contains operators that generate side effects, those side effects do not occur.

Whether the `sizeof` operator's operand is a type name or an expression, the result is an integer value and has the type `size_t`. The type definition `size_t` is declared in the `stddef.h` header file.

- When the operand has an array type, the result is the total number of bytes in the array.
- When the operand is a character-string literal, the result equals the number of characters enclosed within quotation marks plus 1 for the appended null character.
- When the `sizeof` operator's operand is a `$substr` expression, the result is equal to the length of the `$substr` expression's `char_varying` result.

When the operand of the `sizeof` operator has structure, union, or `char_varying` type, the result is the total number of bytes in the object, including internal and trailing alignment padding. For a `char_varying` object, the total number of bytes equals the 2-byte current length field plus the maximum length, not the current length, of the string. To get the current length of a `char_varying` string, use the `strlen_vstr` function.

When the operand of the `sizeof` operator is a parameter declared to have array or function type, the result is the size of the pointer (four bytes) because such an array name or function name is converted to a pointer when used in this context.

The following program illustrates the use of the `sizeof` operator with a variety of operands.

```
short array[40];

struct st
   {
   int i;
   float f;
   } s;

typedef short int S_NUM;

float amt;

main()
{
   printf("sizeof (char) = %d\n", sizeof (char) );            /*
Example 1  */

   printf("sizeof array = %d\n", sizeof array);               /*
Example 2  */

   printf("sizeof (struct st) = %d\n", sizeof (struct st) );  /*
```

(Continued on next page)

```
Example 3  */

   printf("sizeof (S_NUM) = %d\n", sizeof (S_NUM) );          /*
Example 4  */

   printf("sizeof (amt + 99.9) = %d\n", sizeof (amt + 99.9) );/*
Example 5  */
}
```

The output from the preceding program is as follows:

```
sizeof (char) = 1
sizeof array = 80
sizeof (struct st) = 8
sizeof (S_NUM) = 2
sizeof (amt + 99.9) = 8
```

In example 5, the expression `(amt + 99.9)` is not evaluated. In this case, the `sizeof` operation yields the size of the **type** that would result if the expression were evaluated. When a binary operator, such as the `+` operator in example 5, has an operand that is a `double`, the other operand is converted to type `double`, and the result is of type `double`. In example 5, since the constant 99.9 is considered a `double`, the result of the `sizeof` operation yields the size of a `double`: 8 bytes.

Cast Operator

You use the cast operator (`()`) to convert a value to a specified type. The syntax for an expression using the cast operator is as follows:

```
( type_name ) expression
```

The *`type_name`* can be the name of a qualified or unqualified scalar data type, the `char_varying` type, or the `void` type. The *`type_name`* can include an alignment specifier. A *`type_name`* enclosed in parentheses, as shown in the preceding syntax, is called a *cast*. The *`expression`*, which is the operand of the cast operator, must resolve to a scalar type or to the `char_varying` type.

An expression `(`*`type_name`*`)` *`expression`* converts the value of *`expression`* to the type specified in *`type_name`*. A cast does **not** yield an lvalue. A cast that specifies an implicit conversion or no conversion has no effect on the type or value of the operand expression.

The following examples show some valid uses of the cast operator.

```
int i_num;
char ch;
char_varying(30) cv_string = "123";

ch = (char)('A' + 32);       /*  int expression cast to char type
*/
printf("ch = %c\n", ch);      /*  ch = a                           */

i_num = (int)cv_string;         /*  char_varying expression cast to
int type  */
printf("i_num = %d\n", i_num); /*  i_num = 123
*/
```

In general, conversions that involve pointers must be specified by explicit cast. See the “Pointer Operations” section in Chapter 4 for information on using explicit casts in pointer conversions.

Unary Minus Operator

You use the unary minus operator (-) to negate a value. The syntax for an expression using the unary minus operator is as follows:

```
- arithmetic_expression
```

The *arithmetic_expression* is an expression having an arithmetic type.

The integral promotion is performed on the operand, and the result of the expression has the promoted type. An expression preceded by the - operator yields the negative of its operand’s value. The result is not an lvalue. The following examples show how the - operator is used.

```
int num1 = 1, num2;
float f_num;

num2 = -num1;                 /*  num2 = -1    */

f_num = -(9.9 * num2);        /*  f_num = 9.9  */
```

Bitwise Complement Operator

You use the bitwise complement operator (~) to form the bitwise complement of an integral expression. The ~ operator is sometimes called the one’s complement operator. The syntax for an expression using the bitwise complement operator is as follows:

```
~ integral_expression
```

The *integral_expression* is an expression having an integral type.

The integral promotion is performed on the operand, and the result of the expression has the promoted type. An expression preceded by the ~ operator yields the bitwise complement of the expression. The result is not an lvalue.

The bitwise complement operator inverts the bit pattern of its operand. That is, all operand bits that are equal to 1 before the operation are set to 0, and all bits that are equal to 0 before the operation are set to 1. The following example shows how the ~ operator works. A `char` variable named `num` has the bit pattern:

```
11110000
```

The expression `~num` converts the least significant eight bits of the resulting value to the bit pattern:

```
00001111
```

The complete resulting value is an `int` equal to `FFFFFF0F` hexadecimal because the bitwise complement operation promotes the value of its operand to an `int` before taking the one's complement of the value.

Logical Negation Operator

You use the logical negation operator (`!`) to compute the logical negation of an expression. The syntax for an expression using the logical negation operator is as follows:

```
! scalar_expression
```

The *`scalar_expression`* is an expression having a scalar type. The scalar types include the arithmetic and pointer types.

An expression preceded by the `!` operator yields the value 0 if the operand is nonzero, or yields the value 1 if the operand is 0. The result of the expression has the `int` type and is not an lvalue.

The following example uses the `!` operator in an `if` statement's controlling expression.

```
int okay = 1;
   .
   .
   .
if (!okay)
   break;
```

In the preceding example, the controlling expression `(!okay)` is identical to `(okay == 0)`.

Preincrement Operator

You can use the preincrement operator (++) to add the value 1 to an object **before** the value of the object is obtained. The syntax for an expression using the preincrement operator is as follows:

```
++ scalar_variable
```

In VOS C, *`scalar_variable`* is a modifiable lvalue that has a scalar type, but it cannot specify a bit field. The *`scalar_variable`* can be qualified with `volatile` but not with `const`.

An expression containing the preincrement operator yields the value of *`scalar_variable`* after the operand is incremented. The result has the type of the operand and is not an lvalue. Before the expression is evaluated, the constant value 1 is added to *`scalar_variable`*, modifying the object's stored value. The following example shows how the preincrement operator works.

```
int num = 99;

printf("The expression ++num yields %d\n", ++num);
printf("After the expression is evaluated, num = %d\n", num);
```

The output from the preceding program is as follows:

```
The expression ++num yields 100
After the expression is evaluated, num = 100
```

As shown in the preceding example, this preincrement operation has the side effect of assigning the value `num + 1` to the `num` variable.

You can use the preincrement operator with a pointer operand (for example, `++pointer`). In this case, before the value of the pointer is obtained, an address offset is added to the address stored in the pointer. See the "Incrementing and Decrementing Pointers" section in Chapter 4 for information on using the preincrement operator with a pointer.

See the "Addition Operator" section later in this chapter for information on the types that are allowed and the conversions that are performed with an addition operation.

Predecrement Operator

You can use the predecrement operator (`--`) to subtract the value 1 from an object **before** the value of the object is obtained. The syntax for an expression using the predecrement operator is as follows:

```
-- scalar_variable
```

In VOS C, *`scalar_variable`* is a modifiable lvalue that has a scalar type, but it cannot specify a bit field. The *`scalar_variable`* can be qualified with `volatile` but not with `const`.

An expression containing the predecrement operator yields the value of *`scalar_variable`* after the operand is decremented. The result has the type of the operand and is not an lvalue. Before the expression is evaluated, the constant value 1 is subtracted from *`scalar_variable`*, modifying the object's stored value. The following example shows how the predecrement operator works.

```
int num = 1;

printf("The expression --num yields %d\n", --num);
printf("After the expression is evaluated, num = %d\n", num);
```

The output from the preceding program is as follows:

```
The expression --num yields 0
After the expression is evaluated, num = 0
```

As shown in the preceding example, this predecrement operation has the side effect of assigning the value `num - 1` to the `num` variable.

You can use the predecrement operator with a pointer operand (for example, `--pointer`). In this case, before the value of the pointer is obtained, an address offset is subtracted from the address stored in the pointer. See the “Incrementing and Decrementing Pointers” section in Chapter 4 for information on using the predecrement operator with a pointer.

See the “Subtraction Operator” section later in this chapter for information on the types that are allowed and the conversions that are performed with a subtraction operation.

Binary Operators

A *binary operator* is an operator that takes two operands. Table 7-4 shows the binary operators other than the assignment and comma operators, which are discussed later in this chapter.

Table 7-4. Binary Operators *(Page 1 of 2)*

Operator	Name	Example	Operation
`*`	Multiplication	`a * b`	Multiplies `a` by `b`
`/`	Division	`a / b`	Divides `a` by `b`
`%`	Remainder	`a % b`	Computes the remainder when `a` is divided by `b`
`+`	Addition	`a + b`	Adds `a` and `b`
`-`	Subtraction	`a - b`	Subtracts `b` from `a`
`<<`	Left shift	`a << b`	Shifts `a` to the left by `b` bit positions
`>>`	Right shift	`a >> b`	Shifts `a` to the right by `b` bit positions
`<`	Less than	`a < b`	Tests whether `a` is less than `b`
`<=`	Less than or equal to	`a <= b`	Tests whether `a` is less than or equal to `b`
`>`	Greater than	`a > b`	Tests whether `a` is greater than `b`
`>=`	Greater than or equal to	`a >= b`	Tests whether `a` is greater than or equal to `b`
`==`	Equality	`a == b`	Tests whether `a` equals `b`
`!=`	Inequality	`a != b`	Tests whether `a` does not equal `b`
`&`	Bitwise AND	`a &  b`	Forms the bitwise AND of `a` and `b`
`^`	Bitwise exclusive OR	`a ^ b`	Forms the bitwise exclusive OR of `a` and `b`

Table 7-4. Binary Operators *(Page 2 of 2)*

Operator	Name	Example	Operation
`|`	Bitwise inclusive OR	`a | b`	Forms the bitwise inclusive OR of `a` and `b`
`&&`	Logical AND	`a && b`	Tests whether both `a` and `b` are nonzero
`||`	Logical OR	`a || b`	Tests whether either `a` or `b` is nonzero

With many of the binary operators, the usual arithmetic conversions are performed on both operands before the operation. See the "Usual Arithmetic Conversions" section in Chapter 5 for more information on those conversions.

Multiplicative Operators

Three binary operators are multiplicative operators:

- the multiplication operator (`*`)
- the division operator (`/`)
- the remainder operator (`%`).

Multiplication Operator

You use the multiplication operator (`*`) to multiply one value by another value. The syntax for an expression using the multiplication operator is as follows:

```
arithmetic_expression1 * arithmetic_expression2
```

Both *`arithmetic_expression1`* and *`arithmetic_expression2`* are expressions having arithmetic types.

The usual arithmetic conversions are performed on the operands before the result is calculated. An expression containing the multiplication operator yields the product of the operands. The type of the result has the type of the converted operands and is not an lvalue. The following example shows how the `*` operator works.

```
int i_num = 9;
double d_num = 1.1;

printf("i_num * d_num = %.2lf\n", (i_num * d_num) );
```

The output from the preceding program is as follows:

```
i_num * d_num = 9.90
```

Since one of the operands, `d_num`, is a `double`, the usual arithmetic conversions specify that the value of the other operand, `i_num`, be converted to a `double` before the operation, and the result of the multiplication has the `double` type.

Division Operator

You use the division operator (`/`) to divide one value by another value. The syntax for an expression using the division operator is as follows:

```
arithmetic_expression1 / arithmetic_expression2
```

Both *`arithmetic_expression1`* and *`arithmetic_expression2`* are expressions having arithmetic types.

The usual arithmetic conversions are performed on the operands before the result is calculated. An expression containing the division operator yields the quotient from the division of *`arithmetic_expression1`* by *`arithmetic_expression2`*. The type of the result has the type of the converted operands and is not an lvalue.

Division by 0 creates a run-time error condition. See the "Default Signal Handlers" section in Chapter 11 for information on how a zero-divide error is handled. When positive or negative integers are divided and the result is not a whole number, truncation is toward 0. For example, the expression `11 / 4` equals 2, and `11 / -4` equals -2.

The following example shows how the `/` operator works.

```
float f_num = 9.9;
double d_num = 1.2;

printf("f_num / d_num = %.2lf\n", (f_num / d_num) );
```

The output from the preceding program is as follows:

```
f_num / d_num = 8.25
```

Since one of the operands, `d_num`, is a `double`, the usual arithmetic conversions specify that the value of the other operand, `f_num`, be converted to a `double` before the operation, and the result of the division has the `double` type.

Remainder Operator

You use the remainder operator (`%`) to compute the remainder when one value is divided by another value. The remainder operator is sometimes called the modulus operator. The syntax for an expression using the remainder operator is as follows:

```
integral_expression1 % integral_expression2
```

Both *`integral_expression1`* and *`integral_expression2`* are expressions having integral types.

The usual arithmetic conversions are performed on the operands before the result is calculated. An expression containing the remainder operator yields the remainder from the division of *`integral_expression1`* by *`integral_expression2`*. The type of the result has the type of the converted operands and is not an lvalue.

Division by 0 creates a run-time error condition. See the "Default Signal Handlers" section in Chapter 11 for information on how a zero-divide error is handled. The result of the remainder operation always has the same sign as *`integral_expression1`*.

The following example shows how the `%` operator works.

```
short s_num = 11;
int i_num = -4;

printf("s_num %% i_num = %d\n", (s_num % i_num) );    /*  The %%
generates %  */
```

The output from the preceding program is as follows:

```
s_num % i_num = 3
```

The usual arithmetic conversions specify that, since `s_num` is a `short`, its value undergoes the integral promotion to `int`, and the result of the `%` operation has the `int` type.

Additive Operators

Two binary operators are additive operators:

- the addition operator (`+`)
- the subtraction operator (`-`).

Addition Operator

You use the addition operator (`+`) to perform one of the following operations:

- add one arithmetic value to another arithmetic value
- add an integer value to a pointer value.

In VOS C, the `+` operator can also be used to append one `char_varying` string to another `char_varying` string. See the "Concatenation with the `char_varying` Type" section in Chapter 4 for information on this use of the `+` operator.

The syntax for an expression using the addition operator to add one arithmetic value to another is as follows:

```
arithmetic_expression1 + arithmetic_expression2
```

Both *`arithmetic_expression1`* and *`arithmetic_expression2`* are expressions having arithmetic types.

The usual arithmetic conversions are performed on both operands before the result is calculated. An expression containing the addition operator and two arithmetic expressions yields the sum of *`arithmetic_expression1`* and *`arithmetic_expression2`*. The type of the result has the type of the converted operands and is not an lvalue. The following example shows how the `+` operator works.

```
char c = 65;
short s_num = 32;

printf("c + s_num = %d\n", (c + s_num) );
```

The output from the preceding program is as follows:

```
c + s_num = 97
```

The usual arithmetic conversions specify that, since `c` is a `char` and `s_num` is a `short`, both values undergo the integral promotion to `int`, and the result of the addition has the `int` type.

As an alternative to the array-subscript operator, the `+` operator can be used to calculate an address offset within an array. The resulting pointer expression can then be dereferenced to access the contents of the object at that address. In pointer arithmetic operations, the syntax for an expression using the addition operator can be either of the following:

```
pointer_expression + integral_expression
```

or

```
integral_expression + pointer_expression
```

When you use the `+` operator to add an integer value to a pointer value, the *`pointer_expression`* is a pointer to an object type, usually an array element or one past the last array element. The *`integral_expression`* is an expression yielding an integer value and has an integral type. Such a pointer arithmetic expression yields a pointer that is an address at a specified offset after the address indicated by *`pointer_expression`*. The result has the type of the *`pointer_expression`*.

When *`integral_expression`* is added to the *`pointer_expression`*, the compiler calculates an address offset as follows:

```
integral_expression * sizeof(pointed_to_object)
```

The compiler multiplies the value of *`integral_expression`* by a scale factor. A *scale factor* is a value equal to the size (in bytes) of the pointed-to object. The compiler uses scale factors to take into account the size of the pointed-to object. To compute the result of the addition operation, the address offset is added to the address specified in *`pointer_expression`*. If *`pointer_expression`* points to an array member and the array is large enough, the result points to another member of the same array object, appropriately offset from the original member.

Note: Careless use of pointer arithmetic yields incorrect results. You can, for instance, erroneously reference a location beyond the limits of an array without generating a compile-time or run-time error message.

See the "Pointer Arithmetic" section in Chapter 4 for more information on and examples of how to use the addition operator in pointer arithmetic.

Subtraction Operator

You use the subtraction operator (`-`) to perform one of the following operations:

- subtract one arithmetic value from another arithmetic value
- subtract an integer value from a pointer value
- subtract one pointer value from another pointer value.

The syntax for an expression using the subtraction operator to subtract one arithmetic value from another is as follows:

```
arithmetic_expression1 - arithmetic_expression2
```

Both `arithmetic_expression1` and `arithmetic_expression2` are expressions having arithmetic types.

The usual arithmetic conversions are performed on both operands before the result is calculated. An expression containing the subtraction operator and two arithmetic expressions yields the difference resulting from the subtraction of `arithmetic_expression2` from `arithmetic_expression1`. The type of the result has the type of the converted operands and is not an lvalue. The following example shows how the `-` operator works.

```
char c = 97;
short s_num = 32;

printf("c - s_num = %d\n", (c - s_num) );
```

The output from the preceding program is as follows:

```
c - s_num = 65
```

The usual arithmetic conversions specify that, since `c` is a `char` and `s_num` is a `short`, both values undergo the integral promotion to `int`, and the result of the subtraction has the `int` type.

As an alternative to the array-subscript operator, the `-` operator can be used to calculate an address offset within an array. The resulting pointer expression can then be dereferenced to access the contents of the object at that address. In pointer arithmetic operations, the syntax for an expression using the subtraction operator is as follows:

```
pointer_expression - integral_expression
```

When you use the `-` operator to subtract an integer value from a pointer value, the `pointer_expression` is a pointer to an object type, usually an array element or one past the last array element. The `integral_expression` is an expression yielding an integer value and has an integral type. Such a pointer arithmetic expression yields a pointer that is an address at a specified offset before the address indicated by `pointer_expression`. The result has the type of the `pointer_expression`.

Note: Careless use of pointer arithmetic yields incorrect results. You can, for instance, erroneously reference a location beyond the limits of an array without generating a compile-time or run-time error message.

The preceding section, “Addition Operator,” explains how the compiler calculates an address offset. To compute the result of the subtraction operation, the address offset is subtracted from the address specified in `pointer_expression`. If `pointer_expression` points to an array member and the array is large enough, the result points to another member of the same array object, appropriately offset from the original member.

The – operator can also be used to subtract one pointer value from another pointer value when both pointers point to elements of the same array. The syntax for an expression using the subtraction operator to subtract one pointer value from another is as follows:

```
pointer_expression1 - pointer_expression2
```

Both `pointer_expression1` and `pointer_expression2` are pointers to qualified or unqualified versions of compatible object types, usually elements of the same array. Each pointer points to an array element or one past the last array element. Such an expression yields the difference (in bytes) between the two pointers divided by the size of the pointed-to object. Since both pointers point to elements of the same array, the result is the number of elements between the two pointers. The result has the type `ptrdiff_t`, which is a type definition declared in the `stddef.h` header file.

See the "Pointer Arithmetic" section in Chapter 4 for more information on and examples of how to use the subtraction operator in pointer arithmetic.

Bitwise-Shift Operators

You use the bitwise-shift operators to shift a bit value to the left or right a specified number of bit positions. Two binary operators are bitwise-shift operators:

- left-shift operator (<<)
- right-shift operator (>>).

The syntax for expressions using the bitwise-shift operators is as follows:

```
integral_expression1 << integral_expression2

integral_expression1 >> integral_expression2
```

The right operand, `integral_expression2`, is a non-negative value specifying the number of bit positions that the left operand, `integral_expression1`, will be shifted. Both `integral_expression1` and `integral_expression2` have integral types. With both the << and >> operators, if the value of the right operand is negative or is greater than 31, the result of the operation is unpredictable.

With both bitwise-shift operators, the integral promotions are performed on both operands before the shift occurs. The bitwise-shift operators yield the value of the left operand after it has been shifted in one of the following ways:

- left-shifted the number of bit positions specified in the right operand if << is the operator

- right-shifted the number of bit positions specified in the right operand if >> is the operator.

The type of the result is that of the promoted left operand and is not an lvalue. See the "Conversions between Integral Types" section in Chapter 5 for information on what takes place when an integral type is promoted to a longer integral type.

The following example shows how the << operator works. A `signed char` variable named `s_c` holds the value 24 and has the bit pattern:

```
00011000
```

The expression `s_c << 3` yields the following bit pattern after the integral promotion converts `s_c` to an `int`.

```
00000000 00000000 00000000 11000000
```

In a left-shift operation such as the preceding one, vacated bits are filled with zeroes. Bits shifted off to the left are discarded.

The following example shows how the >> operator works. A `signed char` variable named `s_c` holds the value -11 and has the bit pattern:

```
 11110101
```

The expression `s_c >> 2` yields the following bit pattern after integral promotion converts `s_c` to an `int`.

```
11111111 11111111 11111111 11111101
```

If the left operand of the >> operator is signed as in the preceding operation, vacated bits are filled with copies of the sign bit. If the left operand is unsigned, vacated bits are filled with zeroes. In both cases, bits shifted off to the right are discarded.

Relational Operators

You use the relational operators to test which of two operands is greater. Four binary operators are relational operators:

- less-than operator (<)
- less-than-or-equal-to operator (<=)
- greater-than operator (>)
- greater-than-or-equal-to operator (>=).

The syntax for expressions using the relational operators is as follows:

```
operand1 < operand2

operand1 <= operand2

operand1 > operand2

operand1 >= operand2
```

For *`operand1`* and *`operand2`*, the following combinations of operands are allowed.

- Both operands can be arithmetic types.

- Both operands can be pointers to qualified or unqualified versions of compatible object types.

- Both operands can be pointers to qualified or unqualified versions of compatible incomplete types.

- In VOS C, if one operand has the `char_varying` type, the other operand must have the `char_varying` type or must be convertible to `char_varying`.

In `char_varying` comparisons, one or both operands can be a `$substr` built-in function. These `$substr` operands are useful when comparing various types of data, such as structures. See the "`$substr`" section in Chapter 12 for information on the use of `$substr` operands with the relational operators.

Each of the relational operators yields the value 1 if the specified relation is true or the value 0 if the relation is false. The result has the `int` type. The following paragraphs discuss how various types of operands are compared. The next section, "Equality Operators," contains additional information on how certain structure and pointer operands are tested for equality.

If the operands have arithmetic types, the usual arithmetic conversions are performed on both operands before the expression is evaluated. The operand with the higher arithmetic value compares greater.

If the operands are pointers, the comparison depends on the relative locations in the address space of the pointed-to objects. The operand pointing to a higher address compares greater. When the operands are pointers, the following rules determine the result of the comparison.

- When the pointed-to objects are members of the same structure, a pointer to a member declared later compares greater than a pointer to a member declared earlier in the structure.

- When the pointed-to objects are elements of the same array, a pointer to an element with a higher subscript value compares greater than a pointer to an element with a lower subscript value. In this type of comparison, it is valid for one of the pointer operands to point to one past the last element of the array.

- When the pointed-to objects are members of the same union, all pointers to members of the same union compare equal.

When the pointed-to objects are not from the preceding categories of operands, the results of a pointer comparison are **unpredictable**.

If one of the operands has the `char_varying` type, the other operand must have the `char_varying` type or be convertible to `char_varying`. When two `char_varying` types are the operands in a relational operation, the two strings are compared, character by character, until two characters are found that are different. The string whose differing character has the higher ASCII code compares greater. In the case of unequal length strings, the shorter string is padded with spaces to the length of the longer operand.

> **Note:** This method of comparing two `char_varying` strings differs from the method used by the `strcmp_vstr_vstr` library function. When `strcmp_vstr_vstr` is used with two `char_varying` strings, the longer string always compares greater than the shorter if both strings contain the same characters up to the length of the shorter string.

The following example shows how the `<=` operator can be used in the controlling expression of a `for` statement. In the controlling expression `i <= num`, two operands having arithmetic types are compared.

```
int i;
   short num = 24;

   for (i = 0; i <= num; i++)
      .
      .
      .
```

The next example uses the `<` operator as the controlling expression in a `while` statement. In the controlling expression `(ptr1 < ptr2)`, two pointers are compared.

```
int array[10];

   int *ptr1 = &array[0];
   int *ptr2 = ptr1 + 10;

   while (ptr1 < ptr2)
      {
         .
         .
         .
      ptr1++;
      }
```

In the preceding example, the `while` statement continues to execute as long as `ptr1` points to an element of `array` that has a lower subscript value than the element pointed to by `ptr2`.

Equality Operators

You use the equality operator (`==`) to test whether two operands are equal. You use the inequality operator (`!=`) to test whether two operands are not equal.

Note: Unlike the equality operator in some other high-level languages, such as Pascal, the C equality operator consists of two equals signs. Using the assignment operator (`=`) in a controlling expression, where what is desired is a test for equality, is a common programming error and usually yields unexpected and unwanted results.

The syntax for expressions using the equality operators is as follows:

`operand1` `==` *`operand2`*

`operand1` `!=` *`operand2`*

For `operand1` and `operand2`, the following combinations of operands are allowed.

- Both operands can be arithmetic types.
- Both operands can be pointers to qualified or unqualified versions of compatible types.
- One operand can be a pointer to an object or incomplete type, and the other operand can be a pointer to a qualified or unqualified version of `void`.
- One operand can be a pointer, and the other operand can be a null pointer constant.
- In VOS C, if one operand has the `char_varying` type, the other operand must have the `char_varying` type or must be convertible to `char_varying`.

In VOS C, there are two null pointer constants, `NULL` and `OS_NULL_PTR`. See the "Null Pointer Constants" section in Chapter 4 for information on null pointer constants.

When one of the operands is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of `void`, the pointer to an object or incomplete type is converted to the type of the other operand.

The equality operator tests for equality, yielding the value 1 if the two operands are equal and the value 0 if the two operands are not equal. The inequality operator tests for inequality, yielding the value 1 if the two operands are not equal and the value 0 if the two operands are equal.

With the `==` and `!=` operators, the result has the `int` type and is not an lvalue. The following paragraphs discuss how certain types of pointer and structure operands are tested with the `==` and `!=` operators. In all other respects, the equality and inequality operators work analogously to the relational operators. See the previous section, "Relational Operators," for additional information on how operands are compared.

With the `==` and `!=` operators, the following rules apply.

- If two pointers to objects or incomplete types compare equal, they point to the same object.
- If two pointers to functions compare equal, they point to the same function.

In VOS C, two pointers to functions are equal if one of the following conditions is true.

- Both contain `NULL` or `OS_NULL_PTR`.
- Both contain `SIG_IGN`, `SIG_DFL`, or `SIG_ERR`.
- Both locate the same function.

Two pointers to functions locate the same function if each points to an entry value that contains identical values for the display pointer and code pointer elements of the entry-value array. In a multitasking program, the static pointer, which can be different for the same entry executing in different tasks, does not need to be the same.

The constants `SIG_IGN`, `SIG_DFL`, and `SIG_ERR` are defined in the `signal.h` header file and are used with the `signal` function. See the description of the `signal` function in Chapter 11 for more information on these defined constants.

The following example shows how two pointers to objects can be tested for equality to determine whether they point to the same object.

```
int num, *i_ptr1, *i_ptr2;

i_ptr1 = &num;
i_ptr2 = &num;

if (i_ptr1 == i_ptr2)
   {
   puts("Both pointers contain the same address.");
   puts("Both pointers point to the same object.");
   }
```

In the preceding example, the controlling expression `(i_ptr1 == i_ptr2)` yields the value 1.

Bitwise Operators

You use the bitwise operators to manipulate the bits of an operand. Three binary operators are bitwise operators:

- bitwise AND operator (`&`)
- bitwise exclusive-OR operator (`^`)
- bitwise inclusive-OR operator (`|`).

Bitwise AND Operator

You use the bitwise AND operator (`&`) to form the bitwise AND of two operands. The syntax for an expression using the bitwise AND operator is as follows:

```
integral_expression1 & integral_expression2
```

Both *`integral_expression1`* and *`integral_expression2`* are expressions having integral types. The usual arithmetic conversions are performed on the operands before the expression is evaluated. The result has the type of the converted operands and is not an lvalue.

When two values are ANDed, for each bit position, the resulting bit is 1 only if each of the corresponding bits in *`integral_expression1`* **and** *`integral_expression2`* is set to 1. The resulting bit is 0 for any other combination of bit values. If `bit1` and `bit2` are corresponding bits in the two operands, Table 7-5, sometimes called the AND truth table, shows all possible results when `bit1` is ANDed with `bit2`.

Table 7-5. AND Truth Table

bit1	bit2	bit1 & bit2
0	0	0
0	1	0
1	0	0
1	1	1

In the following example, the bitwise AND operator is used with two `short` operands, `value1` and `value2`. The bit patterns for each value and the result of the bitwise AND operation are shown in the comments.

```
int result;

short value1 = 0xF6F6;                    /*  value1 = 11110110
11110110  */

short value2 = 0x0003;                    /*  value2 = 00000000
00000011  */

result = value1 & value2;  /*  result = 0000000000000000 00000000
00000010  */
```

The usual arithmetic conversions specify that the values of both operands are promoted to `int` before the operation. The result has the `int` type and is 32 bits long. Because `value2` has only bits 0 and 1 set equal to 1, the bitwise AND operation clears to 0 the least significant 16 bits except for bits 0 and 1. Bits 0 and 1 in `result` keep the same value they had in `value1`. Therefore, in `result`, bit 0 is 0, and bit 1 is 1.

Bitwise Exclusive-OR Operator

You use the bitwise exclusive-OR operator (`^`) to form the bitwise exclusive-OR of two operands. This operation is sometimes called the XOR operation. The syntax for an expression using the `^` operator is as follows:

```
integral_expression1 ^ integral_expression2
```

Both *`integral_expression1`* and *`integral_expression2`* are expressions having integral types. The usual arithmetic conversions are performed on the operands before the expression is evaluated. The result has the type of the converted operands and is not an lvalue.

When two values are XORed, for each bit position, the resulting bit is 1 if either (but not both) of the corresponding bits in *`integral_expression1`* **or** *`integral_expression2`* is set to 1. The resulting bit is 0 for any other combination of bit values. If `bit1` and `bit2` are corresponding bits in the two operands, Table 7-6, sometimes called the exclusive-OR truth table, shows all possible results when `bit1` is XORed with `bit2`.

Table 7-6. Exclusive-OR Truth Table

`bit1`	`bit2`	`bit1 ^ bit2`
0	0	0
0	1	1
1	0	1
1	1	0

In the following example, the ^ operator is used with two `short` operands, `value1` and `value2`. The bit patterns for each value and the result of the bitwise exclusive-OR operation are shown in the comments.

```
int result;

short value1 = 0xF0F0;                      /*  value1 = 00001111
00001111  */

short value2 = 0xFFFF;                      /*  value2 = 11111111
11111111  */

result = value1 ^ value2;  /*  result = 1111111111111111 11110000
11110000  */
```

The usual arithmetic conversions specify that the values of both operands are promoted to `int` before the operation. The result has the `int` type and is 32 bits long. Because `value2` has all bits set equal to 1, the bitwise exclusive-OR operation clears to 0 all bits in `result` except for the bits that were equal to 0 in `value1` after `value1` was promoted to `int`.

Bitwise Inclusive-OR Operator

You use the bitwise inclusive-OR operator (|) to form the bitwise inclusive-OR of two operands. The syntax for an expression using the | operator is as follows:

`integral_expression1` | *`integral_expression2`*

Both *`integral_expression1`* and *`integral_expression2`* are expressions having integral types. The usual arithmetic conversions are performed on the operands before the expression is evaluated. The result has the type of the converted operands and is not an lvalue.

When two values are inclusive-ORed, for each bit position, the resulting bit is 1 if either or both of the corresponding bits in *`integral_expression1`* and *`integral_expression2`* are set to 1. The resulting bit is 0 only when both bit values are 0. If `bit1` and `bit2` are corresponding bits in the two operands, Table 7-7, sometimes called the inclusive-OR truth table, shows all possible results when `bit1` is inclusive-ORed with `bit2`.

Table 7-7. Inclusive-OR Truth Table

`bit1`	`bit2`	`bit1|bit2`
0	0	0
0	1	1
1	0	1
1	1	1

In the following example, the | operator is used with two `short` operands, `value1` and `value2`. The bit patterns for each value and the result of the bitwise inclusive-OR operation are shown in the comments.

```
int result;

short value1 = 0x55FD;                /*  value1 = 01010101 11111101  */

short value2 = 0x0002;                /*  value2 = 00000000 00000010  */

result = value1 | value2;  /*  result = 0000000000000000 10101010 11111111
*/
```

The usual arithmetic conversions specify that the values of both operands are promoted to `int` before the operation. The result has the `int` type and is 32 bits long. Because `value2` has only bit 1 set to 1, after the bitwise inclusive-OR operation, bit 1 in `result` is set to 1, but all other bits in `result` keep the value that they had in `value1`.

Logical Operators

Two binary operators are logical operators:

- logical AND operator (`&&`)
- logical OR operator (||).

Logical AND Operator

You use the logical AND operator (`&&`) to test whether two operands are nonzero. The `&&` operator is typically used when you want to test that two conditions are true in a controlling expression, such as the controlling expression in a `while` statement. The syntax for an expression using the `&&` operator is as follows:

```
    scalar_expression1 && scalar_expression2
```

Both *`scalar_expression1`* and *`scalar_expression2`* are scalar types.

The `&&` operator yields the value 1 if both of its operands do not compare equal to 0, or it yields the value 0 if either operand compares equal to 0. The result has the `int` type and is not an lvalue. The logical AND operator always evaluates its operands from left to right. Thus, the left operand is evaluated first.

- If the left operand is 0, the right operand is **never** evaluated. In this case, the resulting expression yields the value 0.

- If the left operand is nonzero, the right operand is evaluated.
  - If the right operand is 0, the resulting expression yields the value 0.
  - If the right operand is nonzero, the resulting expression yields the value 1.

Because the operands are evaluated from left to right, if you use a condition that is expensive in terms of processor time as the rightmost operand of the `&&` operator, the program may be more efficient.

The following example shows how the `&&` operator works.

```
while ( (c != ' ') && (c != '\t') && (c != '\n') )
   {
   .
   .
   .
   }
```

In the preceding example, as long as `c` is not equal to a space character, horizontal tab, or newline character, the `while` statement continues to execute.

Logical OR Operator

You use the logical OR operator (||) to test whether either or both of two operands are nonzero. The || operator is typically used when you want to test that at least one of two conditions is true in a controlling expression, such as the controlling expression in an `if` statement. The syntax for an expression using the || operator is as follows:

`scalar_expression1` || *`scalar_expression2`*

Both *`scalar_expression1`* and *`scalar_expression2`* are scalar types.

The || operator yields the value 1 if either of its operands does not compare equal to 0, or it yields the value 0 if either operand compares equal to 0. The result has the `int` type and is not an lvalue. The logical OR operator always evaluates its operands from left to right. Thus, the left operand is evaluated first.

- If the left operand is nonzero, the right operand is **never** evaluated. In this case, the resulting expression yields the value 1.
- If the left operand is 0, the right operand is evaluated.
  - If the right operand is nonzero, the resulting expression yields the value 1.
  - If the right operand is 0, the resulting expression yields the value 0.

Because the operands are evaluated from left to right, if you use a condition that is expensive in terms of processor time as the rightmost operand of the || operator, the program may be more efficient.

The following example shows how the || operator works.

```
if ( (c == ' ') || (c == '\t') || (c == '\n') )
   {
   .
   .
   .
   }
```

In the preceding example, the compound statement of the `if` statement executes when `c` is equal to a space character, horizontal tab, or newline character.

Conditional Operator

You use the conditional operator (`?    :`) to evaluate one of two operands depending on the value of a third operand, and to get the value of the evaluated operand. An expression containing the conditional operator is called a *conditional expression*. The conditional operator is the only C operator that takes three operands. The syntax for an expression using the conditional operator is as follows:

```
operand1 ? operand2 : operand3
```

The *`operand1`* must be an expression that evaluates to a scalar type. Both *`operand2`* and *`operand3`* can have various types (see the discussion and examples later in this section for information on the types allowed).

In a conditional expression, *`operand1`* is always evaluated. Depending on the value of *`operand1`*, one of the following occurs.

- If *`operand1`* is nonzero, *`operand2`* is evaluated. The result has the value of *`operand2`*.
- If *`operand1`* is 0, *`operand3`* is evaluated. The result has the value of *`operand3`*.

A conditional expression does not yield an lvalue. The following example shows how the conditional operator works.

```
int max, y = 100, z = 1000;

max = (y > z) ? y : z;
```

In the preceding example, the conditional expression yields the value 1000, the value of `z`, because the conditional operator's first operand, `(y > z)`, evaluates to 0. Therefore, `max` is assigned the value of the third operand, `z`.

Allowed Operands and the Type of the Result. The following paragraphs discuss the operands that are allowed for *`operand2`* and *`operand3`* and the type that results from those operands.

Both operands can be arithmetic types. The usual arithmetic conversions are performed on *`operand2`* and *`operand3`*. The result has the type of the converted operands. Consider the following example.

```
int max, y = 100;
const int const_z = 1000;

max = (y > const_z) ? y : const_z;
```

In the preceding example, `max` is assigned the value of `const_z`. The result of the preceding conditional expression has the `int` type. The `const` qualifier on the conditional operator's third operand, `const_z`, is ignored.

Both operands can be compatible structure types or union types. The result has that structure or union type.

Both operands can have the `void` type. The result has the `void` type.

One operand can be a pointer, and the other operand can be a null pointer constant. The result has the type of the operand that is a pointer.

One operand can be a pointer to an object or incomplete type, and the other operand can be a pointer to a qualified or unqualified version of `void`. In this case, the other operand is converted to the type "pointer to `void`" before it is evaluated, and the result has that type.

Both operands can be pointers to qualified or unqualified versions of compatible types. The result type is a pointer to the composite type. That is, if the pointed-to types have differing type qualifiers, the result is a pointer to a type qualified with all of the type qualifiers of both pointed-to types. Consider the following example.

```
const int *ptr_to_const_num;

int flag;
int i = 100;
int i1 = 1000;

const int *ptr_to_const_i = &i;
int *ptr_to_i = &i1;

ptr_to_const_num = flag ? ptr_to_const_i : ptr_to_i;
```

In the preceding example, the result of the conditional expression has the type "pointer to `const int`." The conditional expression's second operand is `ptr_to_const_i`, and its third operand is `ptr_to_i`. Because the objects pointed to by the second and third operands are, respectively, `const`-qualified and unqualified, the result is a pointer to a type qualified with the type qualifiers of both pointed-to types. Thus, the result is a pointer to a `const`-qualified `int`.

In VOS C, both operands can have the `char_varying` type only if the conditional expression is the right operand in an assignment.

Assignment Operators

In C, there are two types of assignment operators:

- the simple assignment operator
- the compound assignment operators.

An expression containing one of the assignment operators is called an *assignment expression*.

Simple Assignment Operator

You use the simple assignment operator (=) to store the value specified by the right operand in the object designated by the left operand. The syntax for an expression using the = operator is as follows:

```
left_operand = right_operand
```

The `left_operand` must be a modifiable lvalue. An expression containing the = operator yields the value of the `left_operand` after that operand has been assigned the value of the `right_operand`.

In a simple assignment, the value of the right operand is converted to the type of the assignment expression, and that value is stored in the object designated by the left operand. The side effect of an assignment expression is this modification of the left operand's value. The result of a simple assignment expression is the value stored in the left operand after assignment. A simple assignment expression is not an lvalue.

The *type of the assignment expression* is the type of the left operand unless that operand has a qualified type. In this case, the type is the unqualified version of the left operand's type. See the discussion and examples later in this section for information on how this rule applies to pointer types.

For `left_operand` and `right_operand`, the following combinations of operands are allowed.

- The left operand can be a qualified or unqualified arithmetic type, and the right operand can be an arithmetic type.

- The left operand can be a qualified or unqualified version of a structure or union type that is compatible with the type of the right operand.

- Both operands can be pointers to qualified or unqualified versions of compatible types. In this case, the type pointed to by the left operand must have all of the qualifiers of the type pointed to by the right operand.

- One operand can be a pointer to an object or incomplete type, and the other operand can be a pointer to a qualified or unqualified version of `void`. In this case, the type pointed to by the left operand must have all of the qualifiers of the type pointed to by the right operand.

- The left operand can be a pointer, and the right operand can be a null pointer constant.
- In VOS C, if the left operand has the `char_varying` type, the right operand must have the `char_varying` type or must be convertible to `char_varying`.

In `char_varying` assignments, one or both operands can be a `$substr` built-in function. These `$substr` operands are useful when assigning various types of data, such as array types. See the "`$substr`" section in Chapter 12 for information on the use of `$substr` operands with the assignment operator.

See the "Operations on `char_varying` Data" section in Chapter 4 for information on assignments with `char_varying` operands.

The compiler diagnoses certain **invalidly** qualified operands. Objects that are `const`-qualified cannot be the left operand in an assignment expression. When the left operand in an assignment has the type "pointer to *`type`*," it is invalid for the right operand to have the type "pointer to `const` *`type`*."

The following example shows how the `=` operator works.

```
int i_num = 0;

double d_num = 1.22;

i_num = d_num + i_num;
```

In the preceding example, the assignment `i_num = d_num + i_num` is evaluated in the following manner.

First, the expression `d_num + i_num` is evaluated. The usual arithmetic conversions specify that value of `i_num` be converted to `double` before the addition operation. The expression `d_num + i_num` yields the value 1.22, and the result has the type `double`. The result has the type of the `+` operator's converted operands.

Next, the value of the expression `d_num + i_num` is converted to an `int`, the type of the left operand of the `=` operator. In this conversion, the `double` value 1.22 is truncated to the `int` value 1. The entire assignment expression yields the value 1, and has the `int` type, the type of the `=` operator's left operand. Finally, as a side effect, the value 1 is stored in `i_num`.

When the left operand of an assignment is a qualified version of a pointer, the result type of the assignment expression is the **unqualified** version of the pointer type. The following example and discussion illustrate how the compiler determines an assignment expression's result type when the left operand is a qualified pointer type.

```
int * volatile vol_ptr;

volatile int *ptr_to_vol_int;

int *ptr;
   .
   .
   .
vol_ptr = ptr;                  /*  Example 1  */

ptr_to_vol_int = ptr;           /*  Example 2  */
```

In example 1, the left operand has the type "`volatile` pointer to `int`." The pointer type is `volatile`-qualified. Because the result type of this assignment expression is the unqualified version of the pointer type, the result of the expression `vol_ptr = ptr` has the type "pointer to `int`."

In example 2, the left operand has the type "pointer to `volatile int`." The pointer type is unqualified, but the pointed-to type (`int`) is qualified. The result of the expression `ptr_to_vol_int = ptr` has the type "pointer to `volatile int`."

See the "Syntax and Type Qualifiers" section in Chapter 3 for information on the rules used to interpret declarations that include type qualifiers such as `const` and `volatile`.

Compound Assignment Operators

You use the compound assignment operators to perform a specified operation and store the resulting value in the object designated by the left operand. The syntax for an expression using a compound assignment operator is as follows:

```
operand1 op = operand2
```

The *`operand1`* must be a modifiable lvalue and cannot be `const`-qualified. The *`op`* is one of the following binary operators:

```
*    /    %    +    -    <<    >>    \&    ^    |
```

Depending on the compound assignment operator that you use, the types allowed for *`operand1`* and *`operand2`* vary. See the section in this chapter that explains *`op`* for information on the data types allowed with and conversions performed for each operator. For example, for information on the += operator, see the "Addition Operator" section.

The conversions appropriate to *`op`* are performed on both operands. Then, the operation specified by *`op`* is applied to the two operand values. The value of the expression *`operand1 op`* = *`operand2`* is converted to the type of the assignment expression, and that value is stored in the object designated by the left operand. The side effect of an assignment expression is this modification of the left operand's value. The result of a compound assignment expression is the value stored in the left operand after assignment. A compound assignment expression is not an lvalue.

The *type of the assignment expression* is the type of the left operand unless that operand has a qualified type. In this case, the type is the unqualified version of the left operand's type. See the previous section, "Simple Assignment Operator," for other rules that apply to and examples of the conversions performed for the = operator.

Table 7-8 shows the compound assignment operators and the operation performed by each operator.

Table 7-8. Compound Assignment Operators

Operator	Compound Assignment	Operation
`*=`	*`operand1`* `*=` *`operand2`*	Multiplication assignment: the result of *`operand1`* `*` *`operand2`* is assigned to *`operand1`*.
`/=`	*`operand1`* `/=` *`operand2`*	Division assignment: the result of *`operand1`* `/` *`operand2`* is assigned to *`operand1`*.
`%=`	*`operand1`* `%=` *`operand2`*	Remainder assignment: the result of *`operand1`* `%` *`operand2`* is assigned to *`operand1`*.
`+=`	*`operand1`* `+=` *`operand2`*	Addition assignment: the result of *`operand1`* `+` *`operand2`* is assigned to *`operand1`*.
`-=`	*`operand1`* `-=` *`operand2`*	Subtraction assignment: the result of *`operand1`* `-` *`operand2`* is assigned to *`operand1`*.
`<<=`	*`operand1`* `<<=` *`operand2`*	Left-shift assignment: the result of *`operand1`* `<<` *`operand2`* is assigned to *`operand1`*.
`>>=`	*`operand1`* `>>=` *`operand2`*	Right-shift assignment: the result of *`operand1`* `>>` *`operand2`* is assigned to *`operand1`*.
`&=`	*`operand1`* `&=` *`operand2`*	Bitwise-AND assignment: the result of *`operand1`* `&` *`operand2`* is assigned to *`operand1`*.
`^=`	*`operand1`* `^=` *`operand2`*	Bitwise exclusive-OR assignment: the result of *`operand1`* `^` *`operand2`* is assigned to *`operand1`*.
`|=`	*`operand1`* `|=` *`operand2`*	Bitwise inclusive-OR assignment: the result of *`operand1`* `|` *`operand2`* is assigned to *`operand1`*.

A compound assignment expression *`operand1 op`* `=` *`operand2`* is nearly equivalent to the following simple assignment expression:

```
    operand1 = operand1 op operand2
```

In the compound assignment, *operand1* is evaluated once. However, in the simple assignment, *operand1* is evaluated twice. This difference can change the behavior of a program when the evaluation of *operand1* causes side effects, such as when *operand1* involves a function call.

The following example shows how the `+=` compound assignment operator works.

```
int i = 99;
char c = 1;

i += c;
```

In the preceding assignment, first the value of `c` is promoted to the `int` type because the usual arithmetic conversions are applied to both operands. Next, when the addition operation `i + c` is performed, no conversion to the `int` type of the left operand is needed because the value that results from `i + c` is already an `int`. Finally, as a side effect, the value of the resulting expression, 100, is stored in `i`. The entire assignment expression has the type of the left operand, `int`, and yields the value 100.

Comma Operator

You use the comma operator (`,`) to evaluate one operand as a void expression before evaluating a second operand. The result of the evaluation of the second operand is the value of a comma-operator expression. The syntax for an expression using the comma operator is as follows:

```
expression1 , expression2
```

Both *expression1* and *expression2* must be valid expressions.

The left operand, *expression1*, is evaluated as a void expression. That is, after the left operand is evaluated for its side effects, the value of the resulting expression is discarded. Then the right operand, *expression2*, is evaluated. An expression containing the comma operator yields the value of the right operand. The result has the type of the right operand and is not an lvalue.

Unless it is enclosed in parentheses, the comma operator cannot be used in contexts where the `,` token is a punctuator (for example, in the argument list of a function call or in an initializer list).

The following example shows how the comma operator works when it is part of the argument list in a function call.

```
void func(int, int, int);

int a, b, c;
    .
    .
    .
func(a, (b = 5, b + 3), c);
```

In the preceding function call, the second argument is a parenthesized expression containing a comma operator. First, the left operand `b = 5` is evaluated for its side effects, assigning the value 5 to `b`. Next, the expression `b + 3` is evaluated. The second argument to `func`, the comma-operator expression, yields the value 8. This result is the value of the right operand and has the `int` type.

Constant Expressions

A *constant expression* is an expression that can be evaluated at **compile time** to a constant. Each constant expression evaluates to a constant that is in the range of representable values for its type. You can use a constant expression wherever a constant can be used. The following example declaration contains a constant expression that specifies the number of elements to allocate in an array.

```
char array[8 * sizeof(int) + 1];
```

A constant expression can contain other expressions, which optionally contain one or more operators. Figure 7-2 shows the operators that are allowed within a constant expression.

Three of the unary operators:

```
-   ~   !
```

Any of the binary operators:

```
*   /   %   +   -   <<   >>   <   <=   >   >=   ==   !=   &   ^ | && ||
```

The conditional operator:

```
? :
```

Figure 7-2. Operators Allowed in a Constant Expression

Except when a constant expression is used within the operand of a `sizeof` operator, a constant expression **cannot** contain any of the following operators: assignment (`=`), function-call (`()`), comma (`,`), or any of the increment (`++`) or decrement (`--`) operators. However, within the operand of the `sizeof` operator, any operator is allowed because the `sizeof` operand is never evaluated.

An *integral constant expression* has an integral type. The operands in an integral constant expression can be character constants, integer constants, enumeration constants, and `sizeof` expressions. If the operand of `sizeof` is a `$substr` expression, the second and third arguments to `$substr` must be constant expressions. Cast operators in integral expressions can only be used to convert arithmetic types to integral types, except when the cast is part of an operand to the `sizeof` operator.

You use an integral constant expression to specify the following:

- the size of a bit-field member in a structure
- the value of an enumeration constant when such a value is explicitly given
- the number of members in an array
- the value of a `case` label's constant.

When you use a constant expression as an initializer in the declaration of an object, other types of constant expressions, in addition to integral constant expressions, are allowed. Constant expressions are required as initializers for objects that have static storage duration. As an initializer, a constant expression can evaluate to one of the following:

- an arithmetic constant expression
- an address constant
- an address constant for an object type plus or minus an integral constant expression
- a null pointer constant.

In addition to the preceding, a `$substr` expression, a VOS C extension, is a constant expression that can be used as an initializer **if** all three arguments to `$substr` are constant expressions.

An *arithmetic constant expression* is a constant expression having an arithmetic type. You can, for example, use an arithmetic constant expression to initialize an object having an arithmetic type. The operands in an arithmetic constant expression can be character constants, integer constants, enumeration constants, floating-point constants, and `sizeof` expressions. If the operand of `sizeof` is a `$substr` expression, the second and third arguments to `$substr` must be constant expressions. If you use a cast operator as part of an arithmetic constant expression, the cast can convert an arithmetic type only to another arithmetic type. In VOS C, floating-point arithmetic is **not** allowed in an arithmetic constant expression.

The following example declaration contains an arithmetic constant expression that specifies the initial value of a `float` variable.

```
float f_num = (float)999.999;
```

An *address constant* is a pointer to an lvalue designating an object with static storage duration, or is a pointer to a function. You can, for example, use an address constant expression to initialize a pointer variable. You create an address constant explicitly by using an expression containing the address-of operator (`&`), or implicitly by using an expression of array or function type. To create an address constant, you can use pointer casts and the following operators: array subscript (`[]`), structure/union member (`.`), structure/union pointer (`->`), address-of (`&`), and indirection (`*`) operators. However, in creating the address constant, the value (as opposed to the address) of an object cannot be accessed using any of these operators.

The following example declarations contain address constants that specify the initial value of a pointer variable.

```
int i_num, *num_ptr = &i_num;

char array[10], *array_ptr = &array[9];
```

A *null pointer constant* is an integer constant expression equal to the value 0, or such an expression cast to `void *`. Initializing a pointer to `NULL` indicates that the pointer holds no valid storage location. In VOS C, the `OS_NULL_PTR` constant is also a constant expression that you can assign to a pointer. The "Null Pointer Constants" section in Chapter 4 contains more information on these types of constant expressions.

The semantic rules for the evaluation of a constant expression are the same as the rules used for evaluating a nonconstant expression.

VOS C Operators Summary

Table 7-9 is a summary of the VOS C operators.

Table 7-9. VOS C Operators Summary *(Page 1 of 3)*

Operator	**Name**	**Example**	**Description**
`[]`	Array subscript	`a[b]`	Accesses element `b` of array `a`
`()`	Function call	`f()`	Invokes function `f`
`.`	Structure/union member	`s.a`	Accesses structure member or union member `a`
`->`	Structure/union pointer	`p->a`	Accesses a structure member or union member `a`
`++`	Postincrement	`a++`	Increments `a`'s value after `a` is evaluated
`--`	Postdecrement	`a--`	Decrements `a`'s value after `a` is evaluated
`&`	Address-of	`&a`	Gets the address of object or function `a`
`*`	Indirection	`*p`	Dereferences pointer `p`
`~`	Bitwise complement	`~a`	Forms the bitwise complement of `a`
`!`	Logical negation	`!a`	Computes the logical negation of `a`
`-`	Unary minus	`-a`	Negates the value of `a`
`()`	Cast	`(char)a`	Converts `a`'s value to the specified type
`sizeof`	Sizeof	`sizeof(a)`	Determines the size, in bytes, of `a`
`++`	Preincrement	`++a`	Increments `a`'s value before `a` is evaluated
`--`	Predecrement	`--a`	Decrements `a`'s value before `a` is evaluated

Table 7-9. VOS C Operators Summary *(Page 2 of 3)*

Operator	Name	Example	Description
`*`	Multiplication	`a * b`	Multiplies `a` by `b`
`/`	Division	`a / b`	Divides `a` by `b`
`%`	Remainder	`a % b`	Computes the remainder when `a` is divided by `b`
`+`	Addition	`a + b`	Adds `a` and `b`
`-`	Subtraction	`a - b`	Subtracts `b` from `a`
`<<`	Left shift	`a << b`	Shifts `a` to the left by `b` bit positions
`>>`	Right shift	`a >> b`	Shifts `a` to the right by `b` bit positions
`<`	Less than	`a < b`	Tests whether `a` is less than `b`
`<=`	Less than or equal to	`a <= b`	Tests whether `a` is less than or equal to `b`
`>`	Greater than	`a > b`	Tests whether `a` is greater than `b`
`>=`	Greater than or equal to	`a >= b`	Tests whether `a` is greater than or equal to `b`
`==`	Equality	`a == b`	Tests whether `a` equals `b`
`!=`	Inequality	`a != b`	Tests whether `a` does not equal `b`
`&`	Bitwise AND	`a & b`	Forms the bitwise AND of `a` and `b`
`^`	Bitwise exclusive OR	`a ^ b`	Forms the bitwise exclusive OR of `a` and `b`
`|`	Bitwise inclusive OR	`a | b`	Assigns the result of `a >> b` to `a`
`&&`	Logical AND	`a && b`	Tests whether both `a` and `b` are nonzero
`||`	Logical OR	`a||b`	Tests whether either `a` or `b` is nonzero
`? :`	Conditional	`a ? b : c`	If `a` is nonzero, evaluates `b`; if `a` is 0, evaluates `c`. Yields the value of the evaluated operand
`=`	Assignment	`a = b`	Assigns the value of `b` to `a`
`*=`	Multiplication assignment	`a *= b`	Assigns the result of `a * b` to `a`
`/=`	Division assignment	`a /= b`	Assigns the result of `a / a` to `a`
`%=`	Remainder assignment	`a %= b`	Assigns the result of `a % b` to `a`
`+=`	Addition assignment	`a += b`	Assigns the result of `a + b` to `a`
`-=`	Subtraction assignment	`a -= b`	Assigns the result of `a - b` to `a`
`<<=`	Left-shift assignment	`a <<= b`	Assigns the result of `a << b` to `a`
`>>=`	Right-shift assignment	`a >>= b`	Assigns the result of `a >> b` to `a`

Table 7-9. VOS C Operators Summary *(Page 3 of 3)*

Operator	Name	Example	Description
`&=`	Bitwise-AND assignment	`a &= b`	Assigns the result of `a & b` to `a`
`^=`	Bitwise exclusive-OR assignment	`a ^= b`	Assigns the result of `a ^ b` to `a`
`|=`	Bitwise inclusive-OR assignment	`a|= b`	Assigns the result of `a | b` to `a`
`,`	Comma	`a , b`	Evaluates `a` as a void expression and then evaluates `b`

Chapter 8: Statements

A *statement* specifies an action to perform. Unless the explanations in this chapter state otherwise, statements are executed in the order in which they appear in the program. This chapter explains the statements provided in the C language.

Most C language statements can be grouped into one of three functional categories:

- An *iteration statement* causes a loop body to execute repeatedly until a controlling expression is false (0). The iteration statements are `do`, `for`, and `while`.
- A *jump statement* causes an unconditional jump to another location in the program code. The jump statements are `break`, `continue`, `goto`, and `return`.
- A *selection statement* selects among a set of statements depending on the value of a controlling expression. The selection statements are `if` and `switch`.

In addition to the preceding statements, this chapter discusses statement syntax and compound statements. The chapter also describes expression, labeled, and null statements.

Statement Syntax

Statements can be either single or compound. A single statement is terminated with a semicolon (`;`). A compound statement begins with an opening brace (`{`) and is terminated with a closing brace (`}`). Some C statements, such as `do` and `continue`, have a semicolon as part of their syntax.

In the descriptions in this chapter, two elements of statement syntax appear frequently. These syntax elements are as follows:

- *`statement`* — a single or compound statement
- *`expression`* — a full expression that is not part of another expression.

The expression, iteration, and selection statements require *`expression`* as part of each statement. In a `return` statement, *`expression`* is optional.

A *controlling expression* is a full expression that determines what statement or statements are executed in an `if` or `switch` statement, and how long the loop body is executed in a `do`, `for`, or `while` statement. With the exception of the controlling expression in a `for` statement, a controlling expression must be enclosed in parentheses.

All three expressions used in a `for` statement are enclosed in one set of parentheses.

Compound Statements

A *compound statement*, or block, is a set of statements grouped into one syntactic unit and delimited by braces (`{ }`). The following example is a compound statement, which contains another compound statement.

```
{
   printf("%c", line_buffer[line_length]);
   ++char_count;
   if (char_count == 79)
      {
      printf("\n");
      char_count = 0;
      }
   }
```

The syntax for a compound statement is as follows:

```
{
[declaration;]...
[statement]...
}
```

The *`declaration`* can be a function or object declaration and can include an initializer. Declarations must immediately follow the compound statement's opening brace. The *`statement`* can be any single or compound statement.

If an object declaration with no storage-class specifier or with the `auto` or `register` specifier appears inside a compound statement, the object has automatic storage duration, and the object's identifier has block scope. Storage for automatic objects is allocated when program control enters the block and deallocated when program control exits the block.

When program control enters the block, initializers for objects with automatic storage duration are evaluated, and the values are stored in the objects. Initial values, if any are specified, are set upon **each normal entry** into the block: when the function is invoked or when control enters a compound statement. With `do`, `for`, and `while` statements, when a declaration located within the loop body has an initializer, the object is initialized with each iteration of the loop.

If control enters a compound statement by a transfer to a labeled statement, initial values, if specified, are not set for automatic data within that block.

> **Note:** Transfer of program control into a block through the use of a `goto` statement is not normal entry into the compound statement and, therefore, is not recommended.

See Chapter 3 for more information on automatic storage duration and block scope.

Expression Statements

An *expression statement* consists of an expression followed by a semicolon. When an expression statement executes, the expression is evaluated for its side effects, and the value of the expression is discarded. Most statements in a C program, including assignments and function invocations, are expression statements.

The syntax for an expression statement is as follows:

```
expression ;
```

The following examples are expression statements.

```
monthly_payment = (loan_amount * rate) / factor;

remainder = range % 501;

*temp_ptr--;

--i, ++j;

(void)s$error(&error_code, &caller, &message_text);

i = (j == k ? 0 : j);

file_ptr = creat("file_name", mode);
```

See Chapter 7 for information on expressions.

The `break` Statement

Purpose

The `break` statement terminates execution of the smallest enclosing `do`, `for`, `switch`, or `while` statement.

Syntax

```
break ;
```

Explanation

After the `break` statement executes, program control passes to the point just beyond the terminated statement that contained the `break` statement. The `break` statement can occur only in or as the body of a `do`, `for`, `switch`, or `while` statement.

Examples

The following example uses the `break` statement to terminate execution of the `switch` statement after a selected case executes.

```
switch(case_item)
   {
   case (1): printf("First case\n"); break;

   case (2): printf("Second case\n"); break;

   case (3): printf("Third case \n"); break;

   case (4): printf("Fourth case\n"); break;

   default: printf("All other cases\n");
   }
```

The following example uses the `break` statement to terminate execution of the `for` statement if end-of-file (`EOF`) is encountered.

```
for (i = 0; i < 80; ++i)
   {
   character = getchar();
   if (character == EOF)
      break;
      .
      .
      .
   }
```

The `continue` Statement

Purpose

The `continue` statement transfers program control to the loop-continuation location of the immediately enclosing `do`, `for`, or `while` statement.

Syntax

```
continue ;
```

Explanation

The `continue` statement can occur only in the body of a `do`, `for`, or `while` loop. The `continue` statement has the effect of skipping the remaining portion of the current iteration of the loop and transferring program control to the loop-continuation location, where the controlling expression is evaluated. Figure 8-1 illustrates this effect. The next iteration of the loop begins if the loop's controlling expression evaluates to true (nonzero).

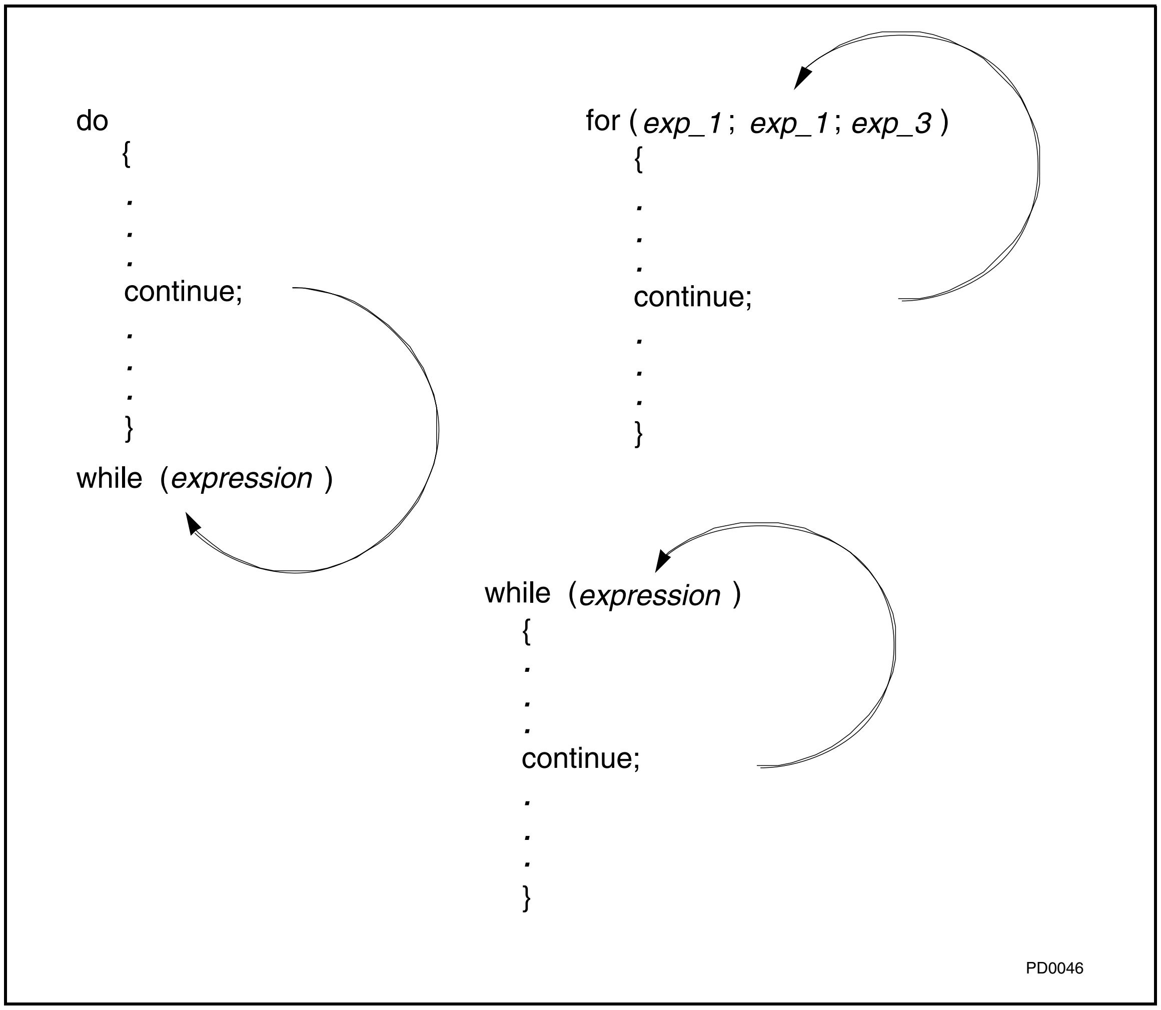


Figure 8-1. Loops and the `continue` Statement

Examples

The following example uses the `continue` statement to skip the remaining statements in a loop if a space character is read.

```
while ( (c = getchar()) != EOF )
   {
   if (c == ' ')
      continue;
      .
      .
      .
   }
```

The `do` Statement

Purpose

The `do` statement always executes a loop once, then evaluates a controlling expression, and executes the loop repeatedly as long as the controlling expression is true.

Syntax

```
do
   statement
while ( expression ) ;
```

Explanation

Unlike the `for` and `while` statements, the `do` statement always executes *`statement`* (the loop body) at least once. The `do` statement tests *`expression`* **after** each execution of *`statement`*. If *`expression`* is false (0), the `do` statement terminates. If *`expression`* is true (nonzero), the `do` statement executes *`statement`* until *`expression`* becomes false.

You can use the `break`, `goto`, and `return` statements to transfer program control out of a `do` statement's loop body. You can use the `continue` statement to skip the remainder of the loop body and transfer program control to the loop-continuation location, where the loop's controlling expression is evaluated.

Examples

The following example uses the `do` statement to create a loop that writes characters into an array, `line_buffer`, until a newline character is read.

```
do
   {
   character = getchar();
   line_buffer[line_length] = character;
   line_length++;
   } while (character != '\n');
```

The `for` Statement

Purpose

The `for` statement evaluates three expressions, executes a loop if the controlling expression is true, and continues to execute the loop until the controlling expression is false.

Syntax

```
for ([expression_1]; [expression_2]; [expression_3 ]) statement
```

Explanation

The `for` statement can contain three parenthesized expressions that set up the environment for the loop.

- The `for` statement evaluates *`expression_1`* once before *`expression_2`* is evaluated for the first time. Typically, *`expression_1`* initializes one or more loop-control variables.
- The `for` statement evaluates *`expression_2`*, the controlling expression, before each iteration of the loop. The `for` statement terminates when *`expression_2`* becomes false (0).
- The `for` statement evaluates *`expression_3`* **after** each pass through the loop. Typically, *`expression_3`* is an operation, such as incrementing one or more loop-control variables.

Both *`expression_1`* and *`expression_2`* are followed by a required semicolon (`;`). No semicolon follows *`expression_3`*. All three expressions are enclosed in one set of parentheses.

The following `while` statement illustrates when and how many times the three expressions used in a `for` statement are evaluated.

```
expression_1;
while ( expression_2 )
   {
   .
   .
   .
   expression_3;
   }
```

All three expressions in the `for` statement are optional. If you omit *`expression_1`* or *`expression_3`*, the omitted expression is evaluated as a void expression (that is, an expression with no value). If you omit *`expression_2`*, it is evaluated as a nonzero constant. The following `for` statement creates an infinite loop.

```
for( ; ; )
   {
   .
   .
   .
   }
```

You can use the `break`, `goto`, and `return` statements to transfer program control out of a `for` statement's loop body. You can use the `continue` statement to skip the remainder of the loop body and transfer program control to the loop-continuation location, where the loop's controlling expression is evaluated.

Examples

The following example uses the `for` statement to control execution of a single statement that writes space characters into an array, `line_buffer`.

```
for (line_length = 0; line_length < 255; ++line_length)
   line_buffer[line_length] = ' ';
```

The following example uses the `for` statement to control execution of a compound statement that displays an array of characters on the terminal's screen.

```
for (line_length = 0, char_count = 0; line_length < 255; ++line_length)
   {
   printf("%c", line_buffer[line_length]);
   ++char_count;
   if (char_count == 79)
      {
      printf("\n");
      char_count = 0;
      }
   }
```

Notice, in the preceding example, that more than one loop-control variable can be initialized before the controlling expression is evaluated for the first time. In this example, `line_length` and `char_count` are separated by a comma operator. Both are set to 0 before the controlling expression is initially evaluated. Similarly, by using a comma operator, it is also possible to have more than one operation performed in *`expression_3`*.

The `goto` Statement

Purpose

The `goto` statement transfers program control to a labeled statement.

Syntax

```
goto label ;
```

Explanation

The `goto` statement unconditionally transfers program control to the statement specified by *`label`*. The specified label must be located **in the same function** as the `goto` statement.

For nonlocal jumps, use the `setjmp` and `longjmp` library functions. See Chapter 11 for more information on these functions.

Examples

The following example uses the `goto` statement to transfer program control to the `done` label.

```
goto done;
   .
   .
   .
done:
   exit(1);
```

The `if` Statement

Purpose

The `if` statement evaluates a controlling expression and executes a statement if the expression is true. When the `if` statement contains an `else` keyword, the statement following the `else` executes if the controlling expression is false.

Syntax

```
  if ( expression )
       statement_1
[else
     e statement_2]
```

Explanation

The `if` statement evaluates *`expression`*. The *`expression`* must have an integral, floating-point, or pointer type. If *`expression`* is true (nonzero), the `if` statement executes *`statement_1`*. If *`expression`* is false (0), the `if` statement does one of the following:

- When an `else` clause is not specified, the `if` statement terminates. No statement is executed.
- When an `else` clause is specified, the `if` statement executes *`statement_2`*.

When an `else` is present, it is associated with the immediately preceding `if` that is in the same block, but not in an enclosed block. In the following example, the `else` is associated with the second `if`.

```
if (expression)

     if (expression)
        statement;
     else
        statement;
```

However, in the next example, the `else` is associated with the first `if` because the second `if` is not in the same block as the `else`.

```
if (expression)
   {                        /*  Start of block  */
   if (expression)
      statement;
   }                        /*  End of block    */
else
   statement;
```

Examples

Some examples of `if` statements are as follows:

```
if (char_count == 79)
   printf("\n");

if (char_count == 79)
   {
   printf("\n");
   char_count = 0;
   }

if ( (number % 2) == 0 && number != 0 )
   printf("The number is even.\n");
else
   if (number != 0)
      printf("The number is odd.\n");
   else
      printf("The number is 0.\n");
```

The Labeled Statement

Purpose

A labeled statement acts as the target of a `goto` statement that specifies the label name.

Syntax

```
        label : statement
```

Explanation

The *`label`* is any valid identifier. Any C statement can be prefixed by a label. The only use of a labeled statement is as the target of a `goto` statement. Labels have function scope. Therefore, a label must be located **in the same function** as the `goto` statement that specifies the label. A label must be followed by a statement, possibly a null statement.

A `case` or `default` label can appear only in a `switch` statement. See “The `switch` Statement” section later in this chapter for information on these labels.

Examples

The following example uses a statement prefixed with the label `try_again` as the target of a `goto` statement.

```
try_again:
   puts("Enter a number from 1 through 10.");
   scanf("%d", &number);

   if ( number < 1 || number > 10 )
      goto try_again;
```

The Null Statement

Purpose

A null statement does not contain an expression and performs no operations.

Syntax

```
;
```

Explanation

A null statement consists of a semicolon with no associated expression. Typically, you use a null statement in those situations where a statement is required by the syntax but no action is needed. For example, a null statement allows a label before the closing brace in a block. Or, a null statement can supply an empty loop body in a `do`, `for`, or `while` statement when the loop-control expression(s) contains all of the required operations.

Examples

The following example uses a null statement with a label that appears immediately before the closing brace of a block.

```
while (cntrl_exp1)
   {
   .
   .
   .
   while (cntrl_exp2)
      {
      .
      .
      .
      if (flag)
         goto end_loop;
      .
      .
      .
      }
   .
   .
   .
   end_loop:
      ;        /*  Null statement  */
   }
```

The following example uses a null statement to provide an empty loop body for a `for` statement.

```
for (count = 0; getchar() != EOF; ++count)
   ;           /*  Null statement  */
```

The **`return`** Statement

Purpose

The `return` statement terminates execution of the current function and returns program control to its caller.

Syntax

```
return [ expression ] ;
```

Explanation

The `return` statement can include an optional *`expression`*. If you use a `return` statement without an *`expression`*, no value is returned to the function's caller. If you use a `return` statement with an *`expression`*, the value of *`expression`* is returned to the caller as the value of the function-call expression. If the returned value does not have the same data type as was specified in the function's definition, *`expression`* is converted, as if by assignment, to the type indicated in the function's definition.

Depending on the level of data-type checking that you specify in a `#pragma` preprocessor control line specifying the `type_checking` option, or in the corresponding command-line argument, the compiler issues an error message in the following situations:

- if you use a `return` statement with an *`expression`* in a function defined as returning `void`
- if you use a `return` statement without an *`expression`* for a function defined as returning some value other than `void`.

A function can include any number of `return` statements, with or without expressions. In a function, it is not possible for a `return` statement to pass back more than one value to the calling function.

When program control reaches a function block's closing brace (`}`), the equivalent of a `return` statement without an expression is executed.

Examples

The following example uses the `return` statement to terminate a function.

```
int  func(void);

main()
{
   int i_num;

   i_num = func();           /*  After this statement executes, i_num
= 9  */

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

int func(void)
{
   double return_value = 9.8;
  return (return_value);   /*  Conversion from double to int occurs
*/
}
```

Notice that, in `func`, the value of `return_value` is converted from `double` to `int` before the value is passed back to `main`.

The `switch` Statement

Purpose

The `switch` statement evaluates a controlling expression and executes one or more consecutive cases depending on the value of the expression.

Syntax

```
switch ( expression )
   statement
```

Explanation

When *`expression`* is evaluated, the integral promotions are performed on *`expression`*. After evaluation, *`expression`* must yield a value of an integral type.

Typically, *`statement`* is a compound statement containing one or more labeled cases and a `default` case. The syntax for a `switch` statement of this type is as follows:

```
switch (expression)
   {

{case (constant_expression) :[statement]} ...

[default :[statement]]

   }
```

Each `case` label's *`constant_expression`* is an integral constant expression. In VOS C, *`constant_expression`* can range from -32,768 through 32,767 **or** from 0 through 65,535. Within a `switch` statement, two cases cannot have the same value for *`constant_expression`*.

> **Note:** If *`expression`* or *`constant_expression`* is a value from 32,768 through 65,535, that value is mapped to the negative values -32,768 through -1, respectively.

As examples of how values for *`expression`* and *`constant expression`* are mapped, the value 65,535 is mapped to -1, the value 65,534 is mapped to -2, and so forth. Therefore, the compiler interprets -1 and 65,535 as duplicate values and issues an error message.

A `switch` statement can contain only one `default` label. However, an enclosed `switch` statement can have its own `default` label and cases with the same *`constant_expression`* values as the enclosing `switch` statement.

Before a case is selected, each *`constant_expression`* is converted to the type of *`expression`*. Next, the `switch` statement does one of the following:

- If a `case` label's *`constant_expression`* equals *`expression`*, `switch` jumps to that `case` label and executes one or more consecutive statements, starting with the statement immediately following the selected label.

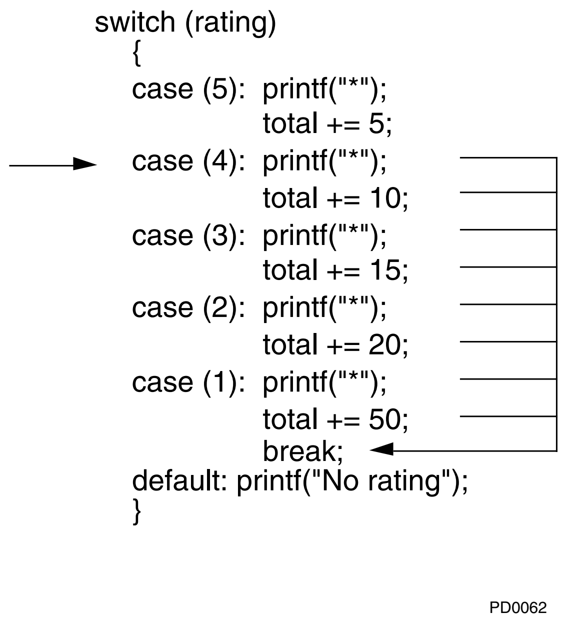
- If no `case` label's *`constant_expression`* equals *`expression`* and a `default` label is present, `switch` jumps to the `default` label and executes the statement specified for that label.

- If no `case` label's *`constant_expression`* is equal to *`expression`* and a `default` label is not present, `switch` jumps to the closing brace (`}`) marking the end of the `switch` statement, and the `switch` statement terminates.

In the preceding process, `case` and `default` labels located in enclosed `switch` statements are not considered for selection.

When a `case` or `default` label is selected, `switch` continues to execute statements until a `break`, `goto`, or `return` statement is encountered, or until the `switch` statement's closing brace is reached. Thus, it is possible for program control to fall through from one `case` label's *`statement`* to the next `case` label's *`statement`* so that a series of statements executes.

Figure 8-2 illustrates this fall-through of program control. In the figure, if `rating` equals 4, the `switch` statement executes all statements between the `case (4)` label and the `break` statement. Therefore, `switch` displays four asterisks on the terminal's screen, and assigns values equaling 95 to `total`. The `switch` statement terminates when the `break` statement is encountered.



```
switch (rating)
   {
   case (5):  printf("*");
              total += 5;
   case (4):  printf("*");
              total += 10;
   case (3):  printf("*");
              total += 15;
   case (2):  printf("*");
              total += 20;
   case (1):  printf("*");
              total += 50;
              break;
   default: printf("No rating");
   }
```

Figure 8-2. The `switch` Statement and Program Control

Examples

The following example uses the `switch` statement to select the appropriate case, based on the value of `character`.

```
switch(character)
   {
   case ('a'):
   case ('A'): puts("First case");
               break;
   case ('b'):
   case ('B'): puts("Second case");
               break;
   case ('c'):
   case ('C'): puts("Third case");
               break;
   default: puts("Not a valid case");
            return;
   }
```

The **`while`** Statement

Purpose

The `while` statement evaluates a controlling expression, executes a loop if the expression is true, and continues to execute the loop until the controlling expression is false.

Syntax

```
while ( expression )
   statement
```

Explanation

The `while` statement evaluates *`expression`* **before** each execution of *`statement`* (the loop body). If *`expression`* is false (0), the `while` statement terminates. If *`expression`* is true (nonzero), the `while` statement executes *`statement`* until *`expression`* becomes false. If *`expression`* is initially false, *`statement`* is never executed.

You can use the `break`, `goto`, and `return` statements to transfer program control out of a `while` statement's loop body. You can use the `continue` statement to skip the remainder of the loop body and transfer program control to the loop-continuation location where the loop's controlling expression is evaluated.

Examples

The following example uses the `while` statement to control execution of a null statement. As long as space characters are entered at the terminal's keyboard, the `while` statement continues to execute and, in effect, discard the space characters.

```
while ( (character = getchar()) == ' ' )
   ;
```

The following example uses the `while` statement to read characters from `stdin` until a newline character is read. The loop body also writes the characters into the `line_buffer` array.

```
while ( character != '\n' )
   {
   line_buffer[line_length] = character;
   line_length++;
   character = getchar();
   }
```

Chapter 9: Preprocessor Directives

The VOS C preprocessor allows you to use certain preprocessor directives, such as `#define`, that make it possible to develop programs that are easier to read, easier to change, and easier to port to a different system.

This chapter explains how you use the VOS C preprocessor directives.

The C Preprocessor: An Overview

When you invoke the C compiler, the `c` command, as part of the compilation, calls the C preprocessor. The C preprocessor processes the source module before the compiler translates the source module into object code. Preprocessing includes a number of separate translation phases. If the last character in a source line is the `\` character, the C preprocessor deletes the character and combines that line with the next line. Next, the C preprocessor decomposes the source text into preprocessing tokens and sequences of white-space characters. The preprocessor treats each comment as a single space character.

A *preprocessing token* can be any of the following: a header name, an identifier, a preprocessing number, a character constant, a character-string literal, an operator, a punctuator, or a single non-white-space character that is not one of the preceding. Preprocessing number tokens lexically include all integer constant and floating-point constant tokens.

After decomposing the source text into preprocessing tokens, the C preprocessor performs a variety of other tasks including the following:

- incorporates into the source module all files specified with the `#include` directive
- executes all other preprocessing directives
- expands all macros.

The preprocessing tasks described earlier, such as decomposition into preprocessing tokens, are also performed on any included files.

If you want to see the output produced by the C preprocessor prior to compilation, you can use the `c_preprocess` command to invoke the standalone C preprocessor. Typically, the `c_preprocess` command is used to produce an expanded source module so that you can debug complex function-like macros. Also, compiling such an expanded source module significantly reduces compile time since the compiler does not have to perform an initial pass over the file.

See the *VOS C User's Guide (R141)* for more information on using the `c_preprocess` command.

Preprocessor Control Line Syntax

A preprocessor control line consists of a sequence of preprocessing tokens. The syntax for a preprocessor control line is as follows:

```
[white space] # directive [preprocessing_tokens]
```

Each control line begins with the `#` token that is either the first character on the line or is optionally preceded by white space. A `#` token and a directive that appears within a character-string literal or comment is not interpreted as a preprocessor directive. Each preprocessor control line is terminated by the end of the line.

The *`directive`* can be any VOS C preprocessor directive. Most directives require other *`preprocessing_tokens`*. See the appropriate section in this chapter for the syntax and interpretation of the *`preprocessing_tokens`* that are specified with each directive. The only white-space characters that can appear between preprocessing tokens within a control line, from just after the introducing `#` to just before the terminating newline character, are the space and horizontal tab characters. A comment is treated as a single white-space character and can appear within a preprocessor control line.

To continue a preprocessor control line for more than one line, you can use a reverse slant (`\`) as the last character on the line. Used in this way, the reverse slant has the effect of concatenating the last character on that line with the first character on the next line. For example, the following preprocessor control lines use a reverse slant to specify a macro definition that continues for two lines.

```
#define MESSAGE "Some macro definitions are just too long to fit \
on a single line."

#define prt_sqr(num)  printf("The number squared = %d\n", \
( (num) * (num) ))
```

In each of the preceding examples, the preprocessor removes the `\` character at the end of the line and concatenates the last character on the line with the first character on the next line.

Macro Replacement

The `#define` preprocessor directive causes a macro name to be defined to the preprocessor. In VOS C, a *macro name* is an identifier that immediately follows the `#define` directive in a preprocessor control line. Figure 9-1 shows an example of the two types of macro definitions: an object-like macro and a function-like macro.

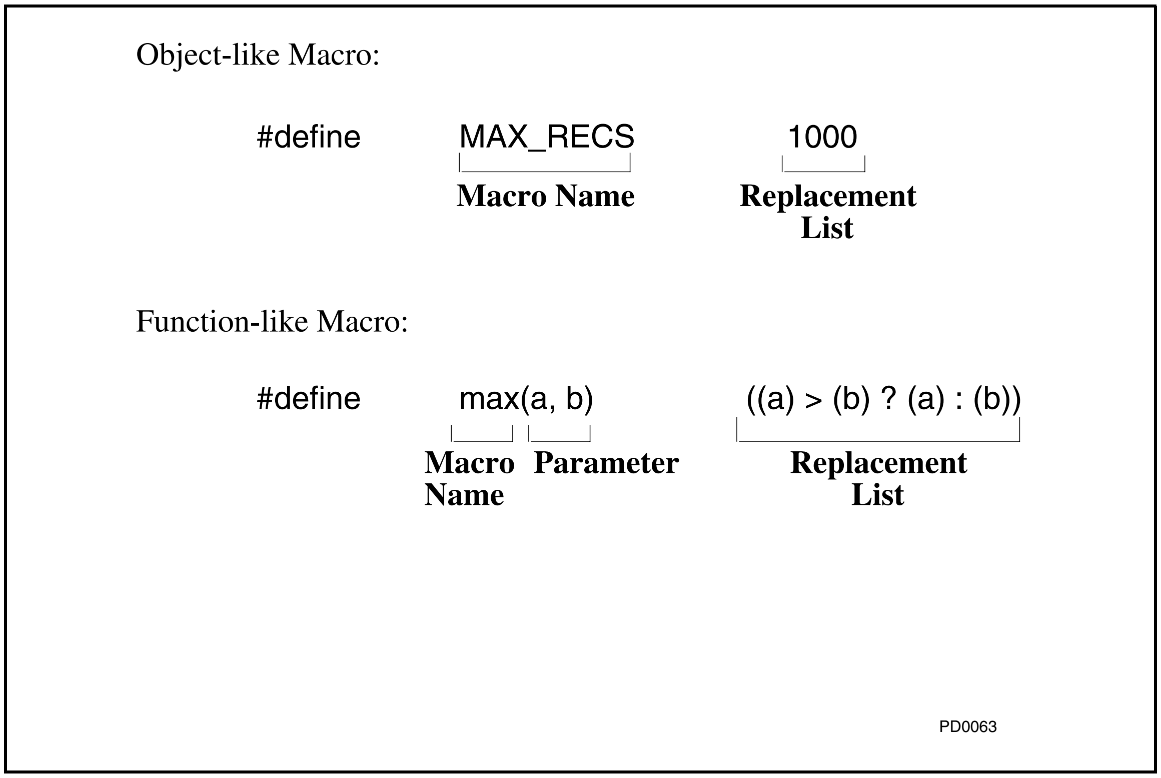


Figure 9-1. Sample Macro Definitions

There is a separate name space for macro names, such as `MAX_RECS` and `max`. The scope of a macro definition extends (independent of block structure) from the point at which the macro name is defined until an `#undef` directive causes the macro name to be no longer defined or until the end of the source module, whichever occurs first.

The simplest form of macro replacement takes place with an object-like macro, such as `MAX_RECS`. With this type of macro, *macro replacement* occurs when, after a macro name has been defined, the C preprocessor scans the source module for the name and replaces the name with the series of tokens specified in the `#define` directive. Macro replacement is also called macro expansion.

This section explains how to define and use the two types of macros:

- object-like macros
- function-like macros.

In addition, this section explains how to use the two operators associated with macro definition: the stringize operator (`#`) and the concatenation operator (`##`).

Note: The VOS Symbolic Debugger recognizes the names of various entities defined in the source modules that comprise the program if you specify the `-table` or `-production_table` command-line argument. However, because macro names are replaced during the preprocessing phase of compilation, the debugger does not recognize macro names.

Object-like Macros

An *object-like macro* is the simplest form of macro and has the following form:

```
#define macro_name [replacement_list]
```

The *`macro_name`* is a name consisting of not more than 256 characters. The characters that are allowed in *`macro_name`* are the same as are allowed for any valid VOS C identifier. The optional *`replacement_list`* consists of one or more preprocessing tokens. Any white-space characters preceding or following the *`replacement_list`* are not considered part of the replacement list. A preprocessor control line using `#define` is terminated by the end of the line.

When an object-like macro is defined using the preceding syntax, each subsequent instance of *`macro_name`* is replaced by preprocessing tokens in the *`replacement_list`*. Macro replacement does not occur within character constants, character-string literals, or comments. The *`replacement_list`* is then rescanned for more macro names. See the "Rescanning and Further Replacement" section later in this chapter for information on rescanning and further macro replacement.

The following example contains two object-like macros and a reference to each macro.

```
#define MAX_RECS 1000

#define FILE_ERROR "Error processing the file."

struct
   {
   int i;
   char c;
   } records[MAX_RECS];

puts(FILE_ERROR);
```

When the preprocessor expands the preceding code, the expanded file contains the following:

```
struct
   {
   int i;
   char c;
   } records[1000];

puts("Error processing the file.");
```

As shown in the preceding example, the `#define` directive is often used to define a numeric constant, such as `MAX_RECS`. This constant is sometimes called a manifest constant or a defined constant. By convention, a defined constant uses all uppercase letters.

If you omit the *`replacement_list`* in the macro definition, the *`macro_name`* is defined to the preprocessor and subsequent instances of the *`macro_name`* are replaced with zero preprocessing tokens. This type of macro definition is sometimes used to control conditional inclusion. For example, the following macro definition causes the `SPECIAL` macro to be defined to the preprocessor.

```
#define SPECIAL

#if defined SPECIAL
    .
    .
    .
```

In the preceding example, after the `SPECIAL` macro is defined, the expression `defined SPECIAL` evaluates to the value 1.

An object-like macro can be redefined by a subsequent `#define` directive provided the second definition is an object-like macro definition and the two replacement lists are identical. Two replacement lists are identical if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

For information on defining an object-like macro a second time with a different definition than when the name was first defined, see the "Undefining a Macro" section later in this chapter.

Function-like Macros

A *function-like macro* is a parameterized form of macro and has the following form:

```
#define macro_name([identifier_list] )[replacement_list]
```

The *`macro_name`* is a name consisting of not more than 256 characters. The characters that are allowed in *`macro_name`* are the same as are allowed for any valid VOS C identifier. There can be no white-space between *`macro_name`* and the opening parenthesis (`(`) that marks the start of the *`identifier_list`*. If a function-like macro has parameters, they are specified in the optional *`identifier_list`*. Each parameter must be uniquely declared within its scope. The scope of a parameter's identifier extends from its declaration in the *`identifier_list`* until the end of the `#define` preprocessor control line. In VOS C, a function-like macro can have not more than 32 parameters.

The optional *`replacement_list`* consists of one or more preprocessing tokens. Any white-space characters preceding or following the *`replacement_list`* are not considered part of the replacement list. A preprocessor control line using `#define` is terminated by the end of the line.

A function-like macro can be redefined by a subsequent `#define` directive provided the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical. Two replacement lists are identical if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

For information on defining a function-like macro a second time with a different definition than when the macro was first defined, see the "Undefining a Macro" section later in this chapter.

Function-like Macro Invocation

When a function-like macro is defined using the preceding syntax, each subsequent instance of the macro name followed by an opening parenthesis (`(`), an optional argument list, and a closing parenthesis (`)`) is an invocation of the macro. In determining where the argument list ends, matching pairs of parentheses that appear in the argument list are skipped. Similar syntactically to a function call, the syntax for a function-like macro invocation is as follows:

```
macro_name ([argument_list] )
```

The number of arguments in the macro invocation must match the number of parameters in the macro definition. You separate individual arguments within the argument list with commas, but a comma between a set of matching inner parentheses does not separate arguments. In VOS C, an argument in a function-like macro invocation cannot exceed 300 characters in length. If a newline character appears within the characters that make up a function-like macro's invocation, the newline is considered a normal white-space character.

Argument Substitution and Macro Expansion

When the preprocessor finds a function-like macro invocation, it uses the following procedure to substitute arguments and expand the macro invocation. Unless a parameter is preceded by the `#` operator, or preceded or followed by the `##` operator, the preprocessor replaces a parameter in the *`replacement_list`* with the value of the corresponding argument after all macros in the argument list have been expanded. In this expansion, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the source module. Then, the preprocessing tokens that constitute the macro invocation are replaced by the expanded *`replacement_list`*. With a function-like macro, this substitution and replacement process is the macro expansion.

The following example contains two function-like macros and an invocation of each macro.

```
#define max(a, b) ((a) > (b) ? (a) : (b))

#define prt_sqr(num)  printf("num squared = %d\n",((num)*(num)))

int x, y = 20, z = 10;

x = max(y, z);

prt_sqr(10);
```

When the preprocessor processes the preceding code, the macro invocations are expanded as follows:

```
int x, y = 20, z = 10;

x =  ((y) > (z) ? (y) : (z)) ;

printf("10 squared = %d\n",((10)*(10))) ;
```

As shown in the `prt_sqr` macro invocation, when one of the parameters of a function-like macro appears in a character-string literal in the replacement list, the VOS C preprocessor replaces the parameter name (`num`) with the token sequence specified in the corresponding argument (10). Thus, `"num squared = %d\n"` becomes `"10 squared = %d\n"`. By default, the VOS C preprocessor does **not** suppress parameter substitution inside a string that appears in a replacement list. This default behavior was common in compilers prior to the establishment of the ANSI C Standard.

> **Note:** To direct the compiler to substitute arguments as specified in the ANSI C Standard, you must explicitly specify the `ansi_rules` option in a `# pragma` preprocessor control line, or select the corresponding command-line argument.

Regardless of whether you select `ansi_rules` or `no_ansi_rules`, you can use the stringize operator (`#`) to substitute a macro argument for a parameter inside a string that appears in a macro's replacement list. See "The ANSI Rules Options" section later in this chapter for information on the `ansi_rules` and `no_ansi_rules` options.

Advantages and Disadvantages of Function-like Macros

Using a function-like macro has advantages and disadvantages over using a true function. Consider the `max` macro that was discussed in the preceding sections.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

One advantage of using a function-like macro such as `max` is that a macro, as opposed to a function, will work for any compatible types of the arguments. Another advantage is that inline code is generated and the overhead of a function call is avoided.

One disadvantage of using a function-like macro such as `max` is that the macro evaluates one or the other of its arguments a second time, generating any side effects twice. Another disadvantage is that the compiler may generate more code for the function-like macro if the macro is invoked several times. A third disadvantage is that you cannot take the address of a function-like macro because a macro has no address.

Operator Precedence Errors

When you use a function-like macro, it is possible that the macro will yield an unexpected result if you do not consider how the rules of operator precedence affect the expanded macro. The following example shows how an incorrectly written macro can yield an unexpected result.

```
#define double_it(x) x + x
```

When the preprocessor expands the expression `5 * double_it(2)`, the resulting preprocessor token sequence is as follows:

```
5 * 2 + 2
```

The rules of operator precedence specify that a multiplication operation is performed before an addition operation. Thus, the preceding expression yields the value 12. This is probably not the result expected from the expression `5 * double_it(2)`. To avoid these types of operator precedence errors in a function-like macro, you should parenthesize each parameter in the replacement list as well as the entire replacement list. For example:

```
#define double_it(x) ((x) + (x))
```

With the parenthesized macro definition, when the preprocessor expands the expression `5 * double_it(2)`, the resulting token sequence is as follows:

```
5 * ((2) + (2))
```

In the preceding example, the expanded macro yields the expected result (the value 20) because the parentheses override the normal operator precedence.

Rescanning and Further Replacement

After the preprocessor replaces each parameter in the *`replacement_list`* with the value of the corresponding argument, the resulting preprocessing token sequence is rescanned with all subsequent preprocessing tokens of the source module for more macro names to replace. With both object-like and function-like macros, after the macro is expanded, the rescanning for additional macro names starts again at the beginning of the expansion. In VOS C, it is an error if the rescanning of the expansion results in an occurrence of the macro name being replaced.

The following example illustrates how rescanning occurs.

```
#define minus(a, b) subtract(a, b)

#define subtract(a, b) ((a) - (b))
   .
   .
   .
minus(y, z)
```

The preprocessor expands and rescans the preceding macro in the following manner.

1. After the original macro invocation, `minus(y, z)`, is expanded, the resulting preprocessor token sequence is:

   ```
   subtract(y, z)
   ```

2. When the preceding preprocessor token sequence is rescanned, the final result is:

   ```
   ((y) - (z))
   ```

When a macro expansion yields a preprocessor token sequence that resembles a preprocessing directive, the preprocessor does **not** interpret the token sequence as a preprocessor directive.

The Stringize Operator

The stringize operator (`#`) immediately followed by a parameter name in a function-like macro's replacement list causes the following to occur: a single character-string literal containing the preprocessing token sequence of the corresponding argument is substituted for the `#` operator and the parameter name.

> **Note:** In VOS C, unless you specify the `ansi_rules` option in a `# pragma` or select the corresponding compiler argument, a parameter name within a character-string literal is replaced with the token sequence of the corresponding argument **regardless of whether the** `#` operator precedes the parameter name.

See the “Argument Substitution and Macro Expansion” section earlier in this chapter for information on how the VOS C preprocessor substitutes arguments within character-string literals.

When the `#` operator creates a character-string literal out of an argument’s tokens, each occurrence of white space between the argument’s preprocessing tokens becomes a single space character in the character-string literal. White space preceding the argument’s first token and following the argument’s last token is deleted. You can include or omit space around the `#` operator.

The following example shows how to use the `#` operator to create a character-string literal.

```
#pragma ansi_rules

#define PRINT_MAX(x, y) printf("Of " #x " and " #y ", the greater \
value is %d\n", (x > y ? x : y) )
   .
   .
   .
PRINT_MAX(  1000  , 100 +   100);
```

When the preprocessor processes the preceding text, the macro is expanded as follows:

```
printf("Of " "1000" " and " "100 +   100" ", the greater "
"value is %d\n", (1000 > 100 +   100 ? 1000 : 100 +   100) );
```

Because the compiler concatenates adjacent character-string literals, the output from the preceding `printf` function call is as follows:

```
Of 1000 and 100 +   100, the greater value is 1000
```

When you use the `#` operator to create a character-string literal, the preprocessor retains the original spelling of each preprocessing token in the corresponding argument except for special handling when the argument is a character constant or character-string literal. In these cases, the preprocessor inserts a `\| character before each " and |`character when either of these two characters appear in a character constant or character-string literal argument. Otherwise, the preprocessor retains the original spelling of each token in the argument when it creates the string.

The Concatenation Operator

In an object-like or function-like macro’s replacement list, the concatenation operator (`##`), immediately preceded or followed by a parameter name, causes the preprocessing token sequence of the corresponding argument to be substituted for the `##` operator and the parameter name and then concatenated with the adjacent preprocessing token.

With the `##` operator, the adjacent tokens are concatenated before the replacement list is re-examined for more macro names to replace. The `##` preprocessing token cannot appear at the beginning or end of a replacement list. You can include or omit space around the `##` operator.

The following example shows how to use the `##` operator to concatenate a token sequence and an argument's token sequence.

```
#define PRINT_STRING(num) printf("The string is %s\n", array ## num)
   .
   .
   .

PRINT_STRING(2);
```

When the preprocessor processes the preceding text, the `PRINT_STRING` macro is expanded as follows:

```
printf("The string is %s\n", array2);
```

Undefining a Macro

You can use the `#undef` preprocessor directive to undefine a macro that has previously been defined. The `#undef` directive has the following form:

```
#undef macro_name
```

The *`macro_name`* is the name of a previously defined macro. The `#undef` directive causes the specified name no longer to be defined as a macro. In the following example, when the `#undef` directive is processed, the macro `MAX_RECS` is no longer defined.

```
#define MAX_RECS 1000
   .
   .
   .
#undef MAX_RECS
```

The preprocessor diagnoses an attempt to define an object-like or function-like macro name a second time with a different definition than when the name was first defined. In the following code fragment, the second definition of `LINE_LENGTH` is diagnosed.

```
#define LINE_LENGTH 80
   .
   .
   .
#define LINE_LENGTH 25
```

However, in VOS C, the preprocessor will proceed with preprocessing as though an `#undef` specifying `LINE_LENGTH` appeared prior to the macro's redefinition. In the preceding example, after the second definition, each subsequent instance of `LINE_LENGTH` will be replaced by the value 25. The preprocessor ignores an `#undef` directive that specifies a name that is currently undefined. In this case, the preprocessor does not issue a diagnostic.

Predefined Macros

A *predefined macro* is a macro that the VOS C preprocessor automatically defines. When a macro is predefined, it will expand to a nonzero value. You can use these predefined macros to control conditional compilation or to provide information about the source module and the compilation.

Table 9-1 lists some of the macros that the preprocessor predefines.

Table 9-1. ANSI C Predefined Macros

Predefined Macro	**Expanded Value**
`__DATE__`	A character-string literal containing the date of the compilation in the form: `"`*`Mnn dd`*`"`
`__FILE__`	A character-string literal containing the name of the source module
`__LINE__`	A decimal integer constant giving the current source line number
`__TIME__`	A character-string literal containing the time of the compilation in the form: `"`*`hh:mm:ss`*`"`

In addition to the macros shown in Table 9-1, the VOS C preprocessor also predefines the following macros:

- `__VOS__` indicates the VOS operating system environment.
- `__PROTOTYPES__` indicates that the compiler supports function prototypes.
- Two or more macros indicate the processor family and processor type based on the value specified in the `processor` option in a `#pragma` preprocessor control line, or in the `-processor` command-line argument.

When you issue the `c_preprocess` or `c` command, the value specified in the `processor` option or the `-processor` command-line argument determines which macro names the preprocessor automatically defines. The preprocessor always predefines a macro name for the processor family:

- `__MC68K__` when the processor value specified is from the MC68000 family
- `__I860__` when the processor value specified is from the i860 family.

In addition to a macro name for the processor family, the preprocessor also predefines one or more additional values for the specific processor type within the family. Table 9-3 shows the macros that are predefined for each specific processor type as indicated by the value given in the `processor` option or the `-processor` command-line argument.

Table 9-2. Predefined Macros for Processor Values

Processor Value	**Corresponding Predefined Macro(s)**
`default`	Varies depending on the default system processor
`mc68000`	`__MC68000__`
`mc68010`	`__MC68010__`
`mc68010/arithmetic_processor`	`__MC68010__ and __MC68010_AP__`
`mc68020`	`__MC68020__`
`mc68020/mc68881`	`__MC68020__ and __MC68881__`
`mc68030`	`__MC68030__`
`mc68030/mc68882`	`__MC68030__ and __MC68882__`
`i80860`	`__I80860__`

In Table 9-2, the `default` value indicates a site-settable default system processor. Unless your system administrator has changed the default system processor, it is one of the following:

- If the current module uses a processor from the MC68000 processor family, the default system processor is the MC68000 (`mc68000`).
- If the current module uses the i860 processor, the default system processor is the i860 (`i80860`).

See the "Conditional Inclusion Examples" section later in this chapter for an example of how to use a predefined macro.

Source File Inclusion

An `#include` preprocessor directive causes the preprocessor to replace that preprocessor control line with the entire contents of the specified file. When a source module contains one or more `#include` directives, the preprocessor searches for each specified file and, if the file is found, incorporates it into the source module prior to compilation. The `#include` directive has the following form:

```
         ["file_name"]
#include [<file_name>]
```

The *`file_name`* is a full or relative path name for a file that the preprocessor is to incorporate into the source module. No spaces are allowed between *`file_name`* and its delimiters (< and > or `"` and `"`). The quotation marks **must** be used as delimiters whenever *`file_name`* is anything other than a simple file name. Typically, the < and > delimiters denote a system-supplied file such as the `stdio.h` header file, and the `"` and `"` delimiters denote a file written by the user. A source module cannot incorporate more than 999 include files through the use of the `#include` directive.

In addition, if an `#include` directive does not match either of the preceding forms, the directive can take the form:

```
#include preprocessing_tokens
```

The *`preprocessing_tokens`* are a sequence of preprocessing tokens that are processed as normal text is processed. Within *`preprocessing_tokens`*, each identifier currently defined as a macro is replaced by its expanded form. In this expansion, adjacent character-string literals are not concatenated into a single string. After expansion, the resulting `#include` directive must match one of the forms where a *`file_name`* was specified, with either < and > or `"` and `"` as delimiters for the *`file_name`*.

The following example illustrates how the preprocessor expands a relative path name when one is specified in an `#include` directive.

```
#include "include_dir>my_file.incl.c"
```

For the preceding `#include` directive, the preprocessor determines the file's full path name by expanding the specified path name relative to the current directory. If the current directory were `%s1#d01>Sales>John_Jones`, the `#include` directive causes a file having the following full path name to be searched for and, if found, incorporated into the source module.

```
%s1#d01>Sales>John_Jones>include_dir>my_file.incl.c
```

When you specify a full or relative path name in an `#include` directive, the given path name must locate the file. The preprocessor does **not** search in any other location for the file.

In contrast, to find a file specified by a simple file name, the preprocessor looks in the directories specified in the process's include library. The *include library* for a process consists of an ordered sequence of path names for one or more directories in which the preprocessor searches for include files. Each path name in the include library specifies a directory containing one or more include files. For example, if you specify the name of a VOS C header file such as `stdio.h`, the preprocessor searches the directories of the include library for the header.

The process's include library consists of those directories listed in *`master_disk`*`>system>include_library` unless you have redefined the include library paths for your process. In the set of directories that make up the process's include library, the binder searches the directories in the order in which they are listed. You can use the `list_library_paths` command to display the directory path names for your process's include library. See the *VOS Commands User's Guide (R089)* for a complete description of search rules.

When you specify an `#include` directive, the following rules determine the form of the file name for which the preprocessor searches.

- If the file has a `.incl.c` suffix, the preprocessor searches for a file of the given name with the `.incl.c` suffix.

- If the file has a `.h` suffix, the preprocessor searches for a file of the given name with the `.h` suffix. If the file is not found with the `.h` suffix, the preprocessor searches for a file of the given name with the `.incl.c` suffix.
- If the file has no suffix, the preprocessor searches for a file having no suffix, then for a file having the `.incl.c` suffix, and finally for a file having the `.h` suffix.

The preceding rules apply whether you specify a full or relative path name or a simple file name in the `#include` directive. If the preprocessor does not find a file, it issues an error message.

Conditional Inclusion

The C preprocessor directives allow you to use conditional inclusion in a source module. *Conditional inclusion* occurs when, based on the value of a constant expression, the preprocessor processes or does not process lines of text in the source file. For example, using conditional inclusion allows you to process and incorporate into the source module differing versions of an include file whose contents vary depending on the environment in which the program module will run.

The following sections explain how to use preprocessor directives to control conditional inclusion.

The `# if`, `# elif`, `# else`, and `# endif` Directives

The `#if` and `#elif` preprocessor directives control conditional inclusion. The if-group in which these directives can appear has the following form:

```
# if constant_expression

 [source_text]

 [# elif constant_expression ]
 [          # else           ]

 [source_text]

  #endif
```

Each *`constant_expression`* is an integral constant expression except for the following:

- The *`constant_expression`* cannot contain a cast.
- The *`constant_expression`* can contain an expression that uses the `defined` operator.
- An identifier within the *`constant_expression`* is interpreted as described in the "Constant Expression Interpretation" section later in this chapter.

An identifier within the *`constant_expression`* is a preprocessing token: either a defined macro or not a defined macro. At this phase of the translation, there are no keywords such as `sizeof`, enumeration constants, and so forth.

The *`source_text`* can contain zero or more lines of source text, including lines with other preprocessor directives. To determine whether or not to process *`source_text`*, the preprocessor checks, in order, each `#if` and `#elif` directive's *`constant_expression`*.

- If *`constant_expression`* evaluates to true (nonzero), the text that it controls is processed.
- If *`constant_expression`* evaluates to false (0), the text that it controls is not processed. In addition, one of the following can occur:
  - When an `#if` or `#elif` directive is followed by an `#else` directive, the text controlled by the `#else` directive is processed if the constant expression specified in the `#if` or `#elif` directive is false.
  - When an `#if` directive is followed by an `#elif` directive, the text controlled by the `#elif` directive is processed if the constant expression specified in the `#elif` directive is true.

An `#endif` directive is required for each `#if` directive.

An if-group can include more than one nested `#elif` directive. The `#elif` directive is a combination of an `#else` directive and an `#if` directive. After the preprocessor evaluates the `#elif` directive's constant expression, the preprocessor processes or does not process the text controlled by `#elif` based on the value of that constant expression. An `#elif` directive can be used only within an if-group or another `#elif` directive.

A preprocessing token cannot appear after the `#else` or `#endif` directive and before the end of the line that terminates the control line containing either of these directives.

The `defined` Operator

In the constant expression of an `#if` or `#elif` directive, you can use the `defined` operator to determine whether a macro name is defined or not defined. The `defined` operator has the following form:

```
defined [(] macro_name [)]
```

The *`macro_name`* is a macro name. The expression containing `defined` and an identifier evaluates to the value 1 if the identifier is currently defined as a macro name. Otherwise, the expression evaluates to the value 0 if the identifier is not currently defined as a macro name. An identifier is currently defined as a macro name if it was predefined by the compiler, or if it was the subject of a `#define` preprocessor directive and was not undefined with an intervening `#undef` directive.

Constant Expression Interpretation

In the `#if` and `#elif` directives, the constant expression that controls conditional inclusion is interpreted in the following manner. Prior to evaluating the constant expression, the preprocessor expands macros in the preprocessing tokens that comprise the constant expression **except** it does not expand macros that are the operands of the `defined` operator. The preprocessor evaluates any `defined` operator expressions.

After the preprocessor expands macros and evaluates `defined` expressions, all remaining identifiers are replaced with the value 0. Then, the resulting tokens are interpreted as C language tokens and evaluated using the same rules as are used for a C language constant expression. See the "Constant Expressions" section in Chapter 7 for information on these expressions.

In VOS C, a character constant in a controlling constant expression has the same value as a character constant that appears in any C language expression. For example, the following constant expressions, one in an `#if` directive and one in an `if` statement, evaluate to true (nonzero).

```
#if 'z' - 'a' == 25

if ('z' - 'a' == 25)
```

Conditional Inclusion Examples

The following example shows how to use preprocessor directives to control conditional inclusion.

```
#pragma processor (mc68010)

#define SPECIAL

#if  defined (__VOS__)                          /*  Example 1  */
#define $LONGMAP $longmap
#define $SHORTMAP $shortmap
#else
#define $LONGMAP        /*  In environments other than VOS, $LONGMAP
*/
#define $SHORTMAP       /*  and $SHORTMAP expand to zero characters
*/
#endif

#if defined(__MC68K__) && !defined(SPECIAL)     /*  Example 2  */
#include "mc68K_structs.incl.c"
#elif defined(__I860__) && !defined(SPECIAL)
#include "i860_structs.incl.c"
#else
#include "special_structs.incl.c"
#endif
```

The VOS preprocessor predefines the `__VOS__` macro. In addition, the preprocessor predefines either the `__MC68K__` or `__I860__` macro depending on the value given in a `#pragma` specifying the `processor` option, or the value selected in the corresponding compiler argument. Therefore, when example 1 is processed, the preprocessor would define `$SHORTMAP` as `$shortmap`, and `$LONGMAP` as `$longmap`. When example 2 is processed, the preprocessor would incorporate, into the source module, only the `special_structs.incl.c` include file because the expression `!defined(SPECIAL)` evaluates to false (0).

See the “Predefined Macros” section earlier in this chapter for more information on predefined macros.

The `# ifdef` and `# ifndef` Directives

The `#ifdef` and `#ifndef` preprocessor directives are combinations of the `#if` directive and the `defined` operator that allow you to process source text based on whether or not a macro name is defined. The `#ifdef` and `#ifndef` directives have the following form:

```
#ifdef macro_name

#ifndef macro_name
```

These two directives, respectively, are the equivalent of and produce the identical results as the following `#if` directives:

```
#if defined macro_name

#if ! defined macro_name
```

The following examples show how to use `#ifdef` and `#ifndef` directives to control conditional inclusion.

```
#ifdef __VOS__                                  /*  Example 1  */
#define $LONGMAP $longmap
#define $SHORTMAP $shortmap
#else
#define $LONGMAP         /*  In environments other than VOS, $LONGMAP
*/
#define $SHORTMAP        /*  and $SHORTMAP expand to zero characters
*/
#endif

#ifndef __PROTOTYPES__                          /*  Example 2  */
#include "nonprototyped_declarations"
#else
#include "prototyped_declarations"
#endif
```

In the preceding examples, example 1 uses the `#ifdef` directive but otherwise is the equivalent of example 1 in the earlier section, “Conditional Inclusion Examples.” If they were compiled under the same conditions, both versions of example 1 would produce identical results.

Example 2 makes use of the `#ifndef` directive and the `__PROTOTYPES__` predefined macro to determine which version of an include file to incorporate into the source module. If a version of the VOS C compiler supports function prototypes, the preprocessor predefines the `__PROTOTYPES__` macro. In example 2, when `__PROTOTYPES__` is not defined, the `nonprototyped_declarations` include file is incorporated into the source module.

When `__PROTOTYPES__` is defined, the `prototyped_declarations` include file is incorporated into the source module.

The # `pragma` Directive

The `#pragma` preprocessor directive allows you to specify certain VOS C options that cause the compiler to perform particular tasks. Some `#pragma` options provide the compiler with information that you cannot supply with command-line arguments. Other `#pragma` options correspond to command-line arguments for the `c` command.

The following `#pragma` options have corresponding command-line arguments: `ansi_rules`, `check_enumeration`, `default_char`, `expanded_macros`, `extension_checking`, `linted`, `mapcase`, `mapping_rules`, `promotion_rules`, `processor`, `registers`, and `type_checking`. For each of these options, the value given in a `#pragma` directive overrides the value specified on the command line. For each of these options, the action taken by the compiler is identical regardless of whether you specify the information in a `#pragma` directive or in a command-line argument. Except for `expanded_macros`, the corresponding command-line argument has the same name as the `#pragma` option but with the `-` prefix added. The command-line argument that corresponds to the `expanded_macros` option is `-show_macros` with the `expanded` value.

Some options can be specified more than once within a source module. The effect of the option applies to all subsequent source text unless another such `#pragma` specifying a different value is specified. Thus, you can enable and disable an option, controlling the area of source text over which the action applies. See “The Registers Options” section later in this chapter for an example of how to enable and disable an option.

The following options can be specified multiple times within a source module.

- `ansi_rules` and `no_ansi_rules`
- `check_enumeration` and `no_check_enumeration`
- `expanded_macros` and `no_expanded_macros`
- `extension_checking`
- `linted` and `no_linted`
- `list` and `no_list`
- `mapcase` and `no_mapcase`
- `registers` and `no_registers`
- `support` and `no_support`
- `type_checking`
- `untyped_storage_sharing` and `no_untyped_storage_sharing`

A `#pragma` specifying an option **not** in the preceding list must appear before any data declarations or function definitions, and can appear only once in the source module.

Pragma Syntax

The syntax for a `#pragma` preprocessor control line is as follows:

```
#pragma option [ ( value ...) ]
```

The allowed values for *`option`* and *`value`* are explained in the sections that follow. In addition, the "Preprocessor Directives Summary" section at the end of this chapter summarizes the options that are available in VOS C.

As with any preprocessor directive, the `#` character must be the first non-white-space character on the line. The parentheses around *`value`* are optional. If more than one *`value`* is allowed with an *`option`*, use a comma to separate the values. Each `#pragma` preprocessor control line is terminated by the end of the line.

The compiler does not diagnose unrecognized pragmas or invalid syntax following the `pragma` directive unless you specify the `system_programming` option in a `#pragma`, or select the corresponding command-line argument. The compiler does issue a diagnostic if you specify a valid *`option`* but an invalid *`value`* for that option.

The ANSI Rules Options

The `ansi_rules` and `no_ansi_rules` options specify whether the compiler suppresses parameter substitution inside character-string literals that appear in macro-definition lines. Protecting strings from parameter substitution is the behavior specified in the ANSI C Standard. The syntax for the ANSI rules options is as follows:

```
        ┌  ansi_rules     ┐
#pragma │                 │
        └  no_ansi_rules  ┘
```

If you specify `ansi_rules`, the compiler suppresses parameter substitution inside character-string literals that appear in macro-definition lines. When you specify `ansi_rules` (or `no_ansi_rules`), you can use the stringize operator (`#`) to substitute a macro argument for a parameter inside a string that appears in a macro definition.

> **Note:** To direct the compiler to expand macros as specified in the ANSI C Standard, you must explicitly select the `ansi_rules` option or the corresponding command-line argument.

If you specify `no_ansi_rules`, the compiler does not suppress parameter substitution inside character-string literals that appear in macro-definition lines. By default, the VOS C compiler does **not** suppress parameter substitution inside these strings. The default behavior was common in compilers prior to the establishment of the ANSI C Standard.

See the "Argument Substitution and Macro Expansion" section earlier in this chapter for more information on and examples of macro parameter substitution.

The Bit-Field Allocation Options

In VOS C, values that are specified with the bit-field allocation `#pragma` options control the three factors that determine how bit fields are allocated within a structure or union. These three factors are as follows:

- *Bit-field size* determines the amount of space that the compiler reserves when it needs to allocate storage for bit fields in a structure or union. By default, the compiler reserves four bytes (an `int`-size space).

- *Bit-field alignment* determines the type of boundary where space for bit fields can begin. By default, the compiler begins allocating bit fields on even-numbered byte boundaries.
- *Bit-packing direction* determines whether the compiler starts allocating bits from the left or right side of the reserved space. By default, the compiler begins allocating bits from the left side of the reserved space.

Three `#pragma` options allow you to change the default values for these three factors and, therefore, to control all aspects of bit-field allocation. The syntax for these `#pragma` options is as follows:

```
#pragma bit_field_size ( size )

#pragma bit_field_align ( type )

#pragma bit_packing ( direction )
```

With these options, you can change the default allocation method used by the compiler. Table 9-3 describes these options and their allowed values.

Table 9-3. Pragma Options for Bit-Field Allocation

Option	**Purpose**	**Allowed Values**

`bit_field_size`	The *`size`* value determines the amount of space that the compiler reserves when it needs to allocate storage for bit fields. The default value for *`size`* is `int`.	`char`
		`short`
		`int`

`bit_field_align`	The *`type`* value determines the type of boundary where space for bit fields can begin. The default value for *`type`* is `short` (aligned on an even-numbered byte boundary).	`char`
		`short`
		`int`

`bit_packing`	The *`direction`* value determines whether the compiler begins allocating bits from the left or right side of the reserved space. The default value for *`direction`* is `left_to_right`.	`left_to_right` `right_to_left`

The compiler generates an error message if you try to specify an individual bit field that is larger than the size specified in `bit_field_size`.

See the "Bit-Field Data" section in Chapter 4 for more information on and examples of bit-field allocation.

The Common Constants Options

The `common_constants` and `no_common_constants` options specify whether the compiler allocates character-string literals in the code region or the static region. A program cannot overwrite character-string literals that are stored in the program's code region. The syntax for the common constants options is as follows:

```
        ┌   common_constants   ┐
#pragma │                      │
        └ no_common_constants  ┘
```

If you specify `common_constants`, the compiler allocates character-string literals in the program's code region. Therefore, trying to overwrite a character-string literal is an attempt to modify a protected page, which causes an error condition to occur. By default, the compiler stores character-string literals in the code region.

If you specify `no_common_constants`, the compiler allocates character-string literals in the program's static region. Therefore, the program can overwrite a character-string literal. Overwriting a character-string literal is not considered a good programming practice.

The `default_char` Option

The `default_char` option specifies whether `char` data items that are declared without an explicit `signed` or `unsigned` keyword are signed or unsigned. The syntax for the `default_char` option is as follows:

```
#pragma default_char ( type )
```

The allowed values for *`type`* are `signed` and `unsigned`.

If you specify `signed`, all `char` data items that are declared without an explicit `signed` or `unsigned` keyword are signed.

If you specify `unsigned`, all `char` data items that are declared without an explicit `signed` or `unsigned` keyword are unsigned. By default, the compiler assumes that `char` types without an explicit `signed` or `unsigned` have the type `unsigned char`.

The `default_char_set` Option

The `default_char_set` option specifies whether the compiler stores character-string literals as canonical strings or as common strings. The syntax for the `default_char_set` option is as follows:

```
#pragma default_char_set ( character_set )
```

The allowed values for *`character_set`* are `none` and `latin_1`. By default, the compiler uses Latin alphabet No. 1 (`latin_1`) as the default character set.

If *`character_set`* is `none`, the compiler stores every character-string literal in the source module as a canonical string. That is, each right graphic single-byte or double-byte character is preceded by a single-shift character. The character-string literal has no default character set and no locking-shift characters. When you specify `none` as the default character set, it is similar to calling the `$shift` built-in for every character-string literal (*`string_literal`*) in the source module, as in the following function invocation:

```
$shift( "string_literal", strlen("string_literal") )
```

Specifying `none` in the `default_char_set` option is equivalent to the preceding call except the result evaluates to a constant value. See the "`$shift`" section in Chapter 12 for more information on that built-in function.

If *`character_set`* is `latin_1`, the compiler stores every character-string literal in the source module as a common string. That is, any right graphic character that is not preceded by a single-shift character is assumed to be a Latin alphabet No. 1 character. Explicitly specifying `latin_1` as the default character set is similar to not specifying any default character set because, by default, the compiler stores character-string literals as common strings. However, when you explicitly specify `latin_1` in the `default_char_set` option, the compiler checks that each string literal is a valid NLS string.

See the *National Language Support User's Guide (R212)* for information on National Language Support and NLS strings.

The Default Mapping Options

The `default_mapping` and `no_default_mapping` options specify whether the compiler diagnoses the absence of a `$shortmap` or `$longmap` specifier in a structure or union declaration. The syntax for the default mapping options is as follows:

```
        ┌   default_mapping    ┐
#pragma │ no_default_mapping   │
        └                      ┘
```

If you specify `default_mapping`, the compiler does **not** issue a diagnostic when you do not use a `$shortmap` or `$longmap` specifier in the definition of a `struct` or `union` tag, or in the definition of a `struct` or `union` object that is not a tagged type. By default, the compiler does not issue a diagnostic for such a declaration.

If you specify `no_default_mapping`, the compiler issues a diagnostic when you do not use a `$shortmap` or `$longmap` specifier in the definition of a `struct` or `union` tag, or in the definition of a `struct` or `union` object that is not a tagged type. The following program fragment illustrates the types of declarations that `no_default_mapping` diagnoses.

```
#pragma no_default_mapping

struct $shortmap tag_1       /*  Example 1:  declares a struct tag  */
   {
   char c;
   int i;
   };

struct $shortmap             /*  Example 2:  declares a struct      */
   {                         /*  object without using a tag         */
   char c;
   int i;
   } struct_var_1;

struct tag_1 struct_var_2;   /*  Example 3:  declares a struct      */
                             /*  object using a tag                 */

struct tag_2                 /*  Example 4:  declares a struct tag  */
   {
   char c;
   int i;
   };

struct                       /*  Example 5:  declares a struct      */
```

```
{                        /*  object without using a tag       */
char c;
int i;
} struct_var_3;
```

Because the preceding program fragment contains a `#pragma` specifying `no_default_mapping`, the compiler issues diagnostics for the two declarations (examples 4 and 5) where no alignment method is indicated.

The compiler does not issue diagnostics for the declarations in examples 1 through 3. Once a tagged type, such as `struct tag_1` in example 1, has an alignment specifier associated with it, all subsequent `struct` declarations that use the tagged type are aligned according to that set of alignment rules, and the compiler issues no diagnostic for these declarations. Therefore, the compiler does not diagnose the `struct` declaration in example 3 even though the declaration has no explicit alignment specifier. In fact, in example 3, it would be invalid to specify an alignment specifier for a `struct` tag that has already been defined.

For information on using the `$shortmap` and `$longmap` alignment specifiers, see the "Alignment Specifiers" section in Chapter 3.

The Enumeration Options

The `check_enumeration` and `no_check_enumeration` options specify whether the compiler issues a diagnostic for an implicit conversion involving an enumeration data item. These implicit conversions occur when an enumeration object or constant is:

- assigned to an object of a different type
- compared against an object or constant of a different type
- used as the operand of an arithmetic operator.

The syntax for the enumeration options is as follows:

```
        ┌    check_enumeration   ┐
#pragma │                        │
        └  no_check_enumeration  ┘
```

If you specify `check_enumeration`, implicit conversions involving enumeration items are diagnosed, as described in the preceding discussion. By default, the compiler does not issue these diagnostics.

If you specify `no_check_enumeration`, enumerated data is treated as if it were defined as the `int` type. In this case, implicit conversions involving enumeration items are diagnosed as though the enumeration item has the `int` type.

> **Note:** Regardless of whether you select `check_enumeration` or `no_check_enumeration`, the compiler does not diagnose implicit conversions involving enumeration items if you specify `none` as the level of data-type checking in the `type_checking` option or in the corresponding command-line argument.

See the "Enumerated Types" section in Chapter 4 for examples of the types of operations that are diagnosed by the `check_enumeration` option.

The Expanded Macros Options

The `expanded_macros` and `no_expanded_macros` options specify whether the compiler expands subsequent macros in a `.list` file. When you select the `c` command's `-list` argument, the compiler generates a compilation listing with the `.list` suffix. The syntax for the expanded macros options is as follows: \noplist.`list` fileCompilation listings

```
         ┌    expanded_macros   ┐
#pragma  │                      │
         └  no_expanded_macros  ┘
```

If you specify `expanded_macros`, the compiler expands subsequent macros when it generates a `.list` file.

If you specify `no_expanded_macros`, the compiler does not expand subsequent macros when it generates a `.list` file. By default, the compiler does not expand macros in a `.list` file.

The `extension_checking` Option

The `extension_checking` option specifies the level of extension checking that the compiler uses to diagnose VOS C language extensions that can affect program transportability. VOS C allows certain programming practices that are not allowed in ANSI C. The syntax for the `extension_checking` option is as follows:

```
#pragma extension_checking ( level )
```

The allowed values for *`level`* are `none`, `minor`, and `all`. By default, the compiler uses `none`.

If you specify `none` as the level of extension checking, the compiler does not perform any checks for VOS C extensions.

If you specify `minor` as the level of extension checking, the compiler produces warnings when you use the following VOS C language extensions:

- the declaration of an anonymous data item (for example, a `struct` or `union` with no name)
- a partially qualified reference to a `struct` member (for example, referring to a structure member `ex_struct.name` as `name`)
- 2-byte character constants
- the use of the address-of operator (`&`) with a constant value in an argument list
- an external array definition for which the number of elements is not specified (thereby implying an `extern` declaration)
- an expression yielding a nonaddress used in a context where an address is required
- an undeclared identifier implicitly defined as `int`.

If you specify `all` as the level of extension checking, the compiler includes all `minor` diagnostics and produces warnings when you use the following VOS C language extensions:

- a built-in function, such as `$substr`
- the Forms Management System `accept` or `screen` statement
- the declaration of a `char_varying` data item
- the use of a generic, string-manipulation function, such as `strcat`, with a `char_varying` argument.

The extensions diagnosed when you specify `all` as the level of extension checking include constructs that are more obviously nonportable. In contrast, the extensions that the compiler diagnoses when you specify `minor` as the level of extension checking are more subtle and more likely to be used unintentionally.

The Linted Options

The `linted` and `no_linted` options specify whether the compiler generates extra code for the `return` statement to load both the address and data registers, disregarding the definition of the function containing the `return` statement. You can use the `linted` option to tell the compiler that any such additional code can be eliminated because one of the following has occurred.

- The program has been linted with the `lint` command and all mismatches between the function return types specified in the declaration and definition of a function have been corrected.

- The programmer has checked that the function return types specified in the declaration and definition of a function are consistent in the source modules comprising the program.

When the `linted` option is specified, the compiler can generate better code for `return` statements.

The syntax for the linted options is as follows:

```
        ┌    linted    ┐
#pragma │              │
        └  no_linted   ┘
```

If you specify `linted`, the compiler assumes that all source modules comprising the program module consistently declare and define the return type of each function, as is required in every correct C program. The compiler does not generate extra code for every `return` statement.

If you specify `no_linted`, the compiler does not assume that all source modules comprising the program module consistently declare and define the return type of each function. The compiler does generate extra code for every `return` statement. By default, the compiler generates the extra code.

Note: In the VOS environment, the `lint` command is available only on those systems that have the USF product installed.

The List Options

The `list` and `no_list` options tell the compiler whether subsequent source text should be included in any `.list` file. When you select the `c` command's `-list` argument, the compiler generates a compilation listing with the `.list` suffix. The syntax for the list options is as follows:

```
        ┌    list    ┐
#pragma │            │
        └  no_list   ┘
```

If you specify `list`, the compiler includes, in the `.list` file, any subsequent source text unless you later use the `nolist` option in a `#pragma`, or use the `#no_list` directive.

If you specify `no_list`, the compiler does **not** include, in the `.list` file, any subsequent source text unless you later use the `list` option in a `#pragma`, or use the `#list` directive.

The Mapcase Options

The `mapcase` and `no_mapcase` options specify whether the compiler interprets uppercase letters differently from lowercase letters. You can make the compiler appear to be case insensitive by specifying the `mapcase` option. The syntax for the mapcase options is as follows:

```
        ┌    mapcase    ┐
#pragma │               │
        └  no_mapcase   ┘
```

If you specify `mapcase`, the compiler interprets uppercase letters as their lowercase counterparts, except for uppercase letters in character constants and character-string literals.

If you specify `no_mapcase`, the compiler interprets uppercase letters differently from their lowercase counterparts. By default, the compiler is case sensitive.

When you compile a source module with the `mapcase` option or the corresponding command-line argument, and the code contains an external variable name or function name with one or more uppercase letters, you **may not be able to bind** the resulting object module with another object module that defines the same external variable or function and that has not been compiled with the `mapcase` option. If the binder encounters a reference to the original name, for example in a binder control file, it will not recognize the original name and its lowercase version as the same name.

The `mapping_rules` Option

The `mapping_rules` option specifies the data alignment rules that the compiler uses when laying out storage for the source module. The syntax for the `mapping_rules` option is as follows:

```
#pragma mapping_rules ( mapping[, checking] )
```

The allowed values for *`mapping`* are `longmap` and `shortmap`. If you specify `longmap`, the compiler uses the longmap alignment rules. If you specify `shortmap`, the compiler uses the shortmap alignment rules. By default, the compiler uses the system-wide default data alignment method. The default method is site-settable.

The allowed values for *`checking`* are `check` and `no_check`. If you specify `check`, the compiler diagnoses alignment padding within structures. If you specify `no_check`, the compiler does not diagnose alignment padding within structures. By default, the compiler does not diagnose such alignment padding.

Specifying a data alignment method through the use of a `#pragma` directive or an alignment specifier such as `$longmap` overrides the data alignment method indicated in the `-mapping_rules` command-line argument. However, the compiler still diagnoses alignment padding within structures if you have used a `#pragma` specifying the `check` value in the `mapping_rules` option.

See the "Data Alignment" section in Chapter 5 for information on data alignment methods.

The `processor` Option

The `processor` option specifies the processor on which the code will run. Currently, Stratus modules use processors from one of the following processor families:

- MC68000
- i860.

On some XA400 and XA600 modules, the Stratus Arithmetic Processor is used as an arithmetic processor. i860 processorProcessors MC68000 processorProcessors Stratus Arithmetic ProcessorProcessors

You select a value for the `processor` option based on the processor type(s) found in the module where the code is to execute. If you do **not** select the `processor` option, the compiler generates code that will run on the default system processor. Unless your system administrator has changed the default system processor, it is one of the following:

- If the current module uses a processor from the MC68000 processor family, the default system processor is the MC68000.

- If the current module uses the i860 processor, the default system processor is the i860.

When you use the `processor` option or the corresponding command-line argument and specify a particular processor type, the compiler creates object code customized for the indicated processor. As a result, the code runs faster on the specified processor, but **cannot**, in some cases, run on alternate processors.

The syntax for the `processor` option is as follows:

```
#pragma processor ( processor_string )
```

Table 9-4 shows the allowed values for *`processor_string`* for the MC68000 processor family. In addition, the table lists the processor indicated by each value and the Stratus modules in which the processor is found. Only programs compiled with the `mc68000` value will run on any MC68*xxx* processor.

Table 9-4. Values for the MC68000 Processor Family

Value	Processor	Model
`default`	default system processor	default

Table 9-4. Values for the MC68000 Processor Family

Value	Processor	Model
`mc68000`	MC68000 or MC68010 with or without an arithmetic processor	XA400
`mc68010`	MC68010 with or without an arithmetic processor	XA400
`mc68010 /arithmetic_processor`	MC68010 with an arithmetic processor	XA600
`mc68020`	MC68020 with or without the MC68881 co-processor	XA2000 Model 100
`mc68020/mc68881`	MC68020 with the MC68881 co-processor	XA2000 Model 110 to XA2000 Model 160
`mc68030`	MC68030 with or without the MC68882 co-processor	XA2000 Model 30 and XA2000 Model 200 to XA2000 Model 260
`mc68030/mc68882`	MC68030 with the MC68882 co-processor	XA2000 Model 30 and XA2000 Model 200 to XA2000 Model 260

Table 9-5 shows the allowed values for *`processor_string`* for the i860 processor family. In addition, the table lists the processor indicated by each value and the Stratus modules in which the processor is found.

Table 9-5. Values for the i860 Processor Family

Value	Processor	Model
`default`	i860	XA/R Model 20
`i80860`	i860	XA/R Model 20

Depending on the value specified in the `processor` option or the corresponding command-line argument, the compiler automatically defines one macro for the processor family and one or more macros corresponding to the processor type(s). See the “Predefined Macros” section earlier in this chapter for information on these predefined macros.

You can compile code on a module containing a different processor than that on which the code will run. *Cross compilation* occurs when a compiler running on a processor from one processor family translates a source module into object code that will execute on a processor of a different processor family. See the *VOS C User’s Guide (R141)* for information on cross compilation.

The `promotion_rules` Option

The `promotion_rules` option specifies the promotion rules that the compiler uses when performing certain data-type promotions. The syntax for the `promotion_rules` option is as follows:

```
#pragma promotion_rules (promotion[, checking])
```

The allowed values for *`promotion`* are `ansi` and `non_ansi`. Table 9-5 summarizes the effects of these two values. By default, the compiler uses the ANSI promotion rules (that is, `ansi`). The `ansi` value tells the compiler to use the promotion rules that are defined in the ANSI C Standard.

Table 9-6. Values for the `promotion_rules` Option

Value	Effect on Data Type Conversions
`ansi`	In integral promotions, `unsigned char` and `unsigned short` values are promoted to `signed int`. The variable's value is preserved. In addition, `float`-to-`double` promotion does **not** occur before `float` data participates in an expression. The `ansi` value is the default.
`non_ansi`	In integral promotions, `unsigned char` and `unsigned short` values are promoted to `unsigned int`. The variable's unsignedness is preserved. In addition, `float` data is promoted to `double` before participating in an expression. This was the method used by some traditional compilers for arithmetic conversions.

Note: For all explanations in the table, the integral promotions described for the `unsigned char` type apply to bit fields having 8 bits or less. Also, integral promotions described for the `unsigned short` type apply to bit fields having more than 8 bits.

The allowed values for *`checking`* are `check` and `no_check`. If you specify `ansi, check`, the compiler issues diagnostics for certain situations where specifying `non_ansi` rather than `ansi` could cause differences in program behavior. If you specify `ansi, no_check`, the compiler does not issue these diagnostics. By default, the compiler does not issue the diagnostics. The `check` and `no_check` values have **no effect** when specified with `non_ansi`.

The `ansi, check` value tells the compiler to use ANSI C's "value preserving" promotion rules **and** to diagnose certain situations that could cause differences in program behavior. The diagnosed situations include code where incompatibilities could arise as a result of selecting ANSI C's promotion rules.

If a program was previously compiled with the `non_ansi` value for `promotion_rules`, it is unlikely that recompiling with the `ansi` value will affect the program's behavior. With the ANSI C promotion rules, `unsigned char` and `unsigned short` variables are promoted to `signed int` when widened by integral promotions. In addition, the variable's value is preserved. A program that was previously compiled with the `non_ansi` value will be affected by recompiling with the `ansi` value only if both of the following conditions are true.

- The program includes an expression containing a bit field, `unsigned char`, or `unsigned short` that produces a 32-bit result (that is, the size of an `int`).
- The sign bit of the result is set **and**, after evaluation, the result of the expression is used in such a way that its signedness is significant.

Signedness is significant when the `unsigned short` variable is used as the left operand of the right-shift operator (`>>`), or as either operand of one of the following operators: `/`, `%`, `<`, `<=`, `>`, or `>=`.

For example, consider the following program fragment in which signedness is significant.

```
unsigned short us = 65535;
signed int i = -1;

if (us > i)
   printf("ANSI (value preserving) promotion rules.\n");
else
   printf("Non-ANSI (unsigned preserving) promotion rules.\n");
```

In the preceding example, the ANSI and non-ANSI promotion rules yield differing results. With ANSI C promotion rules, `us` is greater than `i`. Before evaluating the expression `us > i`, the compiler promotes `us` to `signed int`. In addition, the value of `i` is preserved. With

In contrast, with non-ANSI promotion rules, `us` is **not** greater than `i`. Before evaluating the expression `us > i`, the compiler promotes `us` to `unsigned int`. Next, because one value is unsigned, the compiler promotes the other to `unsigned int`. With the non-ANSI promotion rules, the value of `i` is not preserved. With `unsigned int` data, `0x0000FFFF` (65,535 decimal) is not greater than `0xFFFFFFFF` (4,294,967,295 decimal). `signed int` data, `0x0000FFFF` (65,535 decimal) is greater than `0xFFFFFFFF` (-1 decimal).

The Registers Options

The `registers` and `no_registers` options specify whether the compiler assigns to machine registers any data items declared with the `register` storage class specifier. The syntax for the registers options is as follows:

```
        ┌  registers   ┐
#pragma │              │
        └ no_registers ┘
```

If you specify `registers`, the compiler attempts to allocate machine-register space to data items defined with the `register` storage class specifier. By default, the compiler attempts to allocate these items in registers.

If you specify `no_registers`, the compiler does not allocate machine-register space to data items defined with the `register` storage class specifier. Specifying `no_registers` allows you to suppress the assignment of registers for `register` data items without changing any data declarations in the source module.

Note: When you specify local or global register allocation as an optimization and the `registers` option, the compiler does not necessarily assign the `register` data item to a machine register. The `register` specifier in a declaration acts as an advice, which the compiler uses in its register allocation algorithm. In contrast, when you specify local or global register allocation and the `no_registers` option, the compiler does not consider this advice in its register allocation algorithm.

The following example illustrates how the register options can be specified multiple times in a source module.

```
#pragma no_registers           /*  The registers option is disabled
*/

float func(register float score)
{
   register float multiplier;
   register int i;

#pragma registers              /*  The registers option is enabled
*/

   for (i = 0; i > 999; i++)
        {
        register int count = 0;
            .
            .
            .
        }
}
```

In the preceding example, the `no_registers` option is specified for the declaration of the parameter, `score`, and for the declaration of the first two variables in the function definition, `multiplier` and `i`. Then, the `registers` option is specified for the declaration of `count` in the compound statement associated with the `for` statement. Unless there were a subsequent `no_registers` option, the `registers` option is also specified for any later data declarations in the source module. Using the `no_registers` and `registers` options in this manner effectively disables and then enables the option.

The Support Options

The `support` and `no_support` options tell the compiler whether functions within the source module will be used as system support routines. With a system support routine, the debugger's `trace` request does not display the stack frame associated with the function. The syntax for the support options is as follows:

```
        ┌   support    ┐
#pragma │              │
        └  no_support  ┘
```

If you specify `support`, the compiler identifies (for the debugger) all functions within the source module as system support routines. The debugger will not display the function's name when you issue a `trace` request unless you specify the `-all` argument.

If you specify `no_support`, the compiler does not identify (for the debugger) all functions within the source module as system support routines. The debugger will display the function's name when you issue a `trace` request. By default, the compiler does not treat functions as system support routines.

The System Programming Options

The `system_programming` and `no_system_programming` options specify whether the compiler performs three checks commonly needed for programs used in VOS C system programming. When `system_programming` is specified, the compiler issues diagnostics for the following:

- when a function is declared or defined without a function prototype **if** the function is referenced in the source module
- when alignment padding appears in structures that are allocated using the longmap alignment rules
- when an unrecognized option is used in a `#pragma` directive.

The syntax for the system programming options is as follows:

```
        ┌   system_programming    ┐
#pragma │                         │
        └  no_system_programming  ┘
```

If you specify `system_programming`, the compiler performs the system programming checks.

If you specify `no_system_programming`, the compiler does not perform the system programming checks. By default, the compiler does not perform these checks.

The `type_checking` Option

The `type_checking` option specifies the level of type checking that the compiler uses to diagnose occurrences of implicit or unintended data-type conversions and other programming constructs that can cause error conditions to occur. The syntax for the `type_checking` option is as follows:

```
#pragma type_checking ( level )
```

The allowed values for *`level`* are `none`, `minimum`, `normal`, and `maximum`. By default, the compiler uses the `minimum` level of checking. The `normal` level of type checking is the level most programmers will want for a typical program.

If you specify `none` as the level of type checking, the compiler does not perform any checks for data-type consistency.

If you specify `minimum` as the level of type checking, the compiler produces warnings for the following occurrences.

- Implicit data-type conversions involving pointer and `char_varying` conversions:
  - pointer to pointer with an object of a different data type
  - non-`char_varying` to `char_varying` strings
  - `char_varying` to non-`char_varying` strings.
- Other violations that can affect program execution:
  - returning a value in a function defined as returning `void`

  - using a pointer to a function where a function is required
  - qualifying a structure member with an address expression that does not locate a structure containing such a member
  - specifying an `extern` declaration and definition that have different types
  - using decimal digits in an octal constant
  - omitting a required semicolon
  - omitting an equals sign (`=`) before an initializer list.

If you specify `normal` as the level of type checking, the compiler includes all `minimum` diagnostics and produces warnings for the following occurrences.

- Other implicit data-type conversions, such as constant conversion where the precision is lost (for example, floating-point-to-integer conversion).
- Other violations that can affect program execution:
  - using an expression that does not produce code (for example, `a == 0;`)
  - omitting braces around an initializer list
  - failing to declare or define a function before it is invoked.

If you specify `maximum` as the level of type checking, the compiler includes all `minimum` and `normal` diagnostics and produces warnings for the following occurrences.

- Other implicit data-type conversions where precision or value can be lost:
  - `signed` to `unsigned`
  - `int` or `long` to `short` or `char`
  - `short` to `char`
  - `double` to `float`
  - non-`float` to `float`.
- Other violations that can affect program execution: use of an obsolete compound assignment operator (for example, use of a minus sign (`-`), asterisk (`*`), or ampersand (`&`) immediately after an equals sign (`=`), without an intervening space).

The `maximum` level of type checking may be stricter than many programmers want. The `maximum` level of type checking is intended **for debugging purposes only.** See the *VOS C User's Guide (R141)* for information on choosing a level of type checking.

The Untyped Storage Sharing Options

The `untyped_storage_sharing` and `no_untyped_storage_sharing` options specify whether the compiler flushes knowledge of the contents of all aliasable data objects when any object is accessed through a pointer dereference. An *aliasable data object* is one that can be accessed by the program using an expression other than the object's name (for example, by using a pointer dereference), or is one that can be accessed by another program. Specifying no_untyped_storage_sharing can improve program performance. The syntax for the untyped storage sharing options is as follows:

```
        ┌   untyped_storage_sharing    ┐
#pragma │                              │
        └ no_untyped_storage_sharing   ┘
```

If you specify untyped_storage_sharing, when a data object is accessed through a pointer dereference, the compiler flushes knowledge of **all** aliasable data objects. By default, the compiler flushes knowledge of all such objects.

If you specify no_untyped_storage_sharing, when an object is accessed through a pointer dereference, the compiler flushes knowledge only of all **similarly typed** aliasable data objects.

The following example and explanation illustrate how specifying no_untyped_storage_sharing can improve the performance of your program.

```
extern int i;

void func_1(char *c_ptr)
{
   i = 1;

   *c_ptr = 'X';

   func_2(i);
       .
       .
       .
}
```

If untyped_storage_sharing is specified for the source module containing the preceding example, after a character is modified by dereferencing c_ptr, the compiler assumes that it can no longer be sure of the contents of any aliasable objects. The compiler, therefore, flushes knowledge of all aliasable data objects, such as i. That is, the value of i can no longer be obtained from a machine register. Any subsequent use of the contents of i, as in the invocation of func_2, requires that the value stored in i be obtained from memory.

In C programming, if an object of one type is modified through a pointer to an incompatible object type, subsequent accesses of the object can produce incorrect results. The flushing that occurs with the untyped_storage_sharing option ensures that, when an object is accessed through a pointer, the dereference yields the correct result **regardless** of whether the pointed-to object has been modified by a dereference through an incompatible pointer.

In contrast to the untyped_storage_sharing option, if no_untyped_storage_sharing is specified for the source module containing the preceding example, after a character is modified by dereferencing c_ptr, the compiler assumes that it can continue to be sure of the contents of all aliasable objects other than char objects. The compiler, therefore, flushes knowledge only of all aliasable char data objects. Subsequent accesses of i, as in the invocation of func_2, get the value stored in i from the register, not memory. Because a register access is faster than a memory access, specifying no_untyped_storage_sharing allows the compiler to generate faster code.

Note: With either `untyped_storage_sharing` or `no_untyped_storage_sharing`, if a C program **never** uses a pointer (for example, with a cast) to access an object other than the type defined for the pointer, pointer dereferences yield accurate results.

Other Directives

The VOS C preprocessor also recognizes three other directives that are not part of the ANSI C Standard's description of preprocessing directives. These directives are **not** portable. The following directives are VOS C extensions.

- `#list`
- `#nolist`
- `#page`

The # `list` and # `nolist` Directives

The `#list` and `#nolist` directives tell the compiler whether source text following the directive should be included in any `.list` file. When you select the `c` command's `-list` argument, the compiler generates a compilation listing with the `.list` suffix. The `#list` and `#nolist` directives can appear in the source module any number of times. The syntax for these directives is as follows:

```
#list

#nolist
```

If you specify the `#list` directive, the compiler includes, into the `.list` file, any source text following the directive unless you subsequently use the `#nolist` directive, or use the `no_list` option in a `#pragma`.

If you specify the `#nolist` directive, the compiler does **not** include, into the `.list` file, any source text following the directive unless you subsequently use the `#list` directive, or use the `list` option in a `#pragma`.

Note: You can use the `list` and `no_list` options in `# pragma` directives as a portable alternative to the `# list` and `# nolist` directives.

The # `page` Directive

The `#page` preprocessor directive tells the compiler to start a new page in any `.list` file. When you select the `c` command's `-list` argument, the compiler generates a compilation listing with the `.list` suffix. The `#page` directive can appear in the source module any number of times. The syntax for this directive is as follows:

```
#page
```

If you specify the `#page` directive, the compiler starts a new page in the `.list` file.

Preprocessor Directives Summary

The tables in this section provide a summary of the VOS C preprocessor directives.

- Table 9-7 lists the preprocessor directives available in VOS C.
- Table 9-8 lists the `#pragma` options available in VOS C.

Before using an option in a `# pragma` directive, read the section earlier in this chapter that fully describes the option. Only these earlier sections contain information on any consequences or interactions that may occur when a `# pragma` option is used.

Table 9-7. VOS C Preprocessor Directives Summary

Directive	**Purpose**
`# define`	Defines a preprocessor macro
`# elif`	Processes text when the corresponding `# if` directive is false and when the `# elif` directive's constant expression is true
`# else`	Processes text when the corresponding `# if` or `# elif` directive's constant expression is false
`# endif`	Marks the end of an if-group
`# if`	Processes text when the constant expression is true
`# ifdef`	Processes text when the identifier is defined as a macro
`# ifndef`	Processes text when the identifier is not defined as a macro
`# include`	Incorporates, into the source module, text from a specified file
`# list`	Includes subsequent text into a `.list` file
`# no_list`	Excludes subsequent text from a `.list` file
`# page`	Starts a new page in a `.list` file
`# pragma`	Specifies a VOS C option that causes the compiler to perform a particular action
`# undef`	Undefines a preprocessor macro

Table 9-8. VOS C Pragma Option Summary *(Page 1 of 3)*

`#pragma` $\left[\begin{array}{c}\texttt{ansi\_rules}\\ \texttt{no\_ansi\_rules}\end{array}\right]$ Specifies whether the compiler suppresses parameter substitution inside character-string literals that appear in macro-definition lines. The default is `no_ansi_rules`. By default, the compiler does not suppress parameter substitution inside character-string literals.
`# pragma bit_field_align (` *`type`* `)` Specifies the type of boundary where a bit field can begin. The allowed values for *`type`* are `int`, `short`, and `char`. By default, the compiler begins bit fields on even-numbered byte (`short`) boundaries.
`# pragma bit_field_size (` *`size`* `)` Specifies the amount of space that the compiler reserves when it needs to allocate storage for bit fields in a structure or union. The allowed values for *`size`* are `int`, `short`, and `char`. By default, the compiler reserves four bytes (an `int`-size space).
`# pragma bit_packing (` *`direction`* `)` Specifies whether the compiler starts allocating bits from the left or right side of the reserved space. The allowed values for *`direction`* are `right_to_left` and `left_to_right`. By default, the compiler begins allocating bits from the left side of the reserved space.
`#pragma` $\left[\begin{array}{c}\texttt{common\_constants}\\ \texttt{no\_common\_constants}\end{array}\right]$ Specifies whether the compiler allocates character-string literals in the code region or the static region. A program cannot overwrite character-string literals that are stored in the program's code region. By default, the compiler stores character-string literals in the code region.
`# pragma default_char (` *`type`* `)` Specifies whether `char` data items that are declared without an explicit `signed` or `unsigned` keyword are signed or unsigned. The allowed values for *`type`* are `signed` and `unsigned`. By default, the compiler assumes that `char` types without an explicit `signed` or `unsigned` have the type `unsigned char`.
`# pragma default_char_set (` *`character_set`* `)` Specifies whether the compiler stores character-string literals as canonical strings or as common strings. The allowed values for *`character_set`* are `none` and `latin_1`. By default, the compiler uses Latin alphabet No. 1 (`latin_1`) as the default character set.
`#pragma` $\left[\begin{array}{c}\texttt{default\_mapping}\\ \texttt{no\_default\_mapping}\end{array}\right]$ Specifies whether the compiler diagnoses the absence of a `$shortmap` or `$longmap` specifier in a structure or union declaration. By default, the compiler does not issue a diagnostic for such a declaration.

Table 9-8. VOS C Pragma Option Summary *(Page 2 of 3)*

```
#pragma [   check_enumeration    ]
        [ no_check_enumeration   ]
```

Specifies whether the compiler issues a diagnostic for an implicit conversion involving an enumeration data item. By default, the compiler does not issue these diagnostics.

```
#pragma [    expanded_macros  ]
        [ no_expanded_macros  ]
```

Specifies whether the compiler expands subsequent macros in a `.list` file, if such a file is generated. By default, the compiler does not expand macros in a `.list` file.

```
# pragma extension_checking ( level )
```

Specifies the level of extension checking that the compiler uses to diagnose VOS C language extensions that can affect program transportability. The allowed values for *`level`* are `none`, `minor`, and `all`. By default, the compiler uses `none`.

```
#pragma [   linted   ]
        [ no_linted  ]
```

Specifies whether the compiler generates extra code for the `return` statement to load both the address and data registers, disregarding the definition of the function containing the `return` statement. By default, the compiler generates the extra code.

```
#pragma [   list   ]
        [ no_list  ]
```

Specifies whether source text following the `# pragma` directive should be included in any `.list` file, if such a file is generated. By default, all source text is included in the `.list` file.

```
#pragma [   mapcase   ]
        [ no_mapcase  ]
```

Specifies whether the compiler interprets uppercase letters differently from lowercase letters. By default, the compiler does interpret uppercase letters differently from their lowercase counterparts.

```
#pragma mapping_rules ( mapping [, checking] )
```

Specifies the data alignment rules that the compiler uses when laying out storage for the source module. The allowed values for *`mapping`* are `longmap` and `shortmap`. By default, the compiler uses the system-wide default data alignment method. The allowed values for *`checking`* are `check` and `no_check`. By default, the compiler does not diagnose such alignment padding.

```
# pragma processor ( processor_string )
```

Specifies the processor on which the code will run. By default, the compiler generates code that will run on the default system processor. The allowed values for *`processor_string`* vary depending on the family of the current processor. See “The `processor` Option” section earlier in this chapter for information on the allowed values.

Table 9-8. VOS C Pragma Option Summary *(Page 3 of 3)*

`# pragma promotion_rules (` *`promotion`* `[, ` *`checking`* `])` Specifies the promotion rules that the compiler uses when performing certain data-type promotions. The allowed values for *`promotion`* are `ansi` and `non_ansi`. By default, the compiler uses the ANSI promotion rules (that is, `ansi`). The allowed values for *`checking`* are `check` and `no_check`. By default, the compiler assumes `no_check`.
`#pragma [ registers / no_registers ]` Specifies whether the compiler assigns to machine registers any data items that are declared with the `register` storage class specifier. By default, the compiler attempts to allocate these items in registers.
`#pragma [ support / no_support ]` Specifies whether functions within the source module will be used as system support routines. With a system support routine, the debugger's `trace` request does not display the stack frame associated with the function. By default, the compiler does not assume that functions are system support routines.
`#pragma [ system_programming / no_system_programming ]` Specifies whether the compiler performs three checks commonly needed for programs used in VOS C system programming. See "The System Programming Options" section earlier in this chapter for information on these checks. By default, the compiler does not perform the system programming checks.
`# pragma type_checking (` *`level`* `)` Specifies the level of type checking that the compiler uses to diagnose occurrences of implicit or unintended data-type conversions and other programming constructs that can cause error conditions to occur. The allowed values for *`level`* are `none`, `minimum`, `normal`, and `maximum`. By default, the compiler uses the `minimum` level of checking.
`#pragma [ untyped_storage_sharing / no_untyped_storage_sharing ]` Specifies whether the compiler flushes knowledge of all aliasable data objects when an object is accessed through a pointer dereference. By default, the compiler flushes knowledge of all such objects.

Chapter 10: Input and Output

In C, the language itself does not contain any specialized statements that allow for input to a program or output from a program. All I/O in a C program is performed by calling the appropriate library functions. VOS C provides two general categories of I/O functions:

- the standard I/O functions
- the UNIX I/O functions.

This chapter provides an overview of how you perform input and output with both the standard and the UNIX I/O functions. Topics discussed in this chapter include the following:

- streams
- buffered input and output
- text and binary modes
- the `FILE` structure
- file-position indicator
- end-of-file indicator
- file I/O error handling
- standard I/O functions
- UNIX I/O functions.

The UNIX I/O functions are **not** ANSI-C-compatible. Programs that use the UNIX I/O functions may not be portable. The UNIX I/O functions are no more efficient than the standard I/O functions. The UNIX functions are included in the VOS C library so that existing programs that use them will be compatible with the VOS implementation of the C language.

For a detailed description of each of the I/O functions contained in the VOS C library, see Chapter 11.

Streams

In C, a logical data *stream* is a metaphor for an external file or a physical device: the stream of bytes from and the I/O buffer associated with the file or device. A stream is associated with a file by opening the file, which may involve creating the file. A program reads data from the stream and writes data to the stream associated with a file.

In VOS C, there is no limit on the number of bytes that can be read or written in an I/O operation. However, if you are reading from or writing to a VOS file with fixed, relative, or sequential file organization, a record can contain not more than 32,767 bytes.

In general, this document uses the less figurative term "file" rather than "stream." For example, this document might state that a function "reads a character from a file" (as opposed to a stream).

Opening a File

When a file is opened with `fopen`, `freopen`, or `tmpfile` (the `creat` and `open` functions operate similarly), the opening function allocates an I/O buffer for the file. Thus, all I/O on a stream is buffered unless the program changes the buffering method by a call to `setbuf` or `setvbuf`. See the "Buffered Input and Output" section later in this chapter for more information on buffering.

There are two types of streams: output streams and input streams. When a file is opened for output, or is opened for update and the most recent operation on the file was output, the stream associated with the file is an *output stream*. When a file is opened for input, or is opened for update and the most recent operation on the file was input, the stream associated with the file is an *input stream*. In VOS C, when `fopen` creates and opens a file but the program writes no characters to the file as an output stream, a file of zero length exists when the program terminates.

The three *standard streams* identify the following `FILE` objects:

- `stdin` for reading input from the terminal's keyboard
- `stdout` for writing output to the terminal's screen
- `stderr` for writing diagnostic output to the terminal's screen.

You do not have to open the three standard streams because they are predefined by the system.

In VOS C, whether one file can be simultaneously open multiple times depends on the locking mode that is specified when the file is opened.

Closing a File

A file is disassociated from a controlling stream by closing the file. When a file is closed, the contents of any associated input stream are discarded before the stream is disassociated from the file. When a file is closed, the contents of any associated output stream are flushed before the stream is disassociated from the file. That is, any unwritten contents in the file's I/O buffer are transmitted to the file or device. The VOS system automatically closes all open files and flushes the contents of all output streams in almost all contexts including the following:

- if the `main` function returns
- if the program calls the `abort` or `exit` function.

If a program calls `s$stop_process` to stop the program's process, the system closes all open files but does **not** automatically flush the contents of output streams.

A closed file can be subsequently reopened, by the same or another program, and its contents accessed or modified if the file can be repositioned at the beginning of the file.

Buffered Input and Output

In most programming situations, how the VOS C functions implement buffering is not of concern to the programmer. The system performs buffered I/O in a manner that is efficient and transparent to the user. In a few programming situations, knowledge of buffering and how it is controlled are helpful in producing a more efficient program.

This section explains the following:

- I/O buffers
- the buffering methods available with the VOS C I/O functions
- the functions used to control buffering.

Overview: I/O Buffers and Buffering

All VOS C input and output is, by default, buffered. The system automatically allocates an I/O buffer for a file when the file (or device) is opened with the `creat`, `fopen`, `freopen`, `open`, or `tmpfile` function. A *buffer* is a region of memory where data is temporarily stored after data is read from a file, or before data is written to a file.

The I/O buffer acts as an intermediate storage area between the file and the program (see Figure 10-1). For example, when a program first reads data from an open file using a C function such as `fgetc`, the system fills the file's I/O buffer with data from the file so that the program really accesses the buffer and not the file. Subsequent read operations on the file also read data from this buffer. When the program has read all data in the buffer, the system automatically refills the buffer by reading more data from the file. Thus, buffered I/O reads and writes file data in buffer-size chunks, requiring fewer read and write operations on external devices, such as disks.

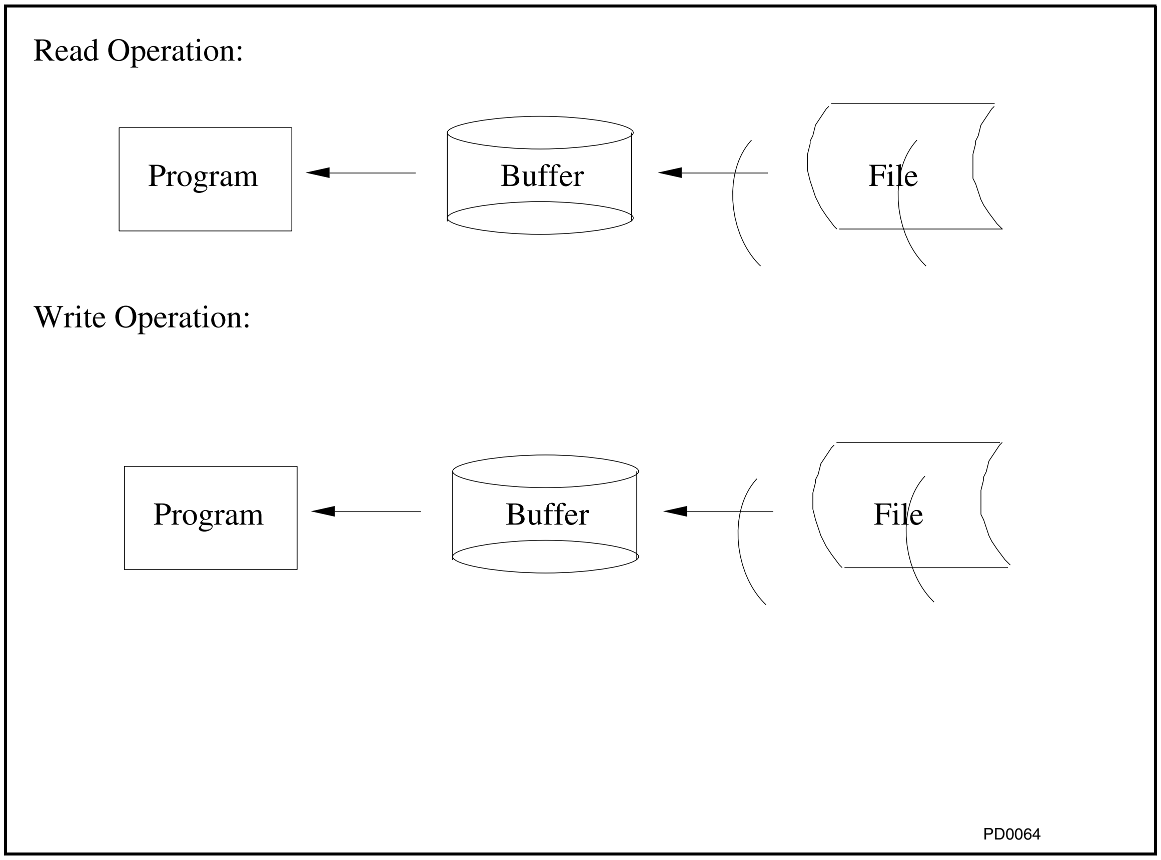


Figure 10-1. Buffered I/O

In almost all cases, buffered I/O is more efficient than unbuffered I/O. For example, character I/O with `getc` is, by default, performed through the use of a macro and an I/O buffer. When `getc` is invoked to read a character from a file, the system fills the file's I/O buffer with data and the `getc` macro accesses a character in the buffer. Subsequent calls to the `getc` macro access the next character in the file's buffer without the overhead of a function call or a disk read.

The one exception where unbuffered I/O may offer performance advantages over buffered I/O occurs when a very large block of data is read or written in a single function call. In general, unbuffered I/O is faster than buffered I/O when the data can be read in large chunks. For example, when copying a large file, you can perform unbuffered I/O in chunks significantly greater than the size of the I/O buffer associated with the file. In addition, unbuffered I/O means that the data in the file will be copied once rather than multiple times as usually happens when I/O is performed through an intermediate I/O buffer on a fully buffered file.

For an example and further explanation of unbuffered I/O, see the "Sample Program Using the UNIX I/O Functions" section later in this chapter.

Buffering Methods

In VOS C, both the standard and UNIX I/O functions can perform input and output on a file using one of three buffering methods:

- unbuffered
- line buffered
- fully buffered.

When a file is *unbuffered*, data is transmitted between the program and the file without any intermediate storage in a buffer. The number of bytes specified in the I/O function call determines the size of the data that is read from or written to the file. For example, every `fgetc` call on an unbuffered file requires that a read operation be performed on the external storage device.

When a file is *line buffered*, data is transmitted between the program and the file using intermediate storage in a buffer. Data is transmitted to or from the buffer as a block when a newline character (`\n`) is encountered. With line buffering, when a buffer used for read operations is empty, input data consisting of the next sequence of characters terminated by a newline character is read from the file. Furthermore, with line buffering, output data is written from the buffer to the file when one of the following occurs:

- when the buffer is full
- when a newline character is encountered in the output
- when the buffer is flushed with the `fflush` function.

In VOS C, the standard streams, `stdin`, `stdout`, and `stderr`, are line buffered by default. For example, input from `stdin` is received by the program only when the user enters a newline character (that is, presses the RETURN key on the terminal's keyboard). The input operation occurs in the following manner. As each character is entered on the terminal's keyboard, the character is written to the associated I/O buffer. When the user presses the RETURN key, the I/O buffer is flushed, and the data is sent to the program.

When a file is *fully buffered*, data is transmitted between the program and the file using intermediate storage in a buffer. Data from the file is read into the I/O buffer as a block when the buffer is empty, and data is written out to the file as a block when the buffer is full. For a stream file or sequential file, the default size of the system-allocated I/O buffer is 512 bytes. For a relative file or fixed file, the default size of the system-allocated I/O buffer is the same as the record size specified when the file is created.

In VOS C, all files except `stdin`, `stdout`, and `stderr` are fully buffered by default.

Note: Although relative and sequential files are fully buffered by default, a file with either of these organizations is always line buffered in effect because the newline character has a special meaning as the end of a record when the file is opened in text mode.

Buffer-Related Functions

You can use the `setvbuf` function (`setbuf` has more limited functionality) to change one or more of the following aspects of buffering for a specified file: the buffer to be used for buffered I/O, the type of buffering used for the file, or the size of the buffer used.

However, certain restrictions apply when you are changing the default buffering.

- For a stream, sequential, or relative file that has been opened in binary mode, you can choose fully buffered or unbuffered.
- For a fixed file, you can choose only fully buffered.
- For `stdin` and `stdout`, you can choose line buffered or unbuffered if `stdin` and `stdout` are associated with a terminal device. Or, if `stdin` and `stdout` are not associated with a terminal device, you can choose fully buffered, line buffered, or unbuffered.
- For `stderr`, you can choose line buffered or unbuffered.
- For a terminal device opened in text mode, you can choose line buffered or unbuffered.
- For a terminal device opened in binary mode, you can choose only unbuffered.

Text and Binary Modes

When a file is opened, you can specify that the system read input from a file or write output to a file in one of two ways: in text mode or in binary mode. The terms “text mode” and “binary mode” refer to the way in which some aspects of input and output (for example, the handling of newline characters) are performed by the system. Depending on the opening function that you use, one of the following arguments determines whether a file is opened in text mode or in binary mode:

- the `mode` argument when you open a file with `fopen` or `freopen`
- the `o_flag` argument when you open a file with `open`.

This document sometimes uses the terms “text file” and “binary file” to indicate text mode and binary mode. Do not confuse these two“types” of files with the VOS file organizations, such as stream file organization or sequential file organization.

Text Mode

When a file is opened in *text mode*, the file is interpreted as a sequence of ASCII characters composed into lines. Each line consists of zero or more characters plus a terminating newline character.

For file I/O operations in text mode to be portable from one operating system to the next, the ANSI C Standard specifies that the following conditions must be true about the data that will be read or written.

- The data must consist only of printable characters and the horizontal tab and newline characters.
- No newline character can be immediately preceded by space characters.
- The last character in the data must be a newline character.

If the preceding conditions are true, data read in from a file opened in text mode will necessarily compare equal to data that was earlier written out to the file. In VOS C, space characters that are written out immediately before a newline character appear when read.

The following discussion on text mode applies to input and output with both the standard I/O and UNIX I/O functions. In VOS C, when you specify that a file is to be opened in text mode, the system may or may not treat the newline character in a special way depending on the file's organization.

For a file with stream file organization (the default), the system does not treat the newline character in a special way. By convention, the newline character marks the end of a record (that is, a line of text). To locate the end of each record, the application program determines where each newline character appears in the file.

For a file with fixed file organization, the system does not treat the newline character in a special way. In a fixed file, a newline character does **not** mark the end of a record. A record in a fixed file always equals the record size regardless of where newline characters appear. Thus, in a fixed file, newline characters can appear within a record.

For a file with sequential or relative file organization, the system does the following:

- On input, the system appends a newline character after every record.
- On output, a newline character in the data to be written marks the end of a record.

For files having sequential or relative file organization, you explicitly insert the newline character into the file's data to delimit a record when you are writing to a file. Depending on where you insert the newline character, a record in a relative file can be equal to or less than the maximum record size, and a record in a sequential file can vary in size. The newline character is considered part of the record though the newline character is not actually written out to the file.

For more information on the VOS C file organizations, see the *VOS C User's Guide (R141)*.

Binary Mode

When a file is opened in *binary mode*, the file is an ordered sequence of characters that can transparently record internal data. Data read in from a file opened in binary mode will necessarily compare equal to data that was earlier written out to the file only if the operations were performed by programs operating on other Stratus modules.

The following discussion on binary mode applies to input and output with both the standard I/O and UNIX I/O functions. In VOS C, when you specify that a file is to be opened in binary mode, the system does not treat the newline character in a special way.

For a file with stream or fixed file organization, the system handles newline characters in the same manner regardless of whether the file is opened in text mode or binary mode. See the "Text Mode" section earlier in this chapter for information on the use of a newline character with these two file organizations.

For a file with sequential or relative file organization that is opened in binary mode, the system does not interpret a newline character as the end of a record when you are writing to a file. Data is written to the file when the I/O buffer is full. For a file with sequential file organization, the number of bytes contained in a record equals the current buffer size associated with the file. For a file with relative file organization, the number of bytes contained in a record equals the record size specified when the file is opened. On input, when a file with sequential or relative file organization is opened in binary mode, the system does **not** append a newline character after every record.

For more information on the VOS C file organizations, see the *VOS C User's Guide (R141)*.

Text and Binary Format: A Comparison

Text mode and binary mode are used for different purposes. Typically, files that are opened in text mode store information in text format: using sequences of ASCII characters. Files that are opened in binary mode normally store information in binary format: using each data type's internal representation. (Of course, ASCII characters too are stored internally in binary format.)

The I/O function that writes the data to the file determines how the information is stored: in text format or binary format. Figure 10-2 shows how the `int` value 999999 would be stored if it were written in text format and binary format using the following functions.

- For text format, `fprintf` uses the conversion specifier `%d` to write the `int` value 999999 as a series of six ASCII characters. Each character has the value 39 hexadecimal, the ASCII code for the character `9`.
- For binary format, `fwrite` writes the `int` value 999999, without any conversion, as a series of four bytes using the VOS C internal representation for an `int`. Figure 10-2 shows each byte of the internal representation in hexadecimal.

For information on the internal representation of the VOS C data types, see Appendix A.

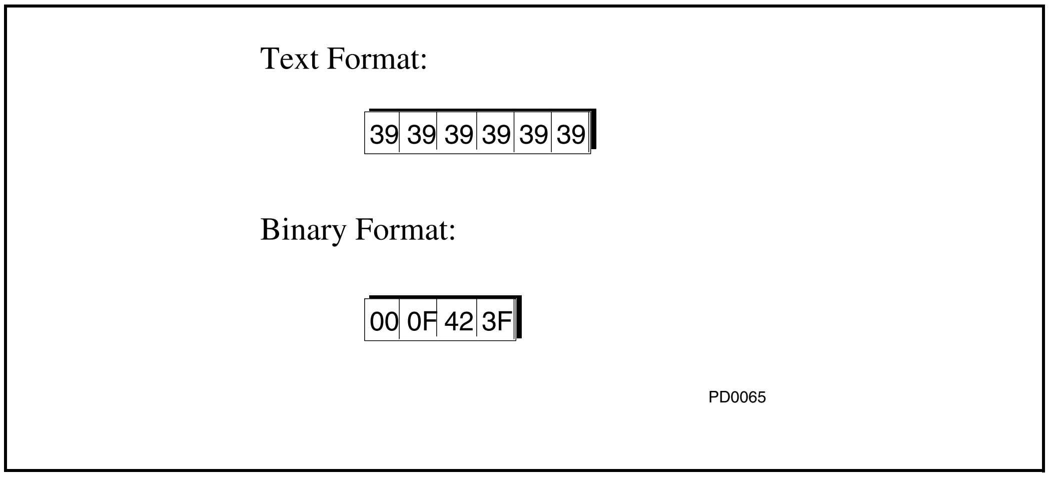


Figure 10-2. Text and Binary Formats

Binary format has some advantages over text format. For storing numeric data, binary format is usually more compact and, particularly for `float` and `double` values, more accurate. When the value of a floating-point variable is converted to text format with a formatted I/O function such as `fprintf`, accuracy is lost due to rounding. As shown in Figure 10-2, the binary I/O function `fwrite` performs no conversions before writing data to a file. Not having to convert numeric data to an ASCII character representation can yield performance gains in some applications.

On the other hand, binary format has some disadvantages. With binary format, the terminal displays the file's contents in hexadecimal form, not an easily readable form for large amounts of data. In addition, you cannot use a text editor to enter data or modify a file's contents. With binary format, you have to write a program to enter data into the file, modify the file's contents, and convert the contents to easily readable form. Also, data stored in binary format may not be readable on other machines unless the data is coded in some "generic" form agreed upon by each machine. This generic form is needed because the internal representation of data may be different from machine to machine.

The `FILE` Structure

When a program opens a file, the system allocates a structure of the type `FILE` and associates that structure with the file. The `fdopen`, `fopen`, `freopen`, and `tmpfile` functions return a pointer to a structure object of the type `FILE`. The `FILE` type is defined in the `stdio.h` header file.

After the file-opening function opens a file and returns a pointer to the `FILE` object, you can specify this pointer to identify the file in subsequent standard I/O operations on the file. In many of the function descriptions in Chapter 11, this pointer is referred to as a *file pointer.* The address of the `FILE` object used to control a file is significant. You cannot use a copy of the `FILE` object to serve in place of the original. The value of a pointer to the `FILE` object is indeterminate after the associated file (including the standard streams) is closed.

In other VOS languages, a structure similar to the `FILE` structure is sometimes called a *file control block*. This structure contains all the information needed to handle a file, such as current position in the file, the address of the associated I/O buffer, and the file descriptor associated with the file.

Not all information within the `FILE` object is accessible to a program. You should **not** directly access any member of the `FILE` object. To find out information about an open file that has an associated file pointer, you must use the following functions.

- `ftell` determines the current position of the file-position indicator for a file.
- `feof` determines whether the end-of-file indicator for a file has been set.
- `ferror` determines whether the error indicator for a file has been set.

Most files, whether opened by the standard I/O or UNIX I/O opening functions, have a file-position indicator, end-of-file indicator, and an error indicator associated with them. When a file is opened, the end-of-file and error indicators are cleared.

The `FILE` object also contains the file descriptor associated with the file. You use this file descriptor to specify a file with the UNIX I/O functions, such as `lockf`. To access the file descriptor associated with a specified file, you can use the `fileno` macro.

File-Position Indicator

If a file can support positioning requests (for example, a disk file can but a terminal cannot), then the file's *file-position indicator* marks the point at which the file is being written or read. The file-position indicator is positioned at the beginning of the file when you open a file with a function such as `fopen`. In VOS C, opening a file in append mode also sets file-position indicator to the beginning of the file though, by default, all output is written to the end of the file. To make it easier for a program to progress through a file in an orderly manner, the system updates the value of the file-position indicator when the program performs a read, write, or positioning operation on the file.

In VOS C, a write operation on a file never truncates the file beyond the point of the newly written data.

You can determine the value of a file's file-position indicator by calling the `ftell` function. You can change the current value of a file's file-position indicator by calling `fseek`, `lseek`, `rewind`, or the `fgetpos` and `fsetpos` functions.

End-of-File Indicator

When no more data remains in a file, the file is positioned at *end-of-file*. The system keeps track of the end-of-file using the file's end-of-file indicator.

If a standard I/O function, such as `fgetc`, determines that end-of-file has been encountered, it communicates that information to the program by returning `EOF` as a return value. The `EOF` macro is defined in the `stdio.h` header file as the value -1. Many standard I/O functions return `EOF` when either of the following occurs:

- when the file is at end-of-file
- when an I/O error condition occurs.

You can use the `feof` and `ferror` functions to determine if end-of-file was encountered or if an error occurred. As an alternative, the program can read the value of the external variables `errno` and `os_errno` to determine why an I/O function returned `EOF`.

> **Note:** If you plan to read the value of `errno` or `os_errno` to determine why an I/O library function returned `EOF`, the program must explicitly set `errno` and `os_errno` to the value 0 before calling the function. No library function sets these external variables to 0 when it is called.

A file's end-of-file indicator stays set until you call the `clearerr`, `fseek`, `fsetpos`, `lseek`, or `rewind` function and specify the file pointer associated with the file. The end-of-file indicator is not set on an empty file. Closing the file also clears the file's end-of-file indicator.

When you are entering data on the terminal's keyboard, you can use the VOS `&eof` character sequence to signal end-of-file for the input stream associated with `stdin`. In some contexts, you must enter a carriage return and then `&eof` for the operating system to recognize that end-of-file has been encountered in keyboard input.

File I/O Error Handling

All opened files have an error indicator associated with them. When an error occurs during an I/O operation on the file, certain library functions set the file's error indicator. You can determine whether a file's error indicator is set by calling the `ferror` function. A file's error indicator stays set until you call the `clearerr` or `rewind` function and specify the file pointer associated with the file. Closing the file also clears the file's error indicator.

In addition, you can read the value of `errno` and `os_errno` to help determine the cause of an I/O error. Many of the I/O functions set the value of the external variables `errno` and `os_errno` to the appropriate error code number when an I/O error occurs. The system sets `errno` and `os_errno` to the value 0 when a program begins.

> **Note:** If you plan to read the value of `errno` or `os_errno` to determine the cause of an I/O error, the program must explicitly set `errno` and `os_errno` to the value 0 before calling the function. No library function sets these external variables to 0 when it is called.

You can use the `perror` function to construct an error message that incorporates the error name associated with the error code number stored in `errno`. The `perror` function writes that error message to `stderr`.

Standard I/O Functions

The standard I/O functions are those functions specified in the ANSI C Standard for input and output. The standard I/O functions are declared in the `stdio.h` header file and perform input and output with files, the terminal's screen and keyboard, and other devices.

This section provides an overview of the VOS C standard I/O functions that are also specified in the ANSI C Standard. Based on the tasks they perform, the standard I/O functions can be divided into the following general categories:

- file-access functions
- formatted I/O functions
- single-character I/O functions
- string I/O functions
- binary (direct) I/O functions
- file-positioning functions.

This section does **not** include information on other I/O functions that are found in the VOS C `stdio.h` header file but that are not specified in the ANSI C Standard. These other functions, such as `putw` and `getw`, are included in the `stdio.h` header so that an existing program that uses them will be compatible with the VOS implementation of the C language.

File-Access Functions

The file-access functions open and close files, flush output buffers (if needed), and change one or more aspects of file I/O buffering. Table 10-1 lists the file-access functions declared in the `stdio.h` header file.

Table 10-1. File-Access Functions

Function	Description
`fclose`	Closes a file associated with a specified file pointer.
`fflush`	Flushes the output buffer for a file associated with a specified file pointer.
`fopen`	Opens a file and associates a stream with the file.
`freopen`	Closes the file associated with a specified file pointer, opens another file, and associates the file pointer and stream with that file.
`setbuf`	Causes a specified buffer, instead of a buffer allocated by the system, to be used for buffered I/O on a file associated with a given file pointer.
`setvbuf`	For a file associated with a specified file pointer, changes one or more aspects of buffering: the buffer to be used in place of the system-allocated buffer, the mode of buffering, or the size of the buffer.

Formatted I/O Functions

The formatted I/O functions read input or write output under the control of a specified format.

- With the formatted **output** functions, you use a format string to indicate how the value of a subsequent argument will be converted into one or more ASCII characters and then written to a specified file or array.
- With the formatted **input** functions, you use a format string to indicate how an input item consisting of one or more characters will be converted and then assigned to a subsequent argument.

Table 10-2 lists the formatted I/O functions.

Table 10-2. Formatted I/O Functions *(Page 1 of 2)*

Function	Description
`fprintf`	Writes a formatted string of characters to a file associated with a specified file pointer.
`fscanf`	Using a specified format, reads a sequence of characters from the file associated with a given file pointer.
`printf`	Writes a formatted string of characters to the standard output file, `stdout`.
`scanf`	Using a specified format, reads a sequence of characters from the standard input file, `stdin`.
`sprintf`	Writes a formatted string of characters to an array.
`sscanf`	Using a specified format, reads a sequence of characters from a given string.

Table 10-2. Formatted I/O Functions *(Page 2 of 2)*

Function	Description
`vfprintf`	Writes a formatted string of characters to a file. Unlike `fprintf`, this function accepts a pointer to a list of arguments, not the arguments themselves.
`vprintf`	Writes a formatted string of characters to the standard output file, `stdout`. Unlike `printf`, this function accepts a pointer to a list of arguments, not the arguments themselves.
`vsprintf`	Writes a formatted string of characters to an array. Unlike `sprintf`, this function accepts a pointer to a list of arguments, not the arguments themselves.

Single-Character I/O Functions

Except for `ungetc`, the single-character I/O functions read one character from a file or write one character to a file. The `ungetc` function serves a special purpose: it pushes back the last character read from the I/O buffer associated with a specified file. Table 10-3 lists the single-character I/O functions and macros.

Table 10-3. Single-Character I/O Functions and Macros

Function	Description
`fgetc`	Reads the next character from a file associated with a specified file pointer.
`fputc`	Writes one character to a file associated with a specified file pointer.
`getc`	Reads the next character from a file associated with a specified file pointer.
`getchar`	Reads the next character from the standard input file, `stdin`.
`putc`	Writes one character to a file associated with a specified file pointer.
`putchar`	Writes one character to the standard output file, `stdout`.
`ungetc`	Pushes a specified character back onto the input stream associated with a file.

String I/O Functions

The string I/O functions read an entire line from or write a string to a file. Table 10-4 lists the string I/O functions.

Table 10-4. String I/O Functions

Function	Description
`fgets`	Reads a line or a specified number of characters from a file associated with a specified file pointer.
`fputs`	Writes a string to a file associated with a specified file pointer.
`gets`	Reads a line from the standard input file, `stdin`.
`puts`	Writes a string to the standard output file, `stdout`.

Binary I/O Functions

The binary I/O functions, typically, read data from or write data to a binary file. In contrast to the formatted I/O functions, such as `fscanf` and `fprintf`, the binary I/O functions perform no conversions on the data that is read or written. The ANSI C Standard calls these the "direct I/O functions." The binary I/O functions are the standard I/O counterpart to the `read` and `write` UNIX I/O functions. Table 10-5 lists the binary I/O functions.

Table 10-5. Binary I/O Functions

Function	Description
`fread`	Reads a specified number of data items, each of an indicated size, from a file associated with a given file pointer.
`fwrite`	Writes a specified number of data items, each of an indicated size, into a file associated with a given file pointer.

File-Positioning Functions

The file-positioning functions set the file-position indicator for a specified file so that the file's contents can be accessed randomly. Using the file-positioning functions, you can change the current position within the file and thus read or write data at any position within a file. See the description of each file-positioning function in Chapter 11 for any restrictions on the use of a particular function. Table 10-6 lists the file-positioning functions.

Table 10-6. File-Positioning Functions *(Page 1 of 2)*

Function	Description
`fgetpos`	Gets and stores the current value of the file-position indicator for a file associated with a specified file pointer.
`fseek`	Sets the file-position indicator on a file associated with a specified file pointer.

Table 10-6. File-Positioning Functions *(Page 2 of 2)*

Function	Description
`fsetpos`	Sets the file-position indicator, for a file associated with a specified file pointer, to an earlier position obtained by the `fgetpos` function.
`ftell`	Gets the current value of the file-position indicator for a file associated with a specified file pointer.
`rewind`	Sets the file-position indicator, for a file associated with a specified file pointer, to the beginning of the file.

Sample Program Using the Standard I/O Functions

The sample program in Figure 10-3 illustrates how to use the standard I/O functions to do the following:

- open a file
- get input from the terminal's keyboard
- write the entered data to a file
- read the data that has been written.

The record that is written to `rel_file` consists of three fields: two strings and a floating-point number. Each field is separated by one space character. The call-out numbers to the left of the source code correspond to the numbered explanations that follow Figure 10-3.

For detailed information on each of the functions called in the sample program, see the function descriptions in Chapter 11.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_RECS 100
#define OKAY 0
#define ERROR 1

void get_data(void);
int write_record(void);
void read_file_and_quit(void);

char last_name[15];                  /* Components of the record */
char first_name[15];
float amount;

FILE *file_ptr;

main()
{
   unsigned short rec_num = 0;
   int c, ret_value;
```

(Continued on next page)

```
   errno = 0;
1. if ((file_ptr = fopen("rel_file", "a+v 36")) == NULL) /* Relative file */
      {
      printf("Error opening rel_file: errno + %d\n”, errno);
      exit(ERROR);
      }

2. while (rec_num < MAX_RECS)
      {
      get_data();

      ret_value = write_record();

      if (ret_value == ERROR)
         {
         puts("Last record was not written. Enter the information again.");
         continue;
         }

      puts("To enter another record, press RETURN.");
      puts("To read the file’s contents and quit, enter Q.");

      c = getchar();
      if ( (c == ’q’)  (c == ’Q’) )
         {
         read_file_and_quit();
         break;
         }
      else
         rec_num++;                            /* Increment rec_num for next record */
         }
      }



3. void get_data(void)
   {
      char buffer[30];

      printf("**** MEMBER INFORMATION ****\n\n");

      printf("Enter last name: ");
      if ( strlen(gets(buffer)) > 0 )
            strncpy(last_name, buffer, 15);

      printf("Enter first name: ");
      if ( strlen(gets(buffer)) > 0 )
         strncpy(first_name, buffer, 15);

      printf("Enter amount: ");
      if ( strlen(gets(buffer)) > 0 )
         amount = (float)atof(buffer);
   }

   int write_record(void)
   {
      int ret_value;
      static int number;
```

(Continued on next page)

```
4.    ret_value = fprintf(file_ptr, "%s %s %6.2f\n", last_name, first_name, amount);
      if (ret_value < 0)
         return(ERROR);
      else
         {
         printf("\nTotalrecords written: %d\n\n", (++number));
         return(OKAY);
         }
   }

   void read_file_and_quit(void)
   {
      unsigned short count = 0;
      int items_read;
      char l_name[15];
      char f_name[15];
      float amt;

      errno = 0;
5.    rewind(file_ptr);           /* Position to beginning of file */
      if (errno != 0)
         {
         printf("Error positioning to beginning-of-file: errno = %d\n", errno);
         exit(ERROR);
         }

      printf("**** DATA CONTAINED IN FILE ****\n\n");

      while (1)
         {
6.       items_read = fscanf(file_ptr, "%s %s %f\n", l_name, f_name, &amt);

         if (items_read == EOF)                    /* Check for end of record */
            break;

         if (items_read != 3)                         /* Check for read error */
            {
            printf("Error reading record number #%d\n", count);
            exit(ERROR);
            }

         printf("Record #%d = %s %s: %6.2f\n", count, f_name,l_name, amt);

         count++;
         }
         errno = 0;
7.       if ( fclose(file_ptr) != 0 )
            printf("Error closing rel_file: errno = %d\n", errno);
      }
```

Figure 10-3. Standard I/O Sample Program

The source code in Figure 10-3 uses the standard I/O functions to do the following tasks.

1. The `fopen` function opens `rel_file`. The `+` in the second argument to `fopen` specifies that the file will be opened for update: reading and writing. In addition, the `a` in the second argument specifies the file will be opened in append mode: all writing takes place at the end of the file. If `rel_file` does not exist, `fopen` creates the file. The `v 36` in the second argument to `fopen` is a VOS C extension that tells `fopen` to create a file with relative file organization and a maximum record size of 36 bytes if `rel_file` does not exist.

   If the file is successfully opened, `fopen` returns `file_ptr`. The program uses this file pointer to specify `rel_file` in all subsequent standard I/O function calls.

2. The `while` statement creates a loop that does the following:

   - calls the `get_data` function so that the user can enter input data for a record
   - calls the `write_record` function to write the record to `rel_file`.

   If the user enters `Q` or `q`, the program calls `read_file_and_quit` to read the data that has been written to `rel_file` and to end the program. If the user enters any character other than `Q` or `q`, the `while` loop continues, and the user can enter another record.

3. The `get_data` function uses a string I/O function, `gets`, to get three strings that the user enters through the terminal's keyboard. If the length of each entered string is greater than 0, the entered data is assigned to the appropriate variable. In a real-world application, the program would need to perform more checks on the input data. Without these additional checks, the information stored in `rel_file` would probably be error-filled.

4. When the `write_record` function is called, the formatted I/O function, `fprintf`, uses the following format string to write each record to `rel_file`, the file associated with `file_ptr`.

   ```
   "%s %s %6.2f\n"
   ```

   The format string specifies a single line of ASCII text per record with one space character between each field of the record. A newline character terminates the record. The `float` value has a minimum width of six characters with two digits to the right of the decimal point.

   If an error occurs during the write operation, `fprintf` returns a negative value. In this case, the `write_record` function returns `ERROR` to `main` to communicate that an error has occurred.

5. When the `read_file_and_quit` function is called, the `rewind` function sets the file-position indicator for `rel_file`, the file associated with `file_ptr`, to the beginning of the file. The program sets the file-position to beginning-of-file so that the contents of the file can be read from beginning to end. Also, when a file is opened for update I/O as was `rel_file` in the earlier call to `fopen`, an output operation cannot

be followed by an input operation without an intervening call to `fflush`, `fseek`, `fsetpos`, or `rewind`.

Because `rewind` itself returns no value, the program assigns the value 0 to the external variable `errno` prior to calling `rewind` and checks the value of `errno` after calling the function to ensure that no error has occurred during the positioning operation. The program must explicitly set `errno` to the value 0 because no library function sets `errno` to 0 when it is called.

6. Another formatted I/O function, `fscanf`, reads each record in `rel_file` until end-of-file is encountered. The format string used with `fscanf` is similar to the format string that was used with `fprintf` when data was written to the file. One exception: the `fscanf` call omits the field width and precision, which was `6.2` in the `fprintf` call. The `fscanf` call will read the `float` item regardless of its width. Although it has meaning in the `fprintf` call, the precision of `.2` is not valid in an `fscanf` conversion specification.

In the `fscanf` call, notice that the `amt` argument is preceded by the address-of operator (`&`). Every `fscanf` conversion specifier, such as `%f`, requires a corresponding argument that is a pointer to the variable where `fscanf` will assign the input item.

The return value of the `fscanf` call is assigned to `items_read`.

- If `items_read` equals `EOF`, `fscanf` encountered end-of-file. Execution of the `while` statement used to read records from the file is terminated.
- If `items_read` is not equal to the value 3, `fscanf` has not successfully read a complete record. The program writes an error message to `stdout` and calls the `exit` function to terminate the program.
- If `items_read` is equal to the value 3, `fscanf` successfully read a complete record. That is, `fscanf` assigned three input items to arguments.

7. The `fclose` function closes `rel_file`, the file associated with `file_ptr`. If `fclose` returns the value 0, it has successfully closed `rel_file`. If `fclose` does not return the value 0, it has not successfully closed the file. In the case of an error, `printf` writes an error message and the value of `errno` to `stdout`. The value stored in `errno` is the appropriate error code number.

UNIX I/O Functions

The UNIX I/O functions are **not** part of the ANSI definition of the C library functions. These functions are included in the VOS C library so that an existing program that uses them will be compatible with the VOS implementation of the C language. Table 10-7 lists the UNIX I/O functions. Each UNIX I/O function is declared in the `c_utilities.h` header file.

Table 10-7. UNIX I/O Functions

Function	Description
`close`	Closes the file associated with a specified file descriptor.
`creat`	Creates a new file or truncates an existing file, and associates a file descriptor with the file.
`lockf`	Performs various region-locking actions on a stream file associated with a specified file descriptor.
`lseek`	Sets the file-position indicator on a file associated with a specified file descriptor.
`open`	Opens a file and associates a file descriptor with the file.
`read`	Reads a specified number of bytes of data from a file associated with a given file descriptor.
`write`	Writes a specified number of bytes of data to a file associated with a given file descriptor.

In VOS C, the UNIX functions, such as `read` and `write`, are fully buffered by default. That is, data is read from or written to the file using an intermediate I/O buffer as is used with the standard I/O functions. See the "Buffered Input and Output" section earlier in this chapter for information on buffering.

In some other implementations, the UNIX functions provide unbuffered, low-level I/O. In these non-VOS implementations, no intermediate I/O buffer is used and the functions directly access the operating system's kernel routines. However, in VOS C, the UNIX functions are fully buffered and do **not** directly access the operating system's kernel routines. In VOS C, the UNIX functions, such as `read` and `write`, call the operating system's `s$` subroutines to access the kernel entry points.

If you want unbuffered I/O on a file, you can change the buffering method from fully buffered to unbuffered by using the `setbuf` or `setvbuf` library function. These two functions require that you specify a file pointer to identify the file. To get the file pointer for the file associated with a specified file descriptor, you can use the `fdopen` function. See the descriptions of `setbuf` and `setvbuf` in Chapter 11 for more information on how the buffering method for a file is modified.

Differences between Standard I/O and UNIX I/O

There are two major differences between the standard I/O functions and the UNIX I/O functions.

The first major difference relates to how you specify a file with the UNIX I/O functions. To indicate the file upon which the I/O operation is to be performed, you use an integer *file descriptor* (sometimes called a file handle) rather than a file pointer as is done with the standard I/O functions. This file descriptor is the value returned by `creat`, `fileno`, or `open`. When a file is opened, the system associates the lowest unused file descriptor value with the file. Usually the lowest unused file descriptor is the value 3 because three file descriptors are already associated with the standard files. The preopened, standard files have the following file descriptor values associated with them: 0 for `stdin`, 1 for `stdout`, and 2 for `stderr`.

The second major difference between the standard I/O functions and the UNIX functions is that the UNIX I/O functions are not well suited for I/O involving text. For example, the UNIX functions do not provide for formatted I/O or string I/O as do the standard I/O functions. Instead of processing text, the UNIX functions are often used in binary mode.

Sample Program Using the UNIX I/O Functions

The sample program in Figure 10-4 illustrates how to use the UNIX I/O functions to do the following:

- open two files
- read data from one file
- write data to another file.

The sample program copies the contents of `source_file` to `target_file`. The call-out numbers to the left of the source code correspond to the numbered explanations that follow Figure 10-4.

For detailed information on each of the functions called in the sample program, see the function descriptions in Chapter 11.

```
#include <c_utilities.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

void exit_func(char *message);

int file_des_1, file_des_2;
FILE *file_ptr_1, *file_ptr_2;

int mode, bytes_read, bytes_written, total_bytes_copied;
char *type, buffer[10000];

main()
```

(Continued on next page)

```
      {
1.       if ( (file_des_1 = open("source_file", O_RDONLY||O_BINARY, mode)) == -1 )
            exit_func("Error opening source_file");

2.       if ( (file_des_2 = open("target_file", O_WRONLY  O_BINARY O_TRUNC, mode)) == -1 )
            exit_func("Error opening target_file");

3.       if ( (file_ptr_1 = fdopen(file_des_1, type)) == NULL )
            exit_func("Error getting file pointer for source_file");

         if ( (file_ptr_2 = fdopen(file_des_2, type)) == NULL )
            exit_func("Error getting file pointer for target_file");

4.       if ( (setvbuf(file_ptr_1, NULL, _IONBF, 0)) == -1 )
            puts("Could not turn off buffering on source_file");

         if ( (setvbuf(file_ptr_2, NULL, _IONBF, 0)) == -1 )
            puts("Could not turn off buffering on target_file");

         puts("Copying the contents of source_file to target_file ...");

5.       while ( bytes_read = read(file_des_1, buffer, 10000) )
            {
            if (bytes_read == -1)
               exit_func("Read operation failed");

6.       if ( (bytes_written = write(file_des_2, buffer, bytes_read)) == -1 )
            exit_func("Write operation failed");

         total_bytes_copied += bytes_written;

         if (bytes_read < 10000)
            break;
         }

         printf("Done: %d bytes copied to target_file.\n”, total_bytes_copied);

7.       if ( close(file_des_1) == -1 )
            exit_func("Error closing source_file");

         if ( close(file_des_2) == -1 )
         exit_func("Error closing target_file");
      }

      void exit_func(char *message)
      {
         perror(message);
         exit(1);
      }
```

Figure 10-4. UNIX I/O Sample Program

The source code in Figure 10-4 uses the UNIX I/O functions to do the following tasks.

1. The first open function opens source_file for input (O_RDONLY) in binary mode (O_BINARY). If the file is successfully opened, open returns file_des_1. The program uses this file descriptor to specify source_file in subsequent UNIX I/O function calls.

2. In a similar manner, the second open function opens target_file for output (O_WRONLY) in binary mode (O_BINARY). The O_TRUNC flag tells open that the previous contents of target_file should be deleted. If the file is successfully opened, open returns file_des_2. The program uses this file descriptor to specify target_file in subsequent UNIX I/O function calls.

3. The two fdopen function calls get the file pointers associated with source_file and target_file. If each fdopen call succeeds in getting the file pointers, the calls return file_ptr_1 and file_ptr_2.

   In general, performing I/O with a mixture of the standard I/O functions, such as setvbuf, and the UNIX I/O functions, such as read, is not recommended. The use of fdopen and setvbuf is an exception. In VOS C, all files except the standard files are opened for fully buffered I/O. This program must use the standard I/O function, setvbuf, to turn off buffering. To use setvbuf, the program needs the two file pointers returned by the fdopen calls.

4. The two setvbuf function invocations turn off buffering for source_file and target_file, the files associated with the file pointers file_ptr_1 and file_ptr_2. Each call specifies that I/O with the given file should be unbuffered (_IONBF). It is not necessary that I/O on a file be unbuffered to use the UNIX I/O functions. However, in this particular application, unbuffered I/O may be faster than buffered I/O. The discussion later in this section explains the possible advantages of using unbuffered I/O for copying files.

5. The read function attempts to read the contents of source_file in chunks of 10,000 bytes and writes that data into buffer. If the read operation succeeds, read returns the number of bytes that have been read from the file and written to buffer. The number of bytes that have been read will be less than 10,000 if the read function encounters the end of source_file before reading 10,000 bytes.

   Notice that with the UNIX I/O functions, such as read and write, you specify a file with a file descriptor, not a file pointer. This file descriptor is returned by a previous call to creat, fileno, or open. In this example, the file_des_1 file descriptor indicates that the read operation will be performed on source_file.

6. After data from source_file has been read into buffer, the write function reads, from buffer, the number of bytes specified in bytes_read and writes that data into target_file. As with the read function, you specify a file with a file descriptor, not a file pointer. If the write operation succeeds, write returns the number of bytes that have been written to target_file.

Because the `read` function call is used as the controlling expression in the `while` statement and the `write` function call is part of `while`'s loop, the program can call `read` and `write` multiple times.

- If the return value of `read` is -1, a read error has occurred, and `exit_func` is called, terminating the program.
- If the return value of `write` is -1, a write error has occurred, and `exit_func` is called, terminating the program.
- If the return value of `read` is less than 10,000, the end of the file has been reached, and the `break` statement terminates execution of the `while` statement.

7. The two `close` function calls close `source_file` and `target_file`. If either `close` function cannot close the file, the program calls `exit_func`, which generates an error message and terminates the program.

Using Unbuffered I/O. The following discussion explains how the use of unbuffered I/O in the preceding sample program differs from the use of fully buffered I/O.

The sample program in Figure 10-4 copies `source_file` by reading and then writing 10,000 bytes at a time while both the source and target files are opened for unbuffered I/O. This chunk of 10,000 bytes is significantly greater than the size of the I/O buffer associated with the file. In this case, a stream file's I/O buffer is, by default, 512 bytes. Thus, using unbuffered I/O means that the program requires fewer disk accesses. This use of unbuffered I/O to copy large chunks of bytes at one time is more efficient than buffered I/O if `source_file` contains large amounts of data.

With unbuffered I/O, each time the program calls the `read` and `write` functions four read-write operations occur. The `read` function does the following:

1. reads data from `source_file`
2. writes the data directly into the program-defined buffer specified in `read`.

With unbuffered I/O, the `write` function does the following:

1. reads data from the program-defined buffer specified in `write`
2. writes the data directly from the program-defined buffer into `target_file`.

In contrast, with fully buffered I/O, each time the program calls the `read` and `write` functions eight read-write operations occur. The `read` function does the following:

1. reads data from `source_file`
2. writes the data into the system-allocated I/O buffer associated with `source_file`
3. reads the data from the I/O buffer
4. writes the data into the program-defined buffer specified in `read`.

With fully buffered I/O, the `write` function does the following:

1. reads data from the program-defined buffer specified in `write`
2. writes the data into the system-allocated I/O buffer associated with `target_file`
3. reads the data from the I/O buffer
4. writes the data into `target_file`.

The ANSI-C-compatible standard I/O functions `fread` and `fwrite` could be used for unbuffered I/O in much the same way as the sample program (Figure 10-4) uses the UNIX I/O functions `read` and `write`.

VOS C I/O Function Summary

Table 10-1 is a summary of the VOS C standard I/O and UNIX I/O functions.

Table 10-8. VOS C I/O Functions Summary *(Page 1 of 2)*

Task	**Standard I/O**	**UNIX I/O**
Opening a file	`fopen` `freopen`	`open`
Creating a file	`fopen` `tmpfile`	`open` `creat`
Closing a file	`fclose`	`close`

Positioning in a file	`fseek`
	`ftell`
	`fgetpos`
	`fsetpos`
	`rewind`
		`lseek`

Binary I/O	`fread` `fwrite`	`read` `write`

Single character I/O	`fgetc`
	`fputc`
	`getc`
	`getchar`
	`putc`
	`putchar`
	`ungetc`
		None

String I/O	`fgets`
	`fputs`
	`gets`
	`puts`
		None

Table 10-8. VOS C I/O Functions Summary *(Page 2 of 2)*

Task	Standard I/O	UNIX I/O

Formatted I/O	`fprintf`
	`fscanf`
	`printf`
	`scanf`
	`sprintf`
	`sscanf`
		None

Region locking	None	`lockf`

In addition to the functions shown in Table 10-1, you can use a function and a macro declared in the `stdio.h` header file if you need to get the file pointer or file descriptor associated with an open file. To get the file pointer for the file associated with a specified file descriptor, you can use the `fdopen` function. Conversely, to get the file descriptor for the file associated with a specified file pointer, you can use the `fileno` macro.

Chapter 11: VOS C Library Functions

Many of the tasks that are performed by the typical C program are not accomplished by using C language statements but by calling functions contained in the C library. This chapter explains how to use the VOS C library functions. It also describes the contents of their associated header files.

> **Note:** For Release 11.0, the version of the VOS C library functions described in this manual is **not** the version that is associated with the standard `stdio.h` header file or the standard object library, the `c_object_library`. To use the new version of the library functions (the version described in this manual), you must follow the procedure explained in the next section, “Using the New Version of the VOS C Library Functions.”

For Release 11.0, the VOS operating system includes two versions of the library functions.

- The *old version* of the library functions is associated with the `stdio.h` header file and the `c_object_library`.
- The *new version* of the library functions is associated with the `new_stdio.h` header file and the `new_c_object_library`. (The `new_stdio.h` header file and the `new_c_object_library` were available prior to Release 11.0.)

In a future release, VOS will not include the old version of the VOS C library functions as part of the release. Therefore, it is **highly recommended** that you use the new version of the library functions.

Using the New Version of the VOS C Library Functions

This section describes how to set up the link and object library path that are needed to use the new version of the VOS C library functions. It also explains how to convert an existing program over to the new version of the library functions.

> **Note:** The procedure described in this section **may not be applicable** to releases of the operating system after Release 11.0. For a release of the operating system after Release 11.0, see the appropriate *Software Release Bulletin* for information on the procedure, if any, that you perform to use the new version of the library functions.

The new version of the library functions requires a different, incompatible version of the `stdio.h` header at compile time and a different set of object modules at bind time. The procedure to convert an existing program over to the new version of the library functions depends on the version of the library functions that was used when the program was originally compiled and bound. In addition to compiling and binding any new source modules with the new version of the library, you must do one of the following:

- If you are modifying an existing program that was compiled and bound with the **old version** of the library functions, all previously existing source modules must be recompiled, and the resulting object modules must be rebound.
- If you are modifying an existing program that was compiled and bound with the **new version** of the library functions (the version available prior to Release 11.0), no recompilation of existing source modules is needed, but all existing object modules must be rebound.

To set up the link and object library path for the new version of the library functions, perform the following procedure.

1. Make sure that you are in the directory where you will compile the source modules.
2. Use the `link` command to create a link from `stdio.h` to `new_stdio.incl.c`, which is located in the `>system>include_library` directory. To create the link, enter the following:

   ```
   link  >system>include_library>new_stdio.incl.c  stdio.incl.c
   ```

3. Use the `delete_library_path` command to remove the old `c_object_library` directory from the object library paths list for your process. To remove the directory, enter the following:

   ```
   delete_library_path  object  >system>c_object_library
   ```

4. Use the `add_library_path` command to add the `new_c_object_library` directory to the object library paths list for your process. To add the directory, enter the following:

   ```
   add_library_path  object  >system>new_c_object_library
   ```

5. Compile all new source modules and, if the old version of the library functions was used in the original compilation, all previously existing source modules.
6. Bind all resulting object modules into a program module.

To make future `stdio.h` header file adjustments easier, it is **recommended** that you create a link to the `new_stdio.h` header file rather that specifying the name of the new version in each source module. Also, if it is appropriate, your system administrator can create the required link and change the object library paths list for an entire module.

Library Function Header Files

All library functions, `#define` macros, and any additional types needed for their use are declared or defined in a header file. A *header file* is a file that the compiler incorporates into the source module. An `#include` directive, specifying the header file's name, is typically placed at the beginning of the source module, hence such a file is often simply called a “header.” A header file is sometimes called an include file.

External identifiers declared in any of the header files are reserved, whether or not the associated file is actually included into the source module. All `extern` identifiers that begin with an underscore are also reserved. If the program redefines a reserved `extern` identifier, even with a semantically equivalent form, the behavior is unpredictable.

Table 11-1 lists the header files contained in the VOS C library. The table also shows what section in this chapter contains information on the header file.

Table 11-1. VOS C Header Files

For Information On This Header	**See This Section in Chapter 11**
`alloca.h`	“Temporary Memory Allocation”
`assert.h`	“Diagnostics”
`ctype.h`	“Character Handling”
`c_utilities.h`	“UNIX Functions”
`fcntl.h`	“UNIX Functions”
`float.h`	“Limits”
`limits.h`	“Limits”
`math.h`	“Mathematics”
`new_stdio.h`	“Standard Input and Output”
`setjmp.h`	“Nonlocal Jumps”
`signal.h`	“Signal Handling”
`stdarg.h`	“Variable Arguments”
`stddef.h`	“Common Definitions”
`stdio.h`	“Standard Input and Output”
`stdlib.h`	“General Utilities”
`string.h`	“String Manipulation”
`time.h`	“Date and Time”

Each header file resides in the `>system>include_library` directory. The `system` directory is one level below the master disk in the directory hierarchy. Within the `include_library` directory, each header file `*header*.h` is named `*header*.incl.c`. For example, within the `include_library` directory, the `stdlib.h` header file is named `stdlib.incl.c`. See the “Source File Inclusion” section in Chapter 9 for information on the `#include` directive and the rules that the compiler uses to search for header files.

Headers can be included in any order. Within a given scope, a header can occur in more than one `#include` directive. However, only one copy of the file is incorporated into the source module. When a header is used, it must be included before the first reference to any of the functions or objects it declares or to any of the types or macros it defines.

Using the Library Functions

When using the VOS C library functions, you should be aware of the following considerations. Each of the following statements applies unless explicitly stated otherwise in the description of the function that appears in this chapter.

- If a function argument has an invalid value, such as a value outside the function's domain or a pointer outside the program's address space, the behavior is unpredictable.
- If a function argument is described as an array, the pointer actually passed to the function must have a value such that all address computations and accesses to objects, which would be valid if the pointer did point to the first element of the array, are valid.
- If a library facility is described as a "function," the facility is guaranteed to be implemented as a function though it may additionally be implemented as a macro. When a library facility is implemented as both a function and a macro, the macro is the default.
- Unless otherwise specified, any macro definition of a function expands to code that evaluates each of its arguments exactly once, protected by parentheses where necessary. Therefore, unless otherwise specified, as in the function descriptions for `getc` and `putc`, it is safe to use arbitrary expressions as arguments to library functions or macros.
- Because a function declared in a header may also be defined as a macro in the header, you should **not** declare a library function if the function's header has been included into the source module.
- If a function can be declared without reference to any type defined in a header file, you can declare the function, either explicitly or implicitly, and use it without including its associated header. However, this practice is not recommended.
- If a library facility is described as a "macro," the facility can be invoked in an expression anywhere a function with a compatible return type could be called.

Finally, for a function that accepts a variable number of arguments, if you do not declare the function explicitly or include its header file into the source module, the behavior is unpredictable.

Suppression of Macro Definitions

A library facility described as a function is guaranteed to be implemented as a function though it may additionally be implemented as a macro. To ensure that an actual function is referenced, you can suppress a macro definition of the function by enclosing the function name in parentheses. The C preprocessor does not expand a function-like macro if the macro name is not immediately followed by an opening parenthesis (`(`). For the same reason, you can take the address of a library function even if it is also defined as a macro.

For example, the ANSI C Standard allows VOS C to implement the `abort` function as both a function and a macro, or to implement `abort` as a function only. If a function-like macro named `abort` were defined in the `stdlib.h` header file, the macro implementation would be the default implementation of `abort`. With these assumptions in mind, consider the following references to `abort`.

```
void (*ptr)(void);

(abort)();

ptr = &abort;
```

In the preceding examples, the function-call operation `(abort)()` is guaranteed to invoke an actual function because the C preprocessor does not expand a function-like macro if the macro name is not immediately followed by an opening parenthesis. The address-of operation `&abort` is guaranteed to yield the address of an actual function for the same syntactic reason.

In addition, you can use the `#undef` preprocessor directive to undefine any macro definition, ensuring that an actual function is referenced. The `#undef` directive must appear **after** the `#include` directive that has incorporated the function's header file into the source module. It is not an error to `#undef` a name that has not appeared previously in a `#define` directive. When you use `#undef` to undefine a macro definition, the prototype for the function, which precedes and is hidden by the macro definition, is once again in effect.

Examples of Library Function Use

The examples in this section illustrate a number of ways that you can incorporate the declaration of a library function (or macro) into a source module. The examples use the `atoi` function from the `stdlib.h` header file.

When you include `atoi`'s header file, an actual function reference or a macro expansion is possible. For example:

```
#include <stdlib.h>
char *str;
   .
   .
   .
i = atoi(str);
```

When you use `atoi`'s header file and then undefine any possible macro definition, an actual function reference is guaranteed. For example:

```
#include <stdlib.h>
#undef atoi
char *str;
   .
   .
   .
i = atoi(str);
```

When you explicitly declare `atoi`, any macro definition in the header file is not incorporated into the source module. Thus, an actual function reference is guaranteed. For example:

```
extern int atoi(const char *);
char *str;
   .
   .
   .
i = atoi(str);
```

When you implicitly declare `atoi`, any macro definition in the header file is not incorporated into the source module. Thus, an actual function reference is guaranteed. For example:

```
char *str;
   .
   .
i = atoi(str);
```

In the preceding example, no function prototype for `atoi` will appear in the source module. Explicitly or implicitly declaring a library function is **not** recommended.

Temporary Memory Allocation

The `alloca.h` header file declares the `alloca` function, which you use to allocate temporary storage in the stack frame of the calling function. Table 11-2 lists the function contained in the `alloca.h` header file.

Table 11-2. Memory Allocation Function

Function Prototype	Description
`void *alloca(int size);`	Allocates a specified number of bytes of temporary storage.

Because the `alloca` function is **not** part of the ANSI C Standard's definition of the library functions, using `alloca` can affect your program's portability. The `alloca` function is included in the VOS C library so that existing programs that use `alloca` will be compatible with the VOS implementation of the C language.

Diagnostics

The `assert.h` header file declares the `assert` macro and references the `NDEBUG` macro, both of which you use to generate diagnostic information when debugging a program. The `NDEBUG` macro is used in the following manner to disable or enable the `assert` macro.

- If you define the `NDEBUG` macro as a macro name prior to the point where an `#include` directive incorporates `assert.h` into the source module, the `assert` macro is disabled.

- If you do **not** define the `NDEBUG` macro as a macro name prior to the point where an `#include` directive incorporates `assert.h` into the source module, the `assert` macro is enabled. That is, if you do nothing, the `assert` macro is enabled.

The `assert.h` header file does **not** define `NDEBUG` as a macro name. Typically, a programmer uses the `assert` macro for debugging purposes during the development of a program. After the program has been debugged, the programmer defines `NDEBUG` as a macro name to disable the diagnostics that the `assert` macro generates. To define `NDEBUG` and disable the `assert` macro, insert the following `#define` directive, as shown, prior to the `#include` directive for the `assert` header.

```
    #define NDEBUG

    #include <assert.h>
```

Table 11-3 lists the macro contained in the `assert.h` header file.

Table 11-3. Diagnostic Macro

Function Prototype	**Description**
`void assert(int` *`expression`*`);`	If a specific expression is false, writes diagnostic information to the standard error file, `stderr`, and aborts the program.

Character Handling

The `ctype.h` header file declares several functions and defines several macros that you use for classifying and converting characters. \tableafternext lists the functions and macros contained in the `ctype.h` header file. The functions and macros in this header allow you to do the following:

- classify ASCII characters according to their type, such as classifying a character as a decimal digit
- perform certain character conversions, such as converting an uppercase letter to a lowercase letter.

The ASCII characters have codes in the range 0 through 127. Codes in this range include characters that can be printed, called *printable characters*, and characters that cannot be printed, called *control codes*. Table 11-4 shows the categories into which ASCII characters can be classified.

Table 11-4. Classification Categories for ASCII Characters *(Page 1 of 2)*

Classification Category	**Description**
Uppercase letters	`A` through `Z`
Lowercase letters	`a` through `z`
Decimal digits	`0` through `9`
Octal digits	`0` through `7`
Hexadecimal digits	`0` through `0`, `A` through `F`, and `a` through `f`

Table 11-4. Classification Categories for ASCII Characters *(Page 2 of 2)*

Classification Category	**Description**
Control characters	Any character that cannot be printed: ASCII character codes 0 through 31, and code 127
Punctuation characters	Any printable character **except** a space, decimal digit, or letter
White-space characters	Horizontal tab, newline, vertical tab, form feed, carriage return, and space

Except for `isascii` and `toascii`, each function and macro in the `ctype.h` header file expects an `int` argument in the range 0 through 255, or the argument must be equal to the constant `EOF` (end-of-file). If the argument is any other value, the functions and macros produce **invalid** results. The `isascii` and `toascii` functions are exceptions. Each takes as an argument any integer value, including the constant `EOF`.

Two character-conversion functions in the `ctype.h` header file, `tolower` and `toupper`, are also defined as macros. The macros `_tolower` and `_toupper` are, in general, faster than the functions. However, neither macro tests whether the character passed as an argument is a valid ASCII character. If this test is needed, the program must perform it.

> **Note:** If you use the `_tolower` or `_toupper` macro, each macro can yield unexpected results if the argument you pass to it is an expression, such as a function invocation, that has side effects.

Table 11-5. Character-Handling Functions and Macros *(Page 1 of 2)*

Function Prototype	**Description**
`int isalnum(int c);`	Tests whether an ASCII character is a letter or digit.
`int isalpha(int c);`	Tests whether an ASCII character is a letter.
`int isascii(int c);`	Tests whether an integer value is a valid ASCII character.
`int iscntrl(int c);`	Tests whether an ASCII character is a control character.
`int isdigit(int c);`	Tests whether an ASCII character is a decimal digit.
`int isgraph(int c);`	Tests whether an ASCII character is a printable character other than a space.
`int islower(int c);`	Tests whether an ASCII character is a lowercase letter.
`int isoctal(int c);`	Tests whether an ASCII character is an octal digit.
`int isprint(int c);`	Tests whether an ASCII character is a printable character including the space character.
`int ispunct(int c);`	Tests whether an ASCII character is a punctuation character.
`int isspace(int c);`	Tests whether an ASCII character is a white-space character.

Table 11-5. Character-Handling Functions and Macros *(Page 2 of 2)*

Function Prototype	Description
`int isupper(int c);`	Tests whether an ASCII character is an uppercase letter.
`int isxdigit(int c);`	Tests whether an ASCII character is a hexadecimal digit.
`int toascii(int c);`	Sets all but the rightmost seven bits of an integer variable to 0 so that the resulting value represents a valid character in the ASCII character set.
`int tolower(int c);`	Converts an uppercase ASCII character to lowercase.
`int _tolower(int c);`	Converts an uppercase ASCII character to lowercase.
`int toupper(int c);`	Converts a lowercase ASCII character to uppercase.
`int _toupper(int c);`	Converts a lowercase ASCII character to uppercase.

UNIX Functions

The `c_utilities.h` header file declares several functions that you use for I/O and other purposes. Table 11-6 provides a summary of the functions contained in the `c_utilities.h` header file. The majority of functions in this header allow you to perform input and output using the UNIX I/O functions: `close`, `creat`, `lseek`, `open`, `read`, and `write`. The functions in the `c_utilities.h` header are VOS C extensions that allow programs written for C under UNIX to be compatible with VOS C.

Note: Because the functions in the `c_utilities` header file are not part of the ANSI C Standard's definition of the library functions, using these functions can affect your program's portability.

Table 11-6. UNIX Functions *(Page 1 of 2)*

Function Prototype	Description
`int access(char *path_name, int mode);`	Checks to determine whether a file or directory exists and whether the caller has a specified type of access.
`int chdir(char *path_name);`	Changes the current directory to the directory specified.
`int close(int file_des);`	Closes the file associated with a specified file descriptor.
`int creat(char *path_name, int mode);`	Creates a new file or truncates an existing file, and associates a file descriptor with the file.
`int isatty(int file_des);`	Tests whether a specified file descriptor is associated with a terminal device.

Table 11-6. UNIX Functions *(Page 2 of 2)*

Function Prototype	Description
`int lockf(int file_des, int command, long size);`	Performs various region-locking actions. on a stream file associated with a specified file descriptor.
`int lseek(int file_des, long offset, int whence);`	Sets the file-position indicator on a file associated with a specified file descriptor.
`int open(char *path_name, int o_flag, ...);`	Opens a file and associates a file descriptor with the file.
`int read(int file_des, char *buffer, unsigned int nbytes);`	Reads a specified number of bytes of data from a file associated with a given file descriptor.
`int sleep(unsigned int seconds);`	Suspends the calling process for a specified number of seconds.
`int write(int file_des, char *buffer,unsigned int nbytes);`	Writes a specified number of bytes of data to a file associated with a given file descriptor.

The UNIX I/O functions shown in Table 11-6 are different from the standard I/O functions declared in the `stdio.h` header file.

One difference between the UNIX I/O functions and the standard I/O functions is that the UNIX I/O functions use a file descriptor, `file_des`, to specify which file to access. This *file descriptor* is a non-negative integer value returned from a call to the `creat` or `open` function, or from the `fileno` macro of the `stdio.h` header file. This file descriptor is a distinct entity from the file pointer returned by, for example, the `fopen` or `freopen` function.

If you are using the `open` function to open a file, you must include the `fcntl.h` header file into the source module so that you can specify a defined constant, such as `O_RDONLY`, for the `o_flag` argument. The `fcntl.h` header file declares the defined constants that are allowed in the `o_flag` argument. See the description of the `open` function for more information on the flags contained in the `fcntl.h` header file.

The `c_utilities.h` header defines six constants (macros) that can be specified with the `lockf` function.

- `F_TEST`
- `F_LOCK`
- `F_TLOCK`
- `F_RLOCK`
- `F_TRLOCK`
- `F_ULOCK`

See the description of `lockf`, later in this chapter, for information about these constants.

See Chapter 10 for more information on the UNIX I/O functions.

Limits

The `limits.h` and `float.h` header files declare several macros that expand to constants for various limits and parameters.

The `limits.h` header file defines certain VOS C limits for the integral types. Table 11-7 lists the defined constants (macros) that are declared in the `limits.h` header.

Table 11-7. Integral Limits

Defined Constant	Value	Description
`CHAR_BIT`	8	Number of bits in a `char`
`CHAR_MIN`	0	Minimum value for `char`
`CHAR_MAX`	255	Maximum value for `char`
`UCHAR_MAX`	255	Maximum value for `unsigned char`
`SCHAR_MIN`	-128	Minimum value for `signed char`
`SCHAR_MAX`	127	Maximum value for `signed char`
`SHRT_MAX`	32,767	Maximum value for `short int`
`SHRT_MIN`	-32,768	Minimum value for `short int`
`USHRT_MAX`	65,535	Maximum value for `unsigned short int`
`INT_MAX`	2,147,483,647	Maximum value for `int`
`INT_MIN`	-2,147,483,648	Minimum value for `int`
`UINT_MAX`	4,294,967,295	Maximum value for `unsigned int`
`LONG_MAX`	2,147,483,647	Maximum value for `long int`
`LONG_MIN`	-2,147,483,648	Minimum value for `long int`
`ULONG_MAX`	4,294,967,295	Maximum value for `unsigned long int`

The `float.h` header file defines certain VOS C limits for the floating-point types. Table 11-8 lists the defined constants (macros) that are declared in the `float.h` header.

Table 11-8. Floating-Point Limits *(Page 1 of 2)*

Defined Constant	Value	Description
`FLT_RADIX`	2	Radix of exponent representation
`FLT_ROUNDS`	1	Rounding mode used for floating-point addition
`FLT_GUARD`	1	Guard digits used for floating-point multiplication
`FLT_NORMALIZE`	1	Normalized form used for floating-point values

Table 11-8. Floating-Point Limits *(Page 2 of 2)*

Defined Constant	Value	Description
`FLT_MAX_EXP`	38	Maximum power of 10 that can be represented in a `float`
`FLT_MIN_EXP`	-38	Minimum power of 10 that can be represented in a `float`
`FLT_DIG`	6	Maximum number of significant digits that can be represented in a `float`
`DBL_MAX_EXP`	307	Maximum power of 10 that can be represented in a `double`
`DBL_MIN_EXP`	-308	Minimum power of 10 that can be represented in a `double`
`DBL_DIG`	15	Maximum number of significant digits that can be represented in a `double`

For information on how the integral and floating-point types are stored internally, see Appendix A.

Mathematics

The `math.h` header file declares several functions and defines several macros that you use for mathematical routines, such as finding an absolute value, and for common mathematical functions, such as the trigonometric functions. Table 11-9 lists these functions by the category of task that each performs.

Table 11-9. Mathematics Functions by Category

Category	Functions
Absolute value, nearest integer, and remainder functions	`abs, ceil, fabs, floor, fmod`
Exponential and logarithmic functions	`exp, frexp, ldexp, log, log10, modf`
Hyperbolic functions	`cosh, sinh, tanh`
Power functions	`pow, sqrt`
Trigonometric functions	`acos, asin, atan, atan2, cos, sin, tan`

Table 11-10 provides a summary of the functions contained in the `math.h` header file.

Table 11-10. Mathematics Functions *(Page 1 of 2)*

Function Prototype	**Description**
`int abs(int x);`	Computes the absolute value of a specified integer number.
`double acos(double x);`	Computes the arc cosine of a floating-point value in the range --1 through 1.
`double asin(double x);`	Computes the arc sine of a floating-point value in the range --1 through 1.
`double atan(double x);`	Computes the arc tangent of a floating-point value.
`double atan2(double y, double x);`	Computes the arc tangent of the ratio of two nonzero, floating-point values.
`double ceil(double x);`	Computes the smallest integer value that is greater than or equal to a specified floating-point number.
`double cos(double x);`	Computes the cosine of an angle, which is expressed in radians.
`double cosh(double x);`	Computes the hyperbolic cosine of a floating-point value.
`double exp(double x);`	Computes the exponential function of a floating-point value.
`double fabs(double x);`	Computes the absolute value of a specified floating-point number.
`double floor(double x);`	Computes the largest integer value that is less than or equal to a specified floating-point number.
`double fmod(double x, double y);`	Computes the remainder after dividing one floating-point value by another floating-point value.
`double frexp(double value, int *exp_ptr);`	Breaks down a floating-point value into a mantissa (range: 0.5 to less than 1.0) and an integer exponent.
`double ldexp(double x, int exp);`	Computes a floating-point value by multiplying a specified mantissa by a specified power of 2.
`double log(double x);`	Computes the natural logarithm of a floating-point value.

Table 11-10. Mathematics Functions *(Page 2 of 2)*

Function Prototype	**Description**
`double log10(double x);`	Computes the base-10 logarithm of a floating-point value.
`double modf(double value, double *iptr);`	Breaks down a floating-point value into its integer and fractional parts.
`double pow(double x, double y);`	Computes the value of one argument raised to the power of another argument.
`double sin(double x);`	Computes the sine of an angle, which is expressed in radians.
`double sinh(double x);`	Computes the hyperbolic sine of a floating-point value.
`double sqrt(double x);`	Computes the square root of a non-negative, floating-point value.
`double tan(double x);`	Computes the tangent of an angle, which is expressed in radians.
`double tanh(double x);`	Computes the hyperbolic tangent of a floating-point value.

The external variable `errno` is defined in the `math.h` header file as an `extern int`. Some of the mathematics functions set `errno` to a positive error code when an error condition occurs. Each function's description specifies whether `errno` is used to indicate error conditions. At program startup, `errno` is initialized to the value 0. Library functions do **not** set `errno` to 0. Therefore, a program that uses `errno` for error checking should set it to 0 before a function call, and inspect the value of `errno` before any subsequent function call.

With the mathematics functions, two of the most common errors are domain and range errors. However, these functions can return other error code values as well.

For the mathematics functions, a *domain error* occurs when an input argument to the function is outside the domain over which the actual mathematical function (as opposed to the VOS implementation of the function) can provide a meaningful answer. When a domain error occurs, the math functions set the external variable `errno` to the value of the macro `EDOM`. The description of each function explains what value the function returns when a domain error occurs.

A *range error* occurs when the result of the function cannot be represented as a value of the type `double`. When a range error occurs, the math functions set the external variable `errno` to the value of the macro `ERANGE`. When the result creates an underflow condition, the function returns the value 0. When the result creates an overflow condition, the function returns the value of `HUGE_VAL` with the same sign as the correct value of the function.

The `math.h` header contains three macros that are used with the math functions. Table 11-11 lists each macro and its meaning.

Table 11-11. Mathematics Macros

Macro	Meaning
`EDOM`	When a function sets `errno` to `EDOM`, an argument to the math function is outside the function's domain. `EDOM` expands to `e$usf_EDOM` (`5033`).
`ERANGE`	When a function sets `errno` to `ERANGE`, the result of a math function is not representable by the return type of `double`. `ERANGE` expands to `e$usf_ERANGE` (`5034`).
`HUGE_VAL`	When a mathematics function's result creates an overflow condition, some functions return positive or negative `HUGE_VAL`.

Nonlocal Jumps

The `setjmp.h` header file declares two functions and one type that you use to achieve the effect of a nonlocal `goto` statement, thus bypassing the normal function call and return mechanism. Table 11-12 provides a summary of the functions contained in the `setjmp.h` header file.

Table 11-12. Nonlocal Jump Functions

Function Prototype	Description
`void longjmp(jmp_buf env, int val);`	Restores the calling environment saved in the program's most recent invocation of the `setjmp` function.
`int setjmp(jmp_buf env);`	Saves its calling environment for later use by the `longjmp` function.

In addition to the preceding functions, the `setjmp.h` header also defines the `jmp_buf` type as an array type suitable for holding the information needed to restore a calling environment.

Signal Handling

The `signal.h` header file declares two functions and defines several macros that you use to send and handle signals. A *signal* is an asynchronous interrupt for an error condition, or exception, that may be reported during program execution. Table 11-13 provides a summary of the functions contained in the `signal.h` header file.

Table 11-13. Signal-Handling Functions

Function Prototype	Description
`int kill(int process_id, int sig);`	Sends a specified signal to the calling program.
`void (*signal(int sig, (*func) (int))) (int);`	Determines how a specified signal will be handled.

In the VOS C environment, signal numbers range from 1 through 8. The `signal.h` header file contains a defined constant (macro) for each value. Table 11-14 lists each constant and the corresponding value and exception condition.

Table 11-14. Signals and Exception Conditions

Value	Signal	Exception Condition
1	`SIGABRT`	Abnormal termination of the program, as is signaled by the `abort` function. Also, the `SIGABRT` signal is sent for many operating system errors.
2	`SIGFPE`	Floating-point exception (computational condition): the VOS zero-divide, floating-point underflow, and floating-point overflow errors.
3	`SIGILL`	Illegal instruction.
4	`SIGINT`	Interrupt (break condition), which is signaled by pressing the terminal's CTRL and BREAK keys simultaneously.
5	`SIGSEGV`	Segmentation violation (illegal memory access).
6	`SIGTERM`	Program termination, which is signaled only by the program through the function invocation `kill(0, SIGTERM)`.
7	`SIGUSR1`	User-defined signal 1.
8	`SIGUSR2`	User-defined signal 2.

The `signal` function allows you to specify how a particular signal will be handled. With the `signal` function, you can specify one of three ways to handle the signal. You can specify that a particular signal will be handled by a signal-handling function defined within your program. Or you can specify that the signal will be ignored or that it will be handled by the default handler for that signal.

The `signal.h` header file defines three constants that you use with the `signal` function.

- `SIG_IGN` specifies that a given signal will be ignored.
- `SIG_DFL` specifies that a given signal will be handled in the default manner.
- `SIG_ERR` is returned by the `signal` function when it is unable to implement the handling method specified for a given signal.

The signal is raised (that is, sent to the program) when the exception condition occurs or the `kill` function is called with the specified signal as an argument. System hardware or system software or the program can raise some signals. The signals `SIGTERM`, `SIGUSR1`, and `SIGUSR2` can be sent only by the program.

If a call to `signal` has established a user-defined function to handle a particular signal, two actions occur when the signal is raised. First, the equivalent of `signal(`*`sig`*`, SIG_DFL);` is executed for that signal. One exception: this first action does not take place when `SIGILL` is raised. Second, the user-defined signal-handler is invoked and passed the error code number associated with the signal. Some signals (for example, `SIGUSR1`) do not have an error code number associated with them.

At program startup, the equivalent of `signal(`*`sig`*`, SIG_DFL);` is executed for every signal.

Default Signal Handlers

If you call `signal` and specify `SIG_DFL` for a particular exception condition or if you never call `signal` to establish a user-defined function to handle an exception condition, a *default handler* is invoked when that condition occurs. The default handler varies depending on what signal is raised and how the signal is raised.

When the operating system raises a signal other than `SIGFPE`, the default handler suspends program execution, reports the error condition, and pauses at break level. At break level, the user is prompted to enter one of six requests:

```
BREAK
Request?  (stop, continue, debug, keep, login, re-enter)
```

See the *Introduction to VOS (R001)* manual for more information on break level and the six break-level requests.

When the operating system raises `SIGFPE`, the default handling depends on the type of computation condition that occurs.

- For a zero-divide error, the default handler suspends program execution, reports the error condition, and pauses at break level. In general, the following rules apply.

  – In an integral division by 0, the operating system does not allow a return from the default handler.

  – In a floating-point division, if only the divisor is 0.0, the user can select `continue` to continue program execution with a result of positive or negative `INFINITY`.

  - In a floating-point division, if both operands are 0.0, the operating system does not allow a return from the default handler.

- For a floating-point underflow error, the default handler reports the error condition. The handler then continues program execution with a result of 0.00 for the floating-point operation.

- For a floating-point overflow error, the default handler suspends program execution, reports the error condition, and pauses at break level. If the user selects `continue`, the handler continues program execution with a result of positive or negative `INFINITY` for the floating-point operation.

When a signal, except for `SIGABRT`, is raised by the program's calling the `kill` function, the default handler reports the error condition and terminates the program. When `SIGABRT` is raised by the program's calling either `abort` or `kill(0, SIGABRT)`, the default handler terminates the program.

Signal-Handling Functions

In the `func` argument to the `signal` function, you can specify a pointer to a user-defined function that handles a particular signal. When the signal is raised, that signal-handling function is invoked. If the operating system raises the signal, the signal-handling function is passed the error code number (oncode) associated with the condition. If the program raises the signal by calling `kill(0, `*`sig`*`);`, the signal-handling function is passed 0 as the error code number.

> **Note:** Passing the error code number to the signal-handling function is unique to VOS C. If portability is important, do not use the passed error code number in the called signal-handling function.

When a signal other than `SIGILL` is raised and a signal-handling function defined within the program is invoked, the handler for that signal is "reset" to `SIG_DFL`. Therefore, the default handler will be invoked the next time that signal is raised **unless** the signal-handling function contains code to call the `signal` function and re-establish a user-defined function as the handler for that signal. The `SIGILL` function is unique in that it does not automatically reset its handler to `SIG_DFL` when the `SIGILL` signal is raised.

A signal-handling function defined within the program can terminate by executing a `return` statement or by calling the `abort`, `exit`, or `longjmp` function. If the signal-handling function performs none of these actions, the program resumes execution at the point where it was interrupted when the signal-handling function ends.

When the operating system raises `SIGFPE` and invokes a user-defined handler, the type of computation condition determines whether program control can return from the handler and what value the arithmetic operation yields. The rules that apply to default handlers for `SIGFPE` also apply to user-defined handlers. See the previous section, "Default Signal Handlers," for more information on default handlers and `SIGFPE`.

When `SIGILL` or `SIGSEGV` is raised and the user-defined handler returns, a default handler is invoked. This default handler reports the error condition and terminates the program.

Because the library functions are not guaranteed to be re-entrant and can modify data with static storage duration, they cannot be used reliably in a signal-handling function that returns. You can call `abort`, `exit`, `longjmp`, or `signal` from within any signal-handler.

Variable Arguments

The `stdarg.h` header file defines a type definition and three macros that you use for accessing the arguments in the varying part of a parameter list in a function call. Table 11-15 provides a summary of the macros contained in the `stdarg.h` header file.

Table 11-15. Variable Arguments Macros

Function Prototype	**Description**
`void va_arg(va_list arg_ptr, *type*);`	Accesses the next argument from the varying part of a parameter list.
`void va_end(va_list arg_ptr);`	Modifies the argument pointer, `arg_ptr`, so that it is no longer useable.
`void va_start(va_list arg_ptr, *type* parm_n);`	Initializes the argument pointer, `arg_ptr`, so that it can be used by `va_arg` and `va_end`.

The prototypes shown in Table 11-15, specifying the number and types of the arguments expected by the macros, are for informational purposes only. In fact, the `stdarg.h` header file does not include any prototypes. Therefore, the arguments passed to the macros are promoted according to the default argument promotion rules. This argument promotion must be taken into account when specifying the *`type`* argument for the `va_arg` macro and the `parm_n` argument for the `va_start` macro.

In addition to the preceding macros, the `stdarg.h` header file defines one type, `va_list`, which you use to declare the pointer `arg_ptr`.

The varying part of a parameter list is denoted by an ellipsis (`. . .`) and can contain a varying number of arguments or arguments of varying types. To access arguments in the varying part of the list, the parameter list for the called function must contain at least one nonvarying parameter. The rightmost nonvarying parameter, designated as `parm_n` in the `va_start` prototype, immediately precedes the `. . .` in the parameter list and plays a special role in the access mechanism.

The `va_start`, `va_arg`, and `va_end` macros are used together to access the arguments from the varying part of the parameter list. Consider the function call and function definition of the `get_min` function in the following example.

```
#include <stdarg.h>
#include <string.h>

void get_min(char *parm_1, int parm_n, ...);
```

(Continued on next page)

```
main()
{
   int number = 1000;
   int arg_3 = 33, arg_4 = 11, arg_5 = 22, arg_6 = -1;
   char name[25];

   strcpy(name, "John Smith");
   get_min(name, number, arg_3, arg_4, arg_5, arg_6);
}

void get_min(char *parm_1, int parm_n, ...)
{
   int min = 999, a = 0;
   va_list arg_ptr;

   va_start(arg_ptr, parm_n);

   while ( (a = va_arg(arg_ptr, int)) != -1 )
      if (a < min)
         min = a;

   va_end(arg_ptr);
      .
      .
      .
}
```

In the preceding example, the arguments in the **nonvarying** part of the parameter list, such as `parm_1` and `parm_n`, can be accessed by name in the usual manner. Within the parameter list, all nonvarying arguments must precede all varying arguments.

The procedure for accessing the arguments in the **varying** part of the parameter list, such as `arg_3` and `arg_4`, is as follows:

1. Call the `va_start` macro to initialize `arg_ptr` to point to the first argument in the varying part of the list. The `va_start` macro takes two arguments. The first argument, `arg_ptr`, is a pointer of the type `va_list`. The second argument is the name of the last nonvarying argument in the parameter list, designated as `parm_n` in this discussion.

2. Call the `va_arg` macro to access, in sequence, the arguments in the varying part of the parameter list. The first call to `va_arg` yields a result of the type and value of the first varying argument after `parm_n`. In the preceding example, the first argument in the varying part of the list is `arg_3`. Successive calls to `va_arg` return the values of the remaining arguments in the varying part of the parameter list. For each call, `va_arg` takes two arguments. The first argument is `arg_ptr`. The second argument is a data type, which determines by how many bytes the value of `arg_ptr` is incremented each time `va_arg` is called. The data type corresponds to the type of the argument you are accessing.

3. Call `va_end` to make `arg_ptr` unuseable and finalize the access process.

The arguments in the varying part of the parameter cannot be accessed again without calling `va_start` and repeating the preceding procedure.

The argument pointer (`arg_ptr`) can be passed as an argument to another function. If that function calls the `va_arg` macro with the passed argument pointer, the value of the argument pointer in the calling function is indeterminate. Upon returning to the calling function, you must call `va_end` before any further reference to `arg_ptr`.

Common Definitions

The `stddef.h` header file declares two type definitions, three macros, and two variables that you use with many of the library functions.

Table 11-16 lists the one function-like macro contained in the `stddef.h` header file.

Table 11-16. Standard Definitions Macro

Function Prototype	**Description**
`size_t offsetof(`*`struct_type`*`,` *`member_name`*`);`	Gets the offset, in bytes, of a specified member within a structure.

In addition to the preceding function-like macro, the `stddef.h` header file also declares the `NULL` and `OS_NULL_PTR` defined constants.

The null pointer constant, `NULL`, is commonly used in many C implementations. `NULL` is an integer constant expression equal to the value 0, or such an expression cast to the type `void *`.

The `OS_NULL_PTR` macro is used in other VOS languages. `OS_NULL_PTR` is an integer constant expression equal to the value 1 cast to the type `void *`. With most VOS software other than VOS C, the null pointer is almost always `OS_NULL_PTR`, not `NULL`. For example, the VOS operating system subroutines return the `OS_NULL_PTR` constant, **not** the `NULL` pointer constant.

> **Note:** Be aware that `NULL` is **not equal to** `OS_NULL_PTR`. Also, functions that specify `OS_NULL_PTR` are not portable. You should, therefore, use the `NULL` pointer constant except in those situations where the operating system requires `OS_NULL_PTR`.

For more information on `NULL` and `OS_NULL_PTR`, see the "Null Pointer Constants" section in Chapter 4.

The `stddef.h` header file also declares two type definitions.

- `ptrdiff_t` is an unsigned integral type capable of storing the result of subtracting two pointers.

- `size_t` is an unsigned integral type capable of storing the result of the `sizeof` operation.

Lastly, the `stddef.h` header file also declares two `extern` variables: `errno` and `os_errno`. When an error occurs during a library function call, some library functions set one or both of these variables to a positive error code. The compiler initializes both `errno` and `os_errno` to the value 0. However, no library function sets either variable to 0. Thus, a program that uses `errno` or `os_errno` for error checking must assign it the value 0 before a function call, and then inspect the variable before any subsequent function call. If the use of `errno` is not documented in the description of a library function, the value of `errno` or `os_errno` may be set to a nonzero value by a call to that function even if there is no error.

Standard Input and Output

The `stdio.h` header file declares two types, several macros, and many functions that you use for performing input and output. See Chapter 10 for an overview of how you perform input and output with the standard I/O functions. Table 11-17 lists these functions and macros by the category of task that each performs.

Table 11-17. Standard I/O Functions and Macros by Category

Category	Functions
File access functions	`fclose, fdopen, fflush, fileno, fopen, freopen, setbuf, setvbuf`
Formatted I/O functions	`fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf`
Single-character I/O functions	`fgetc, fputc, getc, getchar, putc, putchar, ungetc`
String I/O functions	`fgets, fputs, gets, puts`
Binary I/O functions	`fread, fwrite, getw, putw`
File-positioning functions	`fgetpos, fseek, fsetpos, ftell, rewind`
Error-handling functions	`clearerr, feof, ferror, perror`
File and port management functions	`remove, rename, s$c_get_portid, s$c_get_portid_from_fildes, tmpfile, tmpnam`

Table 11-18 provides a summary of the functions and macros contained in the `stdio.h` header file.

Table 11-18. Standard I/O Functions and Macros *(Page 1 of 5)*

Function Prototype	Description
`void clearerr(FILE *file_ptr);`	Clears the end-of-file indicator and the error indicator for a file associated with a specified file pointer.

Table 11-18. Standard I/O Functions and Macros *(Page 2 of 5)*

Function Prototype	Description
`int fclose(FILE *file_ptr);`	Closes a file associated with a specified file pointer.
`FILE *fdopen(int file_des, char *type);`	Gets the file pointer for the file associated with a specified file descriptor.
`int feof(FILE *file_ptr);`	Determines whether the end-of-file indicator has been set for a file associated with a specified file pointer.
`int ferror(FILE *file_ptr);`	Determines whether the error indicator has been set for a file associated with a specified file pointer.
`int fflush(FILE *file_ptr);`	Flushes the output buffer for a file associated with a specified file pointer.
`int fgetc(FILE *file_ptr);`	Reads and returns the next character from a file associated with a specified file pointer.
`int fgetpos(FILE *file_ptr, fpos_t *pos);`	Gets and stores the current value of the file-position indicator for a file associated with a specified file pointer.
`char *fgets(char *s, int n, FILE *file_ptr);`	Reads a line or a specified number of characters from a file associated with a specified file pointer.
`int fileno(FILE *file_ptr);`	Gets the file descriptor for the file associated with a specified file pointer.
`FILE *fopen(const char *path_name, const char *mode);`	Opens a file and associates a stream with the file.
`int fprintf(FILE *file_ptr, const char *format, ...);`	Writes a formatted string of characters to a file associated with a specified file pointer.
`int fputc(int c, FILE *file_ptr);`	Writes one character to a file associated with a specified file pointer.
`int fputs(const char *s, FILE *file_ptr);`	Writes a string to a file associated with a specified file pointer.
`size_t fread (void *ptr, size_t size, size_t number, FILE *file_ptr);`	Reads a specified number of data items, each of an indicated size, from a file associated with a given file pointer.
`FILE *freopen (const char *path_name, const char *mode, FILE *file_ptr);`	Closes the file associated with a specified file pointer, opens another file, and associates the file pointer and stream with that file.

Table 11-18. Standard I/O Functions and Macros *(Page 3 of 5)*

Function Prototype	Description
`int fscanf(FILE *file_ptr, const char *format, ...);`	Using a specified format, reads a sequence of characters from the file associated with a given file pointer.
`int fseek(FILE *file_ptr, long int offset, int whence);`	Sets the file-position indicator for a file associated with a specified file pointer.
`int fsetpos(FILE *file_ptr, const fpos_t *pos);`	Sets the file-position indicator, for a file associated with a specified file pointer, to an earlier position obtained by the `fgetpos` function.
`long ftell(FILE *file_ptr);`	Gets the current value of the file-position indicator for a file associated with a specified file pointer.
`size_t fwrite (const void *ptr, size_t size, size_t number, FILE *file_ptr);`	Writes a specified number of data items, each of an indicated size, into a file associated with a given file pointer.
`int getc(FILE *file_ptr);`	Reads and returns the next character from a file associated with a specified file pointer.
`int getchar(void);`	Reads and returns the next character from the standard input file, `stdin`.
`char *gets(char *s);`	Reads a line from the standard input file, `stdin`.
`int getw(FILE *file_ptr);`	Reads and returns the next four bytes of data from a file associated with a specified file pointer.
`void perror(const char *string);`	Constructs an error message and writes that message to the standard error file, `stderr`.
`int printf(const char *format, ...);`	Writes a formatted string of characters to the standard output file, `stdout`.
`int putc(int c, FILE *file_ptr);`	Writes one character to a file associated with a specified file pointer.
`int putchar(int c);`	Writes one character to the standard output file, `stdout`.
`int puts(const char *s);`	Writes a string to the standard output file, `stdout`.
`int putw(int w, FILE *file_ptr);`	Writes four bytes of data to a file associated with a specified file pointer.
`int remove(const char *filename);`	Deletes a specified file.

Table 11-18. Standard I/O Functions and Macros *(Page 4 of 5)*

Function Prototype	**Description**
`int rename (const char *old_name, const char *new_name);`	Changes the name of a specified file.
`void rewind(FILE *file_ptr);`	Sets the file-position indicator, for a file associated with a specified file pointer, to the beginning of the file.
`int s$c_get_portid(FILE *file_ptr);`	Returns the port ID associated with a file pointer.
`int s$c_get_portid_from_fildes(int file_des);`	Returns the port ID associated with a file descriptor.
`int scanf(const char *format, ...);`	Using a specified format, reads a sequence of characters from the standard input file, `stdin`.
`void setbuf(FILE *file_ptr, char *buffer);`	Causes a specified buffer, instead of a buffer allocated by the system, to be used for buffered I/O on a file associated with a given file pointer.
`int setvbuf (FILE *file_ptr, char *buffer, int mode, size_t size);`	For a file associated with a specified file pointer, changes one or more aspects of buffering: the buffer to be used in place of the system-allocated buffer, the mode of buffering, or the size of the buffer.
`int sprintf(char *s, const char *format, ...);`	Writes a formatted string of characters to an array.
`int sscanf(const char *s, const char *format, ...);`	Using a specified format, reads a sequence of characters from a given string.
`FILE *tmpfile(void);`	Creates and opens a temporary file.
`char *tmpnam(char *string);`	Generates a unique string that is a valid file name.
`int ungetc(int c, FILE *file_ptr);`	Pushes a specified character back onto the input stream associated with a file.
`int vfprintf (FILE *file_ptr, const char *format, va_list arg_ptr);`	Writes a formatted string of characters to a file. Unlike `fprintf`, this function accepts a pointer to a list of arguments, not the arguments themselves.
`int vprintf(const char *format, va_list arg_ptr);`	Writes a formatted string of characters to the standard output file, `stdout`. Unlike `printf`, this function accepts a pointer to a list of arguments, not the arguments themselves.

Table 11-18. Standard I/O Functions and Macros *(Page 5 of 5)*

Function Prototype	Description
`int vsprintf(char *s, const char *format, va_list arg_ptr);`	Writes a formatted string of characters into an array. Unlike `sprintf`, this function accepts a pointer to a list of arguments, not the arguments themselves.

The following functions from the `stdio.h` header are **not** part of the ANSI C Standard's definition of the library functions:

```
fdopen                    putw
fileno                    s$c_get_portid
getw                      s$c_get_portid_from_fildes
```

Using these functions can affect your program's portability. In particular, because of differences in byte ordering and integer size, the data read by the `getw` function or written by the `putw` function may not be the same on an operating system other than VOS

If an output function is writing data to the terminal's screen and the user presses the CANCEL key, the function stops displaying data and signals the `warning` condition. You can use the `s$enable_condition` subroutine to set up a function to handle the `warning` condition. Then within that handler, you can use the `s$control` subroutine with `RESET_OUTPUT_OPCODE` to restore terminal output. See the *VOS C Subroutines Manual (R068)* for information on the `s$enable_condition` subroutine, and the *VOS Communications Software: Asynchronous Communications (R025)* manual for information on `s$control` and restoring terminal output.

In addition to the functions and macros shown in Table 11-17, the `stdio.h` header file also declares two type definitions.

- The `FILE` type defines an object capable of recording all of the information needed to control a file, such as its file-position indicator, a pointer to its associated buffer, and indicators to record whether a read or write error has occurred and whether end-of-file has been reached.

- The `fpos_t` type defines an object type capable of recording all information needed to specify every position within a file. The `fpos_t` type is used with the `fsetpos` and `fgetpos` functions.

Table 11-19 lists some of the defined constants (macros) that are declared in the `stdio.h` header.

Table 11-19. Miscellaneous Standard I/O Defined Constants

Defined Constant	Value	Description
`EOF`	-1	A negative integral constant expression returned by several functions to indicate end-of-file
`L_tmpnam`	33	An integral constant expression that is the size in bytes of an array large enough to hold a string designating a temporary file name generated by the `tmpnam` function
`TMP_MAX`	2,147,483,647	An integral constant expression that is the number of unique file names that can be generated by the `tmpnam` function

The three standard streams, each an expression of the type pointer to `FILE`, are also defined in the `stdio.h` header file.

- `stdin` for reading input from the terminal's keyboard
- `stdout` for writing output to the terminal's screen
- `stderr` for writing diagnostic output to the terminal's screen.

The `stdio.h` header defines four constants that can be specified with either the `setbuf` or `setvbuf` function.

- `_IOFBF`
- `_IOLBF`
- `_IONBF`
- `BUFSIZ`

See the descriptions of `setbuf` and `setvbuf`, later in this chapter, for information about these constants.

The `stdio.h` header file defines three constants that can be specified with the `fseek` function.

- `SEEK_SET`
- `SEEK_CUR`
- `SEEK_END`

See the description of `fseek`, later in this chapter, for information about these constants.

Finally, in VOS C, the `stdio.h` header declares the `extern` variable `_TRADITIONAL_`. You use this variable to change the behavior of append mode with the `fseek` function after opening a file with `fopen`. See the description of `fopen`, later in this chapter, for information on append mode and the `_TRADITIONAL_` variable.

General Utilities

The `stdlib.h` header file declares several functions that you use for various purposes. Table 11-20 lists these functions by the category of task that each performs.

Table 11-20. General Utility Functions by Category

Category	**Functions**
String-conversion functions	`atof, atoi, atol, strtod, strtol, strtoul`
Pseudo-random sequence generation functions	`rand, srand`
Memory management functions	`calloc, free, malloc, realloc`
Environment functions	`abort, exit, getenv, onexit, system`
Sorting functions	`qsort`

Table 11-21 lists the functions contained in the `stdio.h` header file.

Table 11-21. General Utility Functions *(Page 1 of 2)*

Function Prototype	**Description**
`void abort(void);`	Causes abnormal termination of a program.
`double atof(const char *string);`	Converts a string of ASCII characters into a `double` value.
`int atoi(const char *string);`	Converts a string of ASCII characters into an `int` value.
`long atol(const char *string);`	Converts a string of ASCII characters into a `long int` value.
`void *calloc(size_t num_members, size_t size);`	Allocates and clears memory for a given number of objects, each of a specified size.
`void exit(int status);`	Causes normal termination of a program by flushing all output buffers associated with open files, closing all open files, and invoking all functions registered by the `onexit` function.
`void free(void *ptr);`	Deallocates a specified block of memory previously allocated with `calloc`, `malloc`, or `realloc`.
`char *getenv(const char *name);`	Returns the `NULL` pointer constant because VOS C does not maintain a list of environment variables.

Table 11-21. General Utility Functions *(Page 2 of 2)*

Function Prototype	Description
`void *malloc(size_t size);`	Allocates memory for an object of a specified size.
`onexit_t onexit(onexit_t func_ptr);`	Registers a function to be invoked upon normal termination of the program.
`void qsort ( void *pbase, size_t n_elem, size_t e_width, int (*compare) (const void *element_1, const void *element_2) );`	Sorts the elements of an array.
`int rand(void);`	Generates a pseudo-random integer in the range 0 through `RAND_MAX`.
`void *realloc(void *ptr, size_t size);`	Changes the size of a block of memory previously allocated with `calloc`, `malloc`, or `realloc`.
`void srand(unsigned int seed);`	Initializes the sequence of pseudo-random numbers that are returned by the `rand` function.
`double strtod (const char *string, char **end_ptr);`	Converts a string of ASCII characters into a `double` value.
`long strtol (const char *string, char **end_ptr, int base);`	Converts a string of ASCII characters into a `long int` value.
`unsigned long strtoul (const char *string, char **end_ptr, int base);`	Converts a string of ASCII characters into an `unsigned long` value.
`int system(const char *string);`	Starts a subprocess that executes a specified program module.

In addition to the preceding functions, the `stdlib.h` header also declares one defined constant (macro) and one type.

- The `RAND_MAX` constant is an integral constant expression that is the maximum value returned by the `rand` function.
- The `onexit_t` type is used to declare the argument of the `onexit` function.

String Manipulation

The `string.h` header file declares several functions that you use for manipulating strings or for manipulating characters (bytes) in memory. Table 11-22 lists these functions by the category of task that each performs.

Table 11-22. String-Manipulation Functions by Category

Category	Functions
Copying functions	`memccpy, memcpy, memmove, memset, strcpy, strncpy`
Concatenation functions	`strcat, strncat`
Comparison functions	`memcmp, strcmp, strncmp`
Search functions	`memchr, strchr, strcspn, strpbrk, strrchr, strstr, strspn, strtok`
Length determination functions	`strlen`

Table 11-23 provides a summary of the functions contained in the `string.h` header file.

Table 11-23. String-Manipulation Functions *(Page 1 of 2)*

Function Prototype	Description
`void *memccpy (void *mem1, const void *mem2, int c, size_t n);`	Copies bytes from one memory area into another memory area until a specified byte is encountered in the source memory area or until a specified number of bytes are copied, whichever occurs first.
`void *memchr (const void *mem, int c, size_t n);`	Searches a memory area for the first occurrence of a specified byte.
`int memcmp (const void *mem1, const void *mem2, size_t n);`	Compares a specified number of bytes in one memory area with the same number of bytes in another memory area.
`void *memcpy (void *mem1, const void *mem2, size_t n);`	Copies a specified number of bytes from one memory area into another memory area.
`void *memmove (void *mem1, const void *mem2, size_t n);`	Copies a specified number of bytes from one memory area into another memory area. With `memmove`, some parts of the memory areas can overlap.
`void *memset(void *mem, int c, size_t n);`	Copies the value of a given byte into a specified number of bytes in a memory area.
`char *strcat(...);`	Appends one string to another string.
`char *strchr(...);`	Searches for the first occurrence of a specified character within a given string.
`int strcmp(...);`	Compares one string with another string.
`char *strcpy(...);`	Copies one string into another string.

Table 11-23. String-Manipulation Functions *(Page 2 of 2)*

Function Prototype	Description
`size_t strcspn(...);`	Computes the length of the initial segment of a specified string that consists entirely of characters **not** from another string.
`size_t strlen(...);`	Computes the length of a specified string.
`char *strncat(...);`	Appends a specified number of characters from one string to another string.
`int strncmp(...);`	Compares a specified number of characters from one string with characters from another string.
`char *strncpy(...);`	Copies a specified number of characters from one string into another string.
`char *strpbrk(...);`	Searches for the first occurrence, within one string, of any of the characters from another string.
`char *strrchr(...);`	Searches for the last occurrence of a specified character within a given string.
`size_t strspn(...);`	Computes the length of the initial segment of a specified string that consists entirely of characters from another string.
`char *strstr (const char *string1, const char *string2);`	Searches for the first occurrence of a specified sequence of characters within a given string.
`char *strtok (char *string1, const char *string2);`	Breaks a string into a sequence of tokens, each of which is delimited by a character from another string.

Using Generic, String-Manipulation Functions

Except for `strstr` and `strtok`, each of the functions shown in Table 11-23 that begins with `str` is a VOS C *generic, string-manipulation function.* Each function's behavior and return value depend on the type of the arguments passed to it. Each generic, string-manipulation function performs the specified operation whether its arguments are pointers to `char`, pointers to `char_varying` strings, or a combination of the two pointer types. With a generic, string-manipulation function, the VOS C compiler takes a special action if an argument contains the address of a `char_varying` string.

For example, if you call the `strcpy` function with two pointers to `char_varying` strings as arguments, the compiler translates the name of the function (`strcpy`) into the name of the actual function that is invoked: `strcpy_vstr_vstr`. For the generic, string-manipulation functions that take one string argument, the following table shows the form of the actual function names.

Actual Function Name	**Type of string**
`str`*`xxx`*	`char *`
`str`*`xxx`*`_vstr`	`char_varying(`*`n`*`) *`

For generic, string-manipulation functions that take two string arguments, the following table shows the form of the actual function names.

Actual Function Name	**Argument Type of string1**	**Argument Type of string2**
`str`*`xxx`*	`char *`	`char *`
`str`*`xxx`*`_nstr_vstr`	`char *`	`char_varying(`*`n`*`) *`
`str`*`xxx`*`_vstr_nstr`	`char_varying(`*`n`*`) *`	`char *`
`str`*`xxx`*`_vstr_vstr`	`char_varying(`*`n`*`) *`	`char_varying(`*`m`*`) *`

When you invoke a generic, string-manipulation function, you can specify the generic name or the actual function name in your program. For example, with `strchr`, if `string` is a pointer to a `char_varying` string, you can invoke the function using the generic function name, `strchr`, or using the actual function name, `strchr_vstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype. See the "Advantages of Function Prototypes" section in Chapter 6 for information on the checks that occur when a function prototype is present.

Using `$substr` Expressions as Arguments

Whether the function call specifies a generic function name or an actual function name, you can use a `$substr` expression as an one or both of the string arguments with some of the string-manipulation functions. The compiler issues a diagnostic in cases where using a `$substr` expression as an argument yields an undesirable result. The following table lists the argument position where a `$substr` expression is allowed as an argument to the string-manipulation functions.

Function Name	**Argument Position**
`strcat`, `strncat`	Second
`strchr`, `strrchr`	Neither
`strcmp`, `strncmp`	First and second
`strcpy`, `strncpy`	Second
`strlen`	First
`strpbrk`	First
`strspn`, `strcspn`	First and second
`strstr`	First and second

When you invoke a **generic**, string-manipulation function, you can use a `$substr` expression (as opposed to the address of a `$substr` expression) as an argument in contexts where a string argument is an **input-only** argument. Normally, the `$substr` expression itself cannot be used as a function argument.

In cases where a `$substr` expression is used as an input-only argument, the `$substr` expression is converted to an actual `char_varying` object. The `$substr` expression itself is converted to the address of a specified character within the character array part of the `char_varying` object that is passed. The second argument to `$substr` indicates the character whose address is taken.

> **Note:** The address of a `$substr` expression is **not** the address of a `char_varying` string, but rather the address of a `char`.

Because they are used as input-only arguments to a generic, string manipulation function, the following two `$substr` expressions are equivalent.

```
char_varying(15) vstr = "abcdefghij";
char array[10] = "KLMNOPQRS";
short l;
    .
    .
    .
strcpy( &vstr, $substr(array, 3, l) );

     strcpy( &vstr, & (char_varying(n)) $substr(array, 3, l) );
```

In the preceding example, the expression `$substr(array, 3, l)` yields the address of a `char` within the character array part of a `char_varying` object. The length (*n*) of this `char_varying` object is a constant equal to the largest value that the compiler expects `l` to contain.

See the "`$substr`" section in Chapter 12 for detailed information on the `$substr` built-in.

Date and Time

The `time.h` header file declares several type definitions, functions, and one macro that you use for manipulating the date and time. The `clock` function gets the amount of CPU time used by the calling program. The remainder of the functions in this header file get or manipulate one of the following date-time values.

- The *calendar time* is the current date and time. In VOS C, calendar time is the number of seconds since January 1, 1980 Greenwich Mean Time (GMT). You use the `time` function to get the calendar time.

- The *broken-down time* is the calendar time converted into different components, such as seconds, minutes, and hours. When the conversion is complete, broken-down time is stored in a `static` (permanent) structure of the type `struct tm`. You use either the `gmtime` or `localtime` function to convert calendar time into broken-down time.

The `gmtime` function converts the calendar time into broken-down time, expressed as Greenwich Mean Time. In contrast, the `localtime` function converts the calendar time into broken-down time, expressed as local time. In VOS C, *local time* is the calendar time expressed using the current time zone defined for the calling process.

Table 11-24 provides a summary of the functions contained in the `time.h` header file.

Table 11-24. Date and Time Functions *(Page 1 of 2)*

Function Prototype	**Description**
`char *asctime(const struct tm *time_ptr);`	Converts broken-down time into a date-time string.
`clock_t clock(void);`	Gets the amount of CPU time used by the calling program.
`char *ctime(const time_t *timer);`	Converts calendar time into local time in the form of a date-time string.
`double difftime(time_t time1, time_t time0);`	Computes the difference between two calendar times.
`struct tm *gmtime(const time_t *timer);`	Converts calendar time into broken-down time, expressed as Greenwich Mean Time.
`struct tm *localtime(const time_t *timer);`	Converts calendar time into broken-down time, expressed as local time.

Table 11-24. Date and Time Functions *(Page 2 of 2)*

Function Prototype	Description
`time_t time(time_t *timer);`	Gets the current calendar time: the number of seconds since January 1, 1980 GMT.

In addition to the preceding functions, the `time.h` header file also declares the `CLK_TCK` defined constant (macro), which specifies the number of jiffies in a second. The `CPU` time returned by the `clock` function is expressed in jiffies. For information on the use of the `CLK_TCK` constant, see the description of the `clock` function later in this chapter.

The `time.h` header file also declares two type definitions.

- `clock_t`
- `time_t`

Both of these are arithmetic types that are used to represent either the return value of or an argument to one of the date-time functions.

Lastly, the `time.h` header file defines the tag for the structure type `struct tm` as specifying the following structure:

```
struct tm
     {
     long  tm_sec;       /*  seconds after the minute - [0, 59]  */
     long  tm_min;       /*  minutes after the hour - [0, 59]    */
     long  tm_hour;      /*  hours since midnight - [0, 23]      */
     long  tm_mday;      /*  day of the month - [1, 31]          */
     long  tm_mon;       /*  month of the year - [0, 11]         */
     long  tm_year;      /*  years since 1900                    */
     long  tm_wday;      /*  days since Sunday - [0, 6]          */
     long  tm_yday;      /*  day of the year - [0, 365]          */
     long  tm_isdst;     /*  Daylight Savings Time flag          */
     };
```

The `struct tm` structure type holds the components of a calendar time broken into different components, such as seconds, minutes, and hours.

The procedure for using the functions in `time.h` to obtain a date-time string is as follows:

1. Call the `time` function to get the calendar time.
2. Call either `gmtime` or `localtime` to convert the calendar time into, respectively, Greenwich Mean Time or local time.
3. Call `asctime` to convert a broken-down time into a date-time string having, for example, the following format:

   ```
   Mon Sep 30 00:00:00 1999\n\0
   ```

As an alternative to steps 2 and 3, you can call `ctime`, which, in turn, calls both `localtime` and `asctime` to create the date-time string.

VOS C Library Function Reference

This section contains a reference entry for each of the VOS C library functions. The entries are arranged in alphabetical order.

Format for Library Functions. Each function reference entry is presented using the following format.

`function_name`

The name of the function is at the top of the first page of the function reference.

Purpose

Explains briefly what the function does.

Syntax

Lists the header file(s) needed by the function and shows the prototype for the function.

Explanation

Provides information about how to use the function, such as descriptions of the function's arguments.

Return Value

Describes the function's return value, if any.

Examples

Contains one or more examples illustrating how to use the function.

Related Functions

Lists other functions similar to or useful with this function.

`abort`

Purpose

Causes abnormal termination of a program.

Syntax

```
#include <stdlib.h>

void abort(void);
```

Explanation

The `abort` function causes abnormal termination of a program unless the `SIGABRT` signal is being ignored or handled by a signal-handling function defined in the program. The `abort` function sends the `SIGABRT` signal to the calling program.

You can use the `signal` function to specify how the `SIGABRT` signal is handled. If the calling program has not used `signal` to specify how `SIGABRT` is handled, default handling for `SIGABRT` causes abnormal program termination. The default operating system handling for `SIGABRT` closes any open files after flushing any output buffers associated with them. In contrast to program termination with the `exit` function, the default handling for `SIGABRT` does not include calling the functions registered with `onexit`.

See the `signal` function for information on ignoring signals and on setting up signal-handling functions.

Return Value

If the program is ignoring the `SIGABRT` signal, `abort` returns no value. Otherwise, `abort` does not return to its caller.

Examples

The following program fragment uses `abort` to cause abnormal program termination when an error is detected.

```
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

main()
{
   int error_flag = FALSE;
       .
       .
       .
   if (error_flag == TRUE)
      abort();
       .
       .
       .
}
```

Related Functions

`exit`, `kill`, `signal`

abs

Purpose

Computes the absolute value of a specified integer number.

Syntax

```
#include <math.h>

int abs(int x);
```

Explanation

The `abs` function calculates the absolute value of `x`. The `x` argument is a value of the type `int`.

No domain or range errors are possible with `abs`.

Return Value

The `abs` function returns the absolute value of `x`.

If the value of `x` is -2,147,483,648, `abs` returns the same (negative) value.

Examples

The following program uses `abs` to calculate the absolute value of an integer number entered at the terminal's keyboard.

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

main()
{
   int x, abs_value;

   printf("Enter the number for which you want to find the absolute
value.\n");
   scanf("%d", &x);

   abs_value = abs(x);

   printf("The absolute value of %d = %d\n", x, abs_value);
}
```

If `-10` were entered at the terminal's keyboard, the output would be as follows:

```
The absolute value of -10 = 10
```

Related Functions

`ceil`, `fabs`, `floor`

access

Purpose

Checks to determine whether a file or directory exists and whether the caller has a specified type of access.

Syntax

```
#include <c_utilities.h>

int access(char *path_name, int mode);
```

Explanation

The `access` function determines whether the file or directory specified by the string pointed to by `path_name` exists and whether the caller has the type of access given in `mode`. The string pointed to by `path_name` is a full or relative path name. The `mode` argument is an `int` value that specifies one of the types of access shown in the following table.

Value of mode	**Meaning**
00	Checks whether the file or directory exists.
01	Checks whether the caller has execute access if `path_name` is an executable program module.
02	Checks whether the caller has write access if `path_name` locates a file, or modify access if `path_name` locates a directory.
04	Checks whether the caller has read access if `path_name` locates a file, or status access if `path_name` locates a directory.

If the file or directory does not exist, the `access` function sets `errno` to `e$object_not_found` (`1032`). If the caller does not have the type of access specified by `mode`, `access` sets `errno` to `e$usf_EACCES` (`5013`). For other errors, `access` sets `errno` to the error code number returned by the operating system.

For information on types of access and access control, see the *VOS Commands User's Guide (R089).*

Return Value

The `access` function returns the value 0 if the file or directory exists and the caller has the specified type of access.

If the file or directory does not exist or if the caller does not have the specified type of access, the `access` function returns the value -1.

Examples

The following program uses `access` to determine whether a file or directory, whose name is entered on the command line, exists and what type of access the caller has to it.

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

main(int argc, char *argv[])
{
   char *path_name = argv[1];

   if (argc != 2)
      {
      puts("Enter a path name on the command line.");
      exit(1);
      }

   printf("Checking the type of access on: %s\n", path_name);

   errno = 0;
   if ( (access(path_name, 00)) == -1 )
      {
     printf("File or directory does not exist or error occurred:");
      printf("  errno = %d\n", errno);
      }
   else
      if ( (access(path_name, 02)) == 0 )
         {
         puts("Caller has write access if path name is a file,");
        puts("or modify access if path name is a directory.");
         }
      else
         if ( (access(path_name, 04)) == 0 )
            {
            puts("Caller has read access if path name is a file,");
            puts("or status access if path name is a directory.");
            }
         else
            if ( (access(path_name, 01)) == 0 )
               {
               puts("Caller has execute access");
             puts("to a file that is an executable program module.");
               }
            else
              puts("Caller has no access to the file or directory.");
}
```

Related Functions

None

acos

Purpose

Computes the arc cosine of a floating-point value in the range -1 through 1.

Syntax

```
#include <math.h>

double acos(double x);
```

Explanation

The `acos` function calculates the arc cosine of `x`, a `double` value in the range -1 through 1. The returned value is in the range 0 through π radians.

If `x` is not in the range -1 through 1, a domain error occurs, and `acos` sets the external variable `errno` to `EDOM`.

Return Value

The `acos` function returns an arc cosine in radians. If the `x` argument is not in the range -1 through 1, `acos` returns the value 0.

Examples

The following program uses `acos` to calculate the arc cosine of a floating-point value entered at the terminal's keyboard.

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

main()
{
   double x, arc_cosine;

   printf("Enter a number for which you want to find the arc
cosine.\n");
   scanf("%lf", &x);

   errno = 0;
   arc_cosine = acos(x);
```

(Continued on next page)

```
        if (errno != EDOM)
           printf("The arc cosine of %f = %f radians.\n", x, arc_cosine);
        else
           printf("Domain error occurred.\n");
     }
```

If `5.0E-1` were entered at the terminal's keyboard, the output would be as follows:

```
     The arc cosine of 0.500000 = 1.047198 radians.
```

Related Functions

`cos`

alloca

Purpose

Allocates a specified number of bytes of temporary storage.

Syntax

```
#include <alloca.h>

void *alloca(int size);
```

Explanation

The `alloca` function allocates `size` bytes of temporary storage in the stack frame of the calling function. The number of bytes specified in `size` should include any alignment padding that would separate the objects in an array. You can use the `sizeof` operator to ensure that this padding is included in `size`.

The space that `alloca` allocates is temporary. The `alloca` function allows you to allocate storage dynamically as a function executes. For example, you could use `alloca` in situations where the amount of stack space that you need to allocate is not known until the function is called. The amount of space to allocate might, for instance, depend on the number or types of arguments specified in the function call. This space is automatically deallocated by the system when the function containing the `alloca` call returns. In contrast, the space that the `calloc` and `malloc` functions allocate is permanent. The program must use the `free` function to deallocate space that has been allocated by `calloc` or `malloc`.

> **Note:** Do **not** use the `free` function to attempt to deallocate memory that has been allocated by `alloca`.

If `alloca` is successful, it returns a pointer to the first byte of allocated memory. This pointer is a "generic pointer" declared as a "pointer to `void`." A pointer to `void` can be assigned, without a cast, to a pointer to any object type. Nevertheless, many programmers cast the return type of `alloca` so that the type of the returned pointer is explicitly converted to the appropriate data type. See the "Pointers to Void" section in Chapter 4 for more information on that pointer type.

The only way that you can access the contents of the allocated memory is indirectly through the pointer returned by `alloca`. Therefore, assign the returned pointer to a pointer variable of the appropriate type. Then, you can specify the pointer variable itself when the object's address is needed, or use an indirection operation when the contents at that address are needed.

The contents of the memory allocated by `alloca` have unpredictable initial values. If you want the allocated memory to be initialized to binary 0 or any other value, you can use the `memset` function to write the value 0 into each byte of allocated memory.

The starting address of the memory allocated with `alloca` begins on a boundary that is suitably aligned for all Stratus processors.

Notes:

1. Using `alloca` within an argument list in a call to another function yields unpredictable results.
2. Do **not** use the `s$connect_vm_region` subroutine with the memory that `alloca` allocates.

When you attempt to allocate too much memory, the operating system does one of the following:

- If `alloca` tries to allocate memory beyond the stack but within the stack fence, the operating system signals the `error` condition and invokes the default handler for that condition.
- If `alloca` tries to allocate memory beyond both the stack and the stack fence, a fatal error occurs and the program's process may be stopped.

In neither of the preceding cases does `alloca` return to its caller. The amount of stack space that `alloca` can allocate depends on the size of the module's paging partition. The size of the paging partition for each module is site-settable.

Because the `alloca` function is **not** part of the ANSI C Standard's definition of the library functions, using `alloca` can affect your program's portability. The `alloca` function is included in the VOS C library so that existing programs that use `alloca` will be compatible with the VOS implementation of the C language.

Return Value

If `alloca` successfully allocates memory of the specified size, the function returns a pointer to the first byte of allocated memory.

If `alloca` does not successfully allocate memory of the specified size, the function returns the `NULL` pointer constant. The `alloca` function can fail, for example, if `size` is less than 0.

If `size` equals 0, `alloca` returns a valid pointer whose pointed-to storage should not be accessed.

Examples

The following function definition uses `alloca` to allocate stack space for a sequence of characters whose length can vary from one function call to the next. The allocated storage is accessed through the pointer `temp_string`.

```
#include <alloca.h>
#include <string.h>

void string_swap (char * string1, char * string2)
   {
   char *temp_string;

   temp_string = (char *) alloca(strlen(string1) + 1);

   strcpy(temp_string, string1);

   strcpy(string1, string2);

   strcpy(string2, temp_string);
   }
```

Related Functions

`calloc`, `malloc`

asctime

Purpose

Converts broken-down time into a date-time string.

Syntax

```
#include <time.h>

char *asctime(const struct tm *time_ptr);
```

Explanation

The `asctime` function converts broken-down time in the structure pointed to by `time_ptr` into a date-time string. The `time_ptr` argument is a pointer to a structure of the type `struct tm`, which is a type defined in the `time.h` header file. The address stored in `time_ptr` is assigned by an earlier call to the `gmtime` or `localtime` function.

The `asctime` function returns a pointer to a string having, for example, the following format:

```
Mon Sep 30 00:00:00 1999\n\0
```

When the conversion is complete, this string resides in a `static` (permanent) 26-character array declared by the `asctime` function. You can use the pointer returned by `asctime` to access this string. Subsequent calls to `asctime` or `ctime` will overwrite this array.

See the “Date and Time” section earlier in this chapter for information on how to use the functions found in the `time.h` header file.

Return Value

The `asctime` function returns a pointer to the date-time string.

Examples

The following program uses `asctime` to convert broken-down time into a date-time string.

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

time_t timer;
struct tm *time_ptr;
char *time_string;

main()
{
/*  Use the time function to get the calendar time   */
/*  and to assign that value to timer                */

   if (time(&timer) == -1)
      puts("Error occurred:  time function call failed.");

/*  Use the localtime function to convert calendar time into
broken-down   */
/*  time, which will be stored in the structure pointed to by time_ptr
*/

   if ( (time_ptr = localtime(&timer)) == NULL )
      puts("Error occurred:  localtime function call failed.");

/*  Use the asctime function to convert broken-down time in the
structure  */
/*  pointed to by time_ptr into the date-time string
*/

   time_string = asctime(time_ptr);

   printf("The local time is %s", time_string);
}
```

The output from the preceding program might be as follows:

```
The local time is Thu Aug 02 08:29:25 1999
```

Related Functions

`ctime`, `gmtime`, `localtime`, `time`

asin

Purpose

Computes the arc sine of a floating-point value in the range -1 through 1.

Syntax

```
#include <math.h>

double asin(double x);
```

Explanation

The `asin` function calculates the arc sine of `x`, a `double` value in the range -1 through 1. The returned value is in the range -π/2 through π/2 radians.

If `x` is not in the range -1 through 1, a domain error occurs, and `asin` sets the external variable `errno` to `EDOM`.

Return Value

The `asin` function returns an arc sine in radians. If the `x` argument is not in the range -1 through 1, `asin` returns the value 0.

Examples

The following program uses `asin` to calculate the arc sine of a floating-point value entered at the terminal's keyboard.

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

main()
{
   double x, arc_sine;

  printf("Enter a number for which you want to find the arc sine.\n");
   scanf("%lf", &x);

   errno = 0;
   arc_sine = asin(x);
```

(Continued on next page)

```
    if (errno != EDOM)
          printf("The arc sine of %f = %f radians.\n", x, arc_sine);
       else
          printf("Domain error occurred.\n");
    }
```

If `1.00` were entered at the terminal's keyboard, the output would be as follows:

```
The arc sine of 1.000000 = 1.570796 radians.
```

Related Functions

`sin`

assert

Purpose

If a specific expression is false, writes diagnostic information to the standard error file, `stderr`, and aborts the program.

Syntax

```
#include <assert.h>

void assert(int expression);
```

Explanation

The `assert` macro writes diagnostic information to the preopened file pointed to by `stderr` if *`expression`* is false (that is, evaluates to the value 0). The file pointed to by `stderr` is usually associated with the terminal's screen. If *`expression`* is true (that is, evaluates to a nonzero value), `assert` does nothing. The *`expression`* is an expression that resolves to an `int`.

The information written to `stderr` includes the text of the *`expression`* argument, the name of the source module associated with the program module, and the line number where the `assert` macro was invoked. The message is written in the following format:

```
Assertion failed: (expression), in file source_module_name, line
number\n
```

The *`source_module_name`* is the value of the `__FILE__` predefined macro, and *`number`* is the value of the `__LINE__` predefined macro.

After `assert` writes the message, it calls the `abort` function.

The `assert` macro can be disabled by defining the `NDEBUG` macro as a macro name. See the "Diagnostics" section earlier in this chapter for information on `NDEBUG`.

Return Value

The `assert` macro returns no value.

Examples

The following program uses `assert` to write diagnostic information on the terminal's screen if an expression is false.

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

int test_expression;

main()
{
   puts("If you want the assertion to be false, enter 0.");
   puts("If you want the assertion to be true, enter any nonzero
value.");

   scanf("%d", &test_expression);

   assert(test_expression != 0);
}
```

If the value 0 were entered at the terminal's keyboard, the output from the preceding program might be as follows:

```
Assertion failed: (test_expression != 0), in file test_assert.c, line
13
```

Related Functions

`abort`, `exit`

atan

Purpose

Computes the arc tangent of a floating-point value.

Syntax

```
#include <math.h>

double atan(double x);
```

Explanation

The `atan` function calculates the arc tangent of `x`. The `x` argument is a value of the type `double`. The returned value is in the range -π/2 through π/2 radians.

No domain or range errors are possible with the `atan` function.

Return Value

The `atan` function returns an arc tangent in radians.

Examples

The following program uses `atan` to calculate the arc tangent of a floating-point value entered at the terminal's keyboard.

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

main()
{
   double x, arc_tangent;

   printf("Enter a number for which you want to find the arc
tangent.\n");
   scanf("%lf", &x);

   arc_tangent = atan(x);

   printf("The arc tangent of %f = %f radians.\n", x, arc_tangent);
}
```

If `1.000000E+00` were entered at the terminal's keyboard, the output would be as follows:

```
The arc tangent of 1.000000 = 0.785398 radians.
```

Related Functions

`atan2`, `tan`

atan2

Purpose

Computes the arc tangent of the ratio of two nonzero floating-point values.

Syntax

```
#include <math.h>

double atan2(double y, double x);
```

Explanation

The `atan2` function calculates the arc tangent of the ratio `y / x`. Both the `y` and `x` arguments are values of the type `double`. The returned value is in the range -π through π radians. The `atan2` function uses the signs of both arguments to determine the quadrant of the returned value.

If both `y` and `x` are 0, a domain error occurs, and `atan2` sets the external variable `errno` to `EDOM`.

Return Value

The `atan2` function returns the arc tangent, in radians, of the ratio `y / x`. If both `y` and `x` are 0, `atan2` returns 0.

Examples

The following program uses `atan2` to calculate the arc tangent of the ratio of two floating-point values entered at the terminal's keyboard.

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

main()
{
   double y, x, arc_tangent;

   printf("Enter a number for y.\n");
   scanf("%lf", &y);
```

(Continued on next page)

```
      printf("Enter a number for x.\n");
      scanf("%lf", &x);

      errno = 0;
      arc_tangent = atan2(y, x);

      if (errno != EDOM)
         printf("The arc tangent of %f / %f = %f radians.\n", y, x,
   arc_tangent);
      else
         printf("Domain error occurred.\n");
   }
```

If two values, `3.0E+00` for `y` and `4.0E+00` for `x`, were entered at the terminal's keyboard, the output would be as follows:

```
   The arc tangent of 3.000000 / 4.000000 = 0.643501 radians.
```

Related Functions

`atan`, `tan`

atof

Purpose

Converts a string of ASCII characters into a `double` value.

Syntax

```
#include <stdlib.h>

double atof(const char *string);
```

Explanation

The `atof` function converts a string pointed to by `string` into a value of the type `double`.

The string contains the ASCII character representation of a floating-point number. The string can contain leading white-space characters, an optional sign, and a string of digits optionally containing a decimal point. It can end with an optional `e` or `E` followed by an optionally signed integer. The following examples are valid character representations of floating-point numbers that `atof` could convert. In the third example, the symbol   indicates a space character.

```
5E-10

-5.0e+5

 987.123
```

If the function encounters a character that it does not recognize, it ends the conversion. If the return value creates an overflow or underflow out-of-range condition, `atof` sets `errno` to `ERANGE`.

Return Value

The `atof` function returns the converted string as a value of the type `double`.

If the string begins with an unrecognized character, `atof` returns the value 0. On overflow, the function returns `HUGE_VAL`. On underflow, `atof` returns the value 0.

Examples

The following program uses `atof` to convert five strings into floating-point numbers.

```
#include <stdlib.h>

double d;
char *string;

main()
{
   string = "  1.23456e3";
   d = atof(string);             /*  d = 1234.56     */

   string = "1.23456e-3";
   d = atof(string);             /*  d = 0.00123456  */

   string = "1.23456e789";
  d = atof(string);         /*  d = HUGE_VAL  (because of overflow)
*/

   string = "1.23456e-789";
  d = atof(string);         /*  d = 0.0  (because of underflow)
*/

   string = "s1.23456";
   d = atof(string);      /*  d = 0.0  (because of bad beginning
character)  */
}
```

Related Functions

`atoi`, `atol`

atoi

Purpose

Converts a string of ASCII characters into an `int` value.

Syntax

```
#include <stdlib.h>

int atoi(const char *string);
```

Explanation

The `atoi` function converts a string pointed to by `string` into a value of the type `int`.

The string contains the ASCII character representation of an integer. The string can contain leading white-space characters, an optional sign, and a string of digits. The following examples are valid character representations of integers that `atoi` can convert. In the third example, the symbol   indicates a space character.

```
-987654

000001

 67
```

If the function encounters a character other than a digit or a leading sign, it ends the conversion.

Return Value

The `atoi` function returns the converted string as a value of the type `int`.

If the string begins with an unrecognized character, `atoi` returns the value 0. When the converted value creates an overflow or underflow condition, `atoi` returns an unexpected value.

Examples

The following program uses `atoi` to convert five strings into values of the type `int`.

```
#include <stdlib.h>

int i;
char *string;
```

(Continued on next page)

```
main()
{
   string = "  123456";
   i = atoi(string);        /*  i = 123456   */

   string = "  -123456";
   i = atoi(string);        /*  i = -123456  */

   string = "9999999999";
   i = atoi(string);        /*  i = unexpected value (because of
overflow)   */

   string = "-9999999999";
   i = atoi(string);        /*  i = unexpected value (because of
underflow)  */

   string = "e-12";
   i = atoi(string);        /*  i = 0 (because of bad beginning
character)   */
}
```

Related Functions

`atof`, `atol`

atol

Purpose

Converts a string of ASCII characters into a `long` value.

Syntax

```
#include <stdlib.h>

long atol(const char *string);
```

Explanation

The `atol` function converts a string pointed to by `string` into a value of the type `long`. Since values of the type `int` and values of the type `long` both take up four bytes of storage in VOS C, the `atoi` and `atol` functions are identical except for their return types.

The string contains the ASCII character representation of an integer. The string can contain leading white-space characters, an optional sign, and a string of digits. The following examples are valid character representations of integers that `atol` could convert. In the third example, the symbol   indicates a space character.

```
-987654

000001

 67
```

If the function encounters a character other than a digit or a leading sign, it ends the conversion.

Return Value

The `atol` function returns the converted string as a value of the type `long`.

If the string begins with an unrecognized character, `atol` returns the value 0. When the converted value creates an overflow or underflow condition, `atol` returns an unexpected value.

Examples

The following program uses `atol` to convert five strings into values of the type `long`.

```
#include <stdlib.h>

long l;
char *string;

main()
{
   string = "  123456";
   l = atol(string);        /*  l = 123456   */

   string = "  -123456";
   l = atol(string);        /*  l = -123456  */

   string = "9999999999";
   l = atol(string);        /*  l = unexpected value (because of
overflow)   */

   string = "-9999999999";
   l = atol(string);        /*  l = unexpected value (because of
underflow)  */

   string = "e-12";
   l = atol(string);        /*  l = 0 (because of bad beginning
character)   */
}
```

Related Functions

`atof`, `atoi`

calloc

Purpose

Allocates and clears memory for a given number of objects, each of a specified size.

Syntax

```
#include <stdlib.h>

void *calloc(size_t num_members, size_t size);
```

Explanation

The `calloc` function allocates memory for the number of objects indicated by `num_members`, each object of the size specified in `size`. Both the `num_members` and `size` arguments are values of the type `size_t`, which is a type definition declared in the `stddef.h` header file.

The number of bytes specified in `size` should include any alignment padding that would separate the same objects in an array. You can use the `sizeof` operator to ensure that this padding is included in `size`.

In contrast to the `malloc` function, `calloc` initializes all allocated memory to binary 0.

If `calloc` is successful, it returns a pointer to the first byte of allocated memory. This pointer is a "generic pointer" declared as a "pointer to `void`." A pointer to `void` can be assigned, without a cast, to a pointer to any object type. Nevertheless, many programmers cast the return type of `calloc` so that the type of the returned pointer is explicitly converted to the appropriate data type. See the "Pointers to Void" section in Chapter 4 for more information on that pointer type.

The `calloc` function allows you to allocate storage dynamically at run time. The amount of memory a program uses can be determined by run-time events, such as user input or the amount of data in a file.

The only way that you can access the contents of the allocated memory is indirectly through the pointer returned by `calloc`. Therefore, assign the returned pointer to a pointer variable of the appropriate type. Then, you can specify the pointer variable itself when the object's address is needed, or you can use an indirection operation when the contents at that address are needed.

The starting address of the memory allocated with `calloc` begins on a boundary that is suitably aligned for all Stratus processors.

Return Value

If `calloc` successfully allocates the memory specified, the function returns a pointer to the first byte of allocated memory.

If `calloc` does not successfully allocate the memory specified, the function returns the `NULL` pointer constant. The `calloc` function can fail, for example, if the amount of memory required is not free.

If `(num_members * size)` equals 0, `calloc` returns a valid pointer whose pointed-to storage should not be accessed.

Examples

The following program uses `calloc` to allocate memory for 10 structures. In the call to `calloc`, the `sizeof(struct item)` operation specifies how many bytes of memory each structure occupies. After memory is allocated, each structure's members can be accessed through the `ptr` pointer variable. For example, you can access the fifth structure's `amount` member using the expression `(ptr + 4)->amount`.

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

#define BUNCH 10

struct item
   {
   char name[100];
   double amount;
   };

struct item *ptr;

main()
{
   ptr = (struct item *) calloc( BUNCH, sizeof(struct item) );

   if (ptr == NULL)
      {
      puts("Memory was not successfully allocated.");
      exit(1);
      }
   else
      puts("Memory was successfully allocated.");
}
```

Related Functions

`alloca`, `free`, `malloc`, `realloc`

ceil

Purpose

Computes the smallest integer value that is greater than or equal to a specified floating-point number.

Syntax

```
#include <math.h>

double ceil(double x);
```

Explanation

The `ceil` function calculates the smallest integer value that is greater than or equal to `x`. The `x` argument is a value of the type `double`. You can use `ceil` to round a floating-point value up to an integer.

If the value of `x` is too large (greater than approximately 1.84 multiplied by 10^{19}), the `ceil` function sets `errno` to `e$big_arg_causes_error` (`3583`).

Return Value

The `ceil` function returns the ceiling of `x`: the smallest integer value that is greater than or equal to `x`. If `x` is 0, `ceil` returns 0.

If the value of `x` is too large, `ceil` returns `x`.

Examples

The following program uses `ceil` to calculate the ceiling of a floating-point value entered at the terminal's keyboard.

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

short e$big_arg_causes_error;       /*  error code 3583  */

main()
{
   double x, ceiling;

   printf("Enter the number for which you want to find the
ceiling.\n");
   scanf("%lf", &x);
```

(Continued on next page)

```
          errno = 0;
          ceiling = ceil(x);

          if (errno == e$big_arg_causes_error)
             puts("Argument produces an invalid result.");
          else
             printf("The ceiling of %.3lf = %.3lf\n", x, ceiling);
       }
```

If `-123.456` were entered at the terminal's keyboard, the output would be as follows:

```
The ceiling of -123.456 = -123.000
```

Related Functions

`abs`, `fabs`, `floor`

chdir

Purpose

Changes the current directory to the directory specified.

Syntax

```
#include <c_utilities.h>

int chdir(char *path_name);
```

Explanation

The `chdir` function changes the current directory to the directory specified by the string pointed to by `path_name`. The string pointed to by `path_name` is a full or relative path name.

If `chdir` is not successful in changing the current directory, the function sets `errno` to the error code number returned by the operating system.

The directory change remains in effect for the program's process when the program terminates.

Return Value

The `chdir` function returns the value 0 if it is successful in changing the current directory.

If the function cannot change the current directory to the specified directory, `chdir` returns the value -1.

Examples

The following program uses `chdir` to change the current directory to a directory whose path name is entered on the command line.

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

main(int argc, char *argv[])
{
   char *path_name = argv[1];
```

(Continued on next page)

```
        if (argc != 2)
           {
           puts("Enter a path name on the command line.");
           exit(1);
           }

        errno = 0;
        if ( (chdir(path_name)) == 0 )
           {
           printf("The current directory has been changed to:\n");
           printf("%s\n", path_name);
           }
        else
           {
           printf("The current directory could not be changed to:\n");
           printf("  %s\n", path_name);
           printf("errno = %d\n", errno);
           }
     }
```

Related Functions

`system`

clearerr

Purpose

Clears the end-of-file indicator and the error indicator for a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

void clearerr(FILE *file_ptr);
```

Explanation

The `clearerr` function clears the end-of-file indicator and the error indicator for the file pointed to by `file_ptr`. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

Certain C library functions set the end-of-file indicator for a file if a read operation on the file encounters the end of the file. Some functions set the error indicator for a file if an error occurs during an I/O operation on the file.

Typically, you might use the `clearerr` function after you have determined that the end-of-file indicator or error indicator for a file is set. After `clearerr` clears the end-of-file indicator and the error indicator for a file, you can use the `feof` and `ferror` functions to check that these indicators are cleared.

The end-of-file indicator stays set until you call the `clearerr`, `fseek`, `fsetpos`, or `rewind` function and specify the file pointer associated with the file. The end-of-file indicator is not set on an empty file. Closing the file with `fclose` also clears the file's end-of-file indicator.

The error indicator for the file stays set until you call the `clearerr` or `rewind` function and specify the file pointer associated with the file. Closing the file with `fclose` also clears the file's error indicator.

Return Value

The `clearerr` function returns no value.

Examples

The following program creates an error condition by trying to write to a file opened for reading only. Then, the program uses `ferror` to determine whether an error has occurred, and uses `clearerr` to clear the error indicator for the file pointed to by `file_ptr`.

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

main()
{
   char c;
   FILE *file_ptr;

  if ( (file_ptr = fopen("text_file", "r")) == NULL ) /* "r" = reading
only */
      {
      printf("Could not open text_file\n");
      exit(1);
      }

      c = 'a';
     fprintf(file_ptr, "%c", c);  /*  Creates an error:  attempt to
write   */
                           /*  to a file opened for reading only    */
      if (ferror(file_ptr))
         {
         printf("Error occurred: ferror returns nonzero.\n");

         clearerr(file_ptr);

         if ( (ferror(file_ptr)) == 0)
            printf("Error indicator cleared.\n");
         }
      else
         printf("No error occurred: ferror returns %d.\n",
ferror(file_ptr));
}
```

The output from the preceding program is as follows:

```
Error occurred: ferror returns nonzero.
Error indicator cleared.
```

Related Functions

`feof`, `ferror`, `perror`

clock

Purpose

Gets the amount of CPU time used by the calling program.

Syntax

```
#include <time.h>

clock_t clock(void);
```

Explanation

The `clock` function gets the amount of CPU time used by the calling program. The `clock` function takes no arguments. It returns a value of the type `clock_t`, which is a type defined in the `time.h` header file.

In VOS C, the CPU time used is returned in *jiffies*: units of 1/65,536 of a second. To calculate the number of seconds that have elapsed since program startup, divide `clock`'s return value by the value of the constant `CLK_TCK`, which is defined in the `time.h` header file.

> **Note:** If the main entry point of your program is **not** a function named `main`, the VOS C program-startup function `s$start_c_program` is not called, and therefore the `clock` function will not work correctly.

If the main entry point of the program is not a function named `main`, use the following procedure to get the amount of elapsed CPU time.

1. Call the `clock` function once to initialize the system variable that stores elapsed CPU time.
2. Call the `clock` function a second time to get the amount of CPU time that has elapsed since the previous `clock` function call.

See the "Program Startup and the Program Entry Point" section in Chapter 6 for information on program startup.

Return Value

If the operation is successful, the `clock` function returns the amount of CPU time (in units of 1/65,536 of a second) used by the calling program.

If the CPU time used is not available or its value cannot be represented, the `clock` function returns the value -1, cast to the type `clock_t`.

Examples

The following program uses `clock` to get the amount of CPU time used by the program.

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

clock_t jiffies_elapsed = 0;
unsigned long seconds_elapsed = 0;

main()
{
   double total, d_num = 999.999;
   float f_num = (float)111.111;
   int i = 0;

   while (i < 999999)         /*  Loop with some floating-point
arithmetic  */
     {                  /*  so that some seconds are used           */
      total = (d_num / f_num + (double)i);
      i++;
      }

   jiffies_elapsed = clock();

   if (jiffies_elapsed == -1)
      puts("Error calling the clock function.");
   else
      {
      seconds_elapsed = jiffies_elapsed / CLK_TCK;
      printf("The amount of CPU time used = %d seconds\n",
seconds_elapsed);
      }
}
```

Related Functions

`difftime`, `time`

close

Purpose

Closes the file associated with a specified file descriptor.

Syntax

```
#include <c_utilities.h>

int close(int file_des);
```

Explanation

The `close` function closes the file associated with `file_des`. Before the file is closed, any unwritten buffered data for the specified file is written to the file. Any unread buffered data for the file is discarded. The `file_des` argument is an integer file descriptor returned by a previous call to `creat`, `fileno`, or `open`.

When `close` closes the file, the file's output buffer is flushed, and the `FILE` object associated with the file is deleted. If the associated buffer was automatically allocated, it is deallocated. In VOS C, upon normal or abnormal termination of a program, all output buffers are automatically flushed, and all opened files are automatically closed.

If `close` is not successful in closing the file, the function sets the external variables `errno` and `os_errno` to the error code number returned by the operating system. The following table lists some of the error code values that are possible.

Error Code Name	Number
`e$bad_port_number`	`1029`
`e$close_invalid_on_sys_port`	`3425`
`e$invalid_file_pointer`	`3929`
`e$invalid_io_operation`	`1040`
`e$port_not_attached`	`1021`

Return Value

If `close` is successful in closing the file, the function returns the value 0.

If `close` detects an error while attempting to close the file, the function returns the value -1.

Examples

The following program uses `close` to close the file associated with the file descriptor `file_des` after a line has been written to a file.

```
#include <c_utilities.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int file_des, mode, i, ch;
char buffer[100];

main()
{
   if ( (file_des = open("out_file", O_WRONLY, mode)) == -1)
      {
      printf("Error opening out_file.\n");
      exit(1);
      }

   do
      {
      ch = getchar();
      buffer[i++] = (char)ch;
      }  while (ch != '\n');

   write(file_des, buffer, i);

   if ( (close(file_des)) == -1)
      {
      printf("Error closing out_file.\n");
      exit (1);
      }
}
```

Related Functions

`creat`, `fclose`, `fopen`, `open`

cos

Purpose

Computes the cosine of an angle, which is expressed in radians.

Syntax

```
#include <math.h>

double cos(double x);
```

Explanation

The `cos` function calculates the cosine of `x`. The `x` argument is an angle expressed in radians.

If the absolute value of `x` exceeds 32,765, `cos` sets `errno` to the error code number returned by the operating system.

Return Value

The `cos` function returns a cosine. When the absolute value of `x` exceeds 32,765, `cos` returns the value 0.

Examples

The following program uses `cos` to calculate the cosine of an angle, expressed in radians.

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

main()
{
   double x, cosine;

   printf("Enter an angle (in radians) for which ");
   printf("you want to find the cosine.\n ");
   scanf("%le", &x);

   errno = 0;
   cosine = cos(x);

   if (errno != 0)
      printf("Error condition occurred:  errno = %d\n", errno);
   else
      printf("The cosine of %lE radians = %lE.\n", x, cosine);
}
```

If `1.57` were entered at the terminal's keyboard, the output would be as follows:

```
The cosine of 1.570000E+00 radians = 7.963267E-04.
```

Related Functions

`acos`, `sin`, `tan`

cosh

Purpose

Computes the hyperbolic cosine of a floating-point value.

Syntax

```
#include <math.h>

double cosh(double x);
```

Explanation

The `cosh` function calculates the hyperbolic cosine of `x`. The `x` argument is a value of the type `double`.

If the result is outside the range of allowable values, a range error occurs, and `cosh` sets `errno` to `ERANGE`.

Return Value

The `cosh` function returns a hyperbolic cosine. When a range error occurs, `cosh` returns `HUGE_VAL`.

Examples

The following program uses `cosh` to calculate the hyperbolic cosine of a number entered at the terminal's keyboard.

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

main()
{
   double x, h_cosine;

   printf("Enter a number for which you want ");
   printf("to find the hyperbolic cosine.\n");
   scanf("%le", &x);

   errno = 0;
   h_cosine = cosh(x);
```

(Continued on next page)

```
      if (errno == ERANGE)
         puts("A range error occurred.");
      else
         printf("The hyperbolic cosine of %lE = %lE.\n", x, h_cosine);
   }
```

If `1.0` were entered at the terminal's keyboard, the output would be as follows:

```
The hyperbolic cosine of 1.000000E+00 = 1.543081E+00.
```

Related Functions

`sinh`, `tanh`

creat

Purpose

Creates a new file or truncates an existing file, and associates a file descriptor with the file.

Syntax

```
#include <c_utilities.h>

int creat(char *path_name, int mode);
```

Explanation

The `creat` function creates a new file identified by `path_name` and associates a file descriptor with the file. The new file has stream file organization. The `path_name` argument is a pointer to a character string that is a full or relative path name specifying the name of the file to create.

In the VOS environment, `creat` does not use the `mode` argument. Nevertheless, you should include the `mode` argument when you call the function because the function prototype requires the `mode` argument. Since the function does not use `mode`, you can specify any `int` value.

If the file specified by `path_name` already exists, the file is truncated and opened for writing only. Truncating a file deletes the file's contents. If the file specified by `path_name` does not exist, the file is created and opened for writing only. In either case, the file-position indicator is set to the beginning of the file.

The access control list for the file is set to the default access control list specified for the directory that will contain the file. Also, the author of the file is set to the user name of the process that creates the file.

You use the file descriptor returned by `creat` to identify the file with many of the UNIX I/O functions, such as `lseek` and `write`.

If the `creat` operation fails, the function sets the external variables `errno` and `os_errno` to the error code number returned by the operating system. See the `open` function for the possible values that `creat` uses when setting these variables.

Return Value

If the `creat` operation succeeds, the function returns a file descriptor in the form of a non-negative integer.

If the `creat` operation fails, the function returns -1.

Examples

The following program uses `creat` to create a file named `new_file` and to open the file for writing. Then, it writes one line, entered at the terminal's keyboard, to the file.

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

int file_des, mode, i, ch;
long count;
static char buffer[100];

main()
{
   if ( (file_des = creat("%s1#d01>Sales>Jones>new_file", mode)) ==
-1)
      {
      printf("Error creating new_file.\n");
      exit(1);
      }

   for(i = 0; ((ch = getchar()) != '\n'); i++)
      {
      buffer[i] = (char)ch;
      }

   write(file_des, buffer, i);

   if ( (close(file_des)) == -1)
      {
      printf("Error closing file.\n");
      exit (1);
      }
}
```

Related Functions

`close`, `open`, `fdopen`, `fileno`, `fopen`, `freopen`

`ctime`

Purpose

Converts calendar time into local time in the form of a date-time string.

Syntax

```
#include <time.h>

char *ctime(const time_t *timer);
```

Explanation

The `ctime` function converts the calendar time pointed to by `timer` into local time in the form of a date-time string. The `timer` argument is a pointer to a variable of the type `time_t`, which is a type defined in the `time.h` header file.

The `ctime` function returns a pointer to a string having, for example, the following format:

```
Mon Sep 30 00:00:00 1999\n\0
```

When the conversion is complete, this string resides in a `static` (permanent) 26-character array declared by the `asctime` function, which is called by `ctime`. You can use the pointer returned by `ctime` to access this string. Subsequent calls to `asctime` or `ctime` will overwrite this array.

A call to `ctime` is equivalent to the following `asctime` call:

```
asctime(localtime(timer));
```

See the "Date and Time" section earlier in this chapter for information on how to use the functions found in the `time.h` header file.

Return Value

The `ctime` function returns a pointer to the date-time string.

Examples

The following program uses `ctime` to convert calendar time into a date-time string.

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

time_t timer;
char *time_string;

main()
{
/*  Use the time function to get the calendar time               */
/*  and to assign that value to timer                            */

   if (time(&timer) == -1)
      puts("Error occurred:  time function call failed.");

/*  Use the ctime function to convert calendar time in the object  */
/*  pointed to by timer into the date-time string                */

   time_string = ctime(&timer);

   printf("The local time is %s", time_string);
}
```

The output from the preceding program might be as follows:

```
The local time is Thu Aug 02 10:31:08 1999
```

Related Functions

`asctime`, `gmtime`, `localtime`, `time`

difftime

Purpose

Computes the difference between two calendar times.

Syntax

```
#include <time.h>

double difftime(time_t time1, time_t time0);
```

Explanation

The `difftime` function computes the difference between two calendar times: `time1 - time0`. Both `time1` and `time0` are variables of the type `time_t`, which is a type defined in the `time.h` header file.

Both `time1` and `time0` are calendar times that are obtained by calling the `time` function. In VOS C, *calendar time* is the number of seconds from 00:00:00 hour January 1, 1980 Greenwich Mean Time to the current date and time.

Return Value

The `difftime` function returns the difference (in seconds) between two calendar times. This difference is expressed as a value of the type `double`.

Examples

The following program uses `difftime` to calculate the difference between two calendar times.

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

time_t time1, time0;
double difference;

main()
{
   int num;

   time(&time0);

   puts("Enter an integer value.");
   scanf("%d", &num);
```

(Continued on next page)

```
   time(&time1);

   difference = difftime(time1, time0);

   printf("It took you %.0f seconds to enter the value.\n",
difference);
}
```

Related Functions

`clock`, `time`

exit

Purpose

Causes normal termination of a program by flushing all output buffers associated with open files, closing all open files, and invoking all functions registered by the `onexit` function.

Syntax

```
#include <stdlib.h>

void exit(int status);
```

Explanation

The `exit` function causes normal program termination to occur. The function invokes, in the reverse order of their registration, any functions registered with the `onexit` function. After all such functions are called, the function flushes all output buffers associated with open files, closes all open files, and deletes all files created by the `tmpfile` function. Finally, `exit` returns the program's process to command level.

The `status` argument is an `int` value that indicates either successful or abnormal program termination. The value that you pass in the `status` argument can be one of the following:

- The value 0 usually signifies that the program has successfully terminated.
- A nonzero value other than -1 usually signifies that the program has abnormally terminated.
- The value -1 signifies that the program has abnormally terminated and tells the operating system **not** to assign the value of the `status` argument to the `command_status` variable.

The significance of the value -1 is unique to the VOS operating system. If `status` equals -1, the value of the `command_status` variable will be 0 unless the program called `s$error` and passed a nonzero value as the `error_code` argument.

For information on the `command_status` variable, see the *VOS Commands User's Guide (R089).*

For information on the `s$error` subroutine, see the *VOS C Subroutines Manual (R068).*

Return Value

The `exit` function never returns to its caller.

Examples

The following program fragment uses `exit` to terminate the program.

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

int file_des, mode;

main()
{
   if ( (file_des = creat("test_file", mode)) == -1 )
      {
      printf("Error creating file.\n");
      exit(1);
      }
      .
      .
      .
}
```

Related Functions

`abort`, `onexit`

exp

Purpose

Computes the exponential function of a floating-point value.

Syntax

```
#include <math.h>

double exp(double x);
```

Explanation

The `exp` function calculates the exponential function of `x`. The `x` argument is a value of the type `double`. The exponential function of `x` is $e^{|x|}$, where `e` is approximately 2.718, the base of the natural logarithm.

If the result cannot be represented by a `double`, a range error occurs, and `exp` sets `errno` to `ERANGE`.

Return Value

The `exp` function returns the exponential function of `x`.

When the result creates a range error, `exp` returns `HUGE_VAL`. On underflow, `exp` returns the value 0.

Examples

The following program uses `exp` to calculate the exponential function of a floating-point value entered at the terminal's keyboard.

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

main()
{
   double x, exponential;

   printf("Enter a number for which you want to find the
exponential.\n");
   scanf("%le", &x);
```

(Continued on next page)

```
         errno = 0;
         exponential = exp(x);

         if (errno == ERANGE)
            puts("A range error occurred.");
         else
            printf("The exponential of %lE = %lE.\n", x, exponential);
      }
```

If `2.0` were entered at the terminal's keyboard, the output would be as follows:

```
The exponential of 2.000000E+00 = 7.389056E+00.
```

Related Functions

`log`, `log10`, `pow`

`fabs`

Purpose

Computes the absolute value of a specified floating-point number.

Syntax

```
#include <math.h>

double fabs(double x);
```

Explanation

The `fabs` function calculates the absolute value of `x`. The `x` argument is a value of the type `double`.

No domain or range errors are possible with `fabs`.

Return Value

The `fabs` function returns the absolute value of `x`.

Examples

The following program uses `fabs` to calculate the absolute value of a floating-point number entered at the terminal's keyboard.

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

main()
{
   double x, abs_value;

   printf("Enter the number for which you want to find the absolute
value.\n");
   scanf("%lf", &x);

   abs_value = fabs(x);

   printf("The absolute value of %.3lf = %.3lf\n", x, abs_value);
}
```

If `-10.123` were entered at the terminal's keyboard, the output would be as follows:

```
The absolute value of -10.123 = 10.123
```

Related Functions

`abs`, `ceil`, `floor`

fclose

Purpose

Closes a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int fclose(FILE *file_ptr);
```

Explanation

The `fclose` function closes the file specified by `file_ptr`. Before the file is closed, any unwritten buffered data for the stream is written to the specified file. Any unread buffered data for the stream is discarded. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

When `fclose` closes the file, the file's output buffer is flushed, and the `FILE` object associated with the file is deleted. If the associated buffer was automatically allocated, it is deallocated. In VOS C, upon normal or abnormal termination of a program, all output buffers are automatically flushed, and all opened files are automatically closed.

If `fclose` is not successful in closing the file, `fclose` sets the external variables `errno` and `os_errno` to the error code number returned by the operating system. See the `close` function for the values that `fclose` uses when setting these variables.

Return Value

If `fclose` is successful in closing the file, the function returns the value 0.

If `fclose` detects an error while attempting to close the file, the function returns `EOF`.

Examples

The following program fragment uses `fclose` to close a file associated with a file pointer.

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

FILE *file_ptr;
```

(Continued on next page)

```
main()
{
   if ( (file_ptr = fopen("out_file", "w")) == NULL )
      {
      puts("Error occurred opening out_file.");
      exit(1);
      }
      .
      .
      .

   errno = 0;
   if ( fclose(file_ptr) != 0 )
     printf("Error occurred closing out_file:  errno = %d\n", errno);
}
```

Related Functions

`close`, `fopen`

fdopen

Purpose

Gets the file pointer for the file associated with a specified file descriptor.

Syntax

```
#include <stdio.h>

FILE *fdopen(int file_des, char *type);
```

Explanation

The `fdopen` function gets the file pointer for the file associated with the file descriptor, `file_des`. Once a file pointer has been obtained, you can use it to access the associated file with the standard I/O functions. The `file_des` argument is a file descriptor returned by a previous call to the `open` or `creat` function.

In VOS C, the `type` argument is ignored by `fdopen`. The type of I/O allowed for a file corresponds to the type of I/O that was specified when the file was opened with `open` or `creat`. For example, if you called `open` and specified `O_RDONLY` as the `o_flag`, the file can only be read. Even though the `type` argument is ignored by `fdopen`, you should include `type` when you call the function because the function prototype requires the argument. You can specify any pointer to `char` value, including `NULL`, for `type`.

Return Value

If successful, the `fdopen` function returns the file pointer associated with `file_des`.

If there is no file pointer associated with `file_des`, `fdopen` returns the `NULL` pointer constant. There will be no associated file pointer if, for example, the file has not been successfully opened by a previous call to `open` or `creat`.

Examples

The following program fragment uses `fdopen` to get the file pointer associated with a specified file descriptor.

```
#include <stdio.h>
#include <c_utilities.h>
#include <fcntl.h>
#include <stdlib.h>

int file_des;

FILE *file_ptr;
char *type = NULL;              /*  The value of type is ignored by
fdopen  */

main()
{
   if ( (file_des = open("in_file", O_RDWR)) == -1)
      {
      printf("Error opening in_file.\n");
      exit(1);
      }

/*  in_file is now opened for access with the UNIX I/O functions
*/
      .
      .
      .
   if ( (file_ptr = fdopen(file_des, type)) == NULL )
      {
      puts("Error getting file pointer with fdopen.");
      exit(1);
      }

/*  file_ptr can now be used to access in_file with standard I/O
functions  */
      .
      .
      .
   close(file_des);
}
```

After the call to `fdopen` returns a valid file pointer, `in_file` can be read or written using the standard I/O functions, such as `fscanf`, `fgetc`, `fprintf`, and `fputc`. Notice that a previous call to `open` specified that the file was to be opened for read and write I/O (`O_RDWR`). For the standard I/O functions, this same type of I/O is allowed.

Related Functions

`close`, `creat`, `fileno`, `open`

feof

Purpose

Determines whether the end-of-file indicator has been set for a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int feof(FILE *file_ptr);
```

Explanation

The `feof` function tests whether the end-of-file indicator has been set for the file specified by `file_ptr`. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

Certain C library functions, such as `fgetc` or `fread`, set the end-of-file indicator for a file if a read operation attempts to read **beyond** the end of the file. Typically, you might use the `feof` function after a read operation (for example, `fread`) does not return the expected number of characters. The `feof` function returns a nonzero value if the read operation encountered the end of the file.

The end-of-file indicator can be cleared by calling the `clearerr`, `fseek`, `fsetpos`, or `rewind` function and specifying the file pointer associated with the file. The end-of-file indicator is not set on an empty file.

Return Value

If the end-of-file indicator for the file specified by `file_ptr` has not been set by a previous input operation, the `feof` function returns the value 0. If the end-of-file indicator for the specified file has been set, the `feof` function returns a nonzero value.

Examples

The following program reads the contents of `text_file` and displays the contents on the terminal's screen. Then, it uses `feof` to determine whether the end-of-file indicator has been set for the file specified by `file_ptr`.

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

main()
{
   int c;
   FILE *file_ptr;

  if ( (file_ptr = fopen("text_file", "r")) == NULL ) /* "r" = reading
only */
      {
      printf("Could not open text_file.\n");
      exit(1);
      }

   while ( (c = fgetc(file_ptr)) != EOF)
      putchar(c);
   putchar('\n');

   if (feof(file_ptr))
      printf("We encountered the end of the file.\n");
   else
      printf("A read error occurred.\n");
}
```

Related Functions

`clearerr`, `ferror`, `fseek`, `fsetpos`, `perror`, `rewind`

ferror

Purpose

Determines whether the error indicator has been set for a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int ferror(FILE *file_ptr);
```

Explanation

The `ferror` function tests whether the error indicator has been set for the file specified by `file_ptr`. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

Certain C library functions set the error indicator for a file if an error occurs during an I/O operation on the file. Typically, you might use the `ferror` function immediately after a standard I/O function, such as `fgetc`, has returned a value that indicates an error may have occurred. The `ferror` function returns a nonzero value if the error indicator for a specified file has been set.

The error indicator for the file stays set until you call the `clearerr` or `rewind` function and specify the file pointer associated with the file. Closing the file with `fclose` also clears the file's error indicator.

Return Value

If the error indicator for the file specified by `file_ptr` has not been set, the `ferror` function returns the value 0. If the error indicator for the specified file has been set, the `ferror` function returns a nonzero value.

> **Note:** In VOS C, the `ferror` function does not return the error code number associated with the error. Examine `errno` or use `perror` to find out what specific error occurred.

Examples

The following program uses `ferror` to determine whether the error indicator has been set for the file specified by `file_ptr`.

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

main()
{
   char c;
   FILE *file_ptr;

  if ( (file_ptr = fopen("text_file", "r")) == NULL ) /* "r" = reading
only */
      {
      printf("Could not open text_file\n");
      exit(1);
      }

   c = 'a';
   fprintf(file_ptr, "%c", c);  /*  Creates an error:  attempt to
write      */
                         /*  to a file opened for reading only      */
   if (ferror(file_ptr))
      {
      printf("Error occurred: ferror returns nonzero.\n");

      clearerr(file_ptr);

      if ( (ferror(file_ptr)) == 0)
         printf("Error indicator cleared.\n");
      }
   else
      printf("No error occurred: ferror returns %d.\n",
ferror(file_ptr));
}
```

The output from the preceding program is as follows:

```
Error occurred: ferror returns nonzero.
Error indicator cleared.
```

Related Functions

`clearerr`, `feof`, `perror`, `rewind`

fflush

Purpose

Flushes the output buffer for a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int fflush(FILE *file_ptr);
```

Explanation

The `fflush` function flushes the output buffer for the file pointed to by `file_ptr`. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

If the file was opened for output, or was opened for update and the most recent operation on the file was output, `fflush` causes any unwritten buffered data associated with the file to be written. If the file was opened for update, the next operation, after the `fflush` call, can be either a read or a write operation.

If the file was opened for input, or was opened for update and the most recent operation on the file was input, `fflush` produces unpredictable results.

When `file_ptr` is `NULL`, `fflush` performs the flushing action on all files open for output, or opened for update where the most recent operation was output.

If an error occurs, `fflush` sets the external variables `errno` and `os_errno` to the error code number returned by the operating system. The following table lists some of the error code values that are possible.

Error Code Name	Number
`e$caller_must_wait`	1277
`e$invalid_file_pointer`	3929
`e$port_not_attached`	1021
`e$record_format_error`	4265
`e$record_too_long`	1041

Normally, most programs do not need to call `fflush` to flush the output buffer associated with a specified file. This type of buffer is automatically flushed when one of the following occurs:

- when the buffer becomes full
- when the file is closed
- when the program terminates.

In addition, when you open a file for update and the most recent operation on the file has been output, calling the `fseek`, `fsetpos`, or `rewind` function causes the system to flush the file's output buffer.

Return Value

The `fflush` function returns the value 0 if the flushing operation is successful.

If an error occurs, `fflush` returns `EOF`.

Examples

The following program fragment uses `fflush` to flush the output buffer for the file associated with a specified file pointer.

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

FILE *file_ptr;
int i;

char out_buffer[25] = "abcdefghijklmnopqrst";

main()
{
   if ( (file_ptr = fopen("out_file", "w+")) == NULL )
      {
      puts("Error occurred opening out_file.");
      exit(1);
      }

   for (i = 0; i < 20; i++)
         fputc( (int)out_buffer[i], file_ptr );

   if (fflush(file_ptr) == 0)
      puts("The file's output buffer was flushed.");
   else
      puts("The file's output buffer was not flushed.");
         .
         .
         .
   fclose(file_ptr);
}
```

Related Functions

`fclose`, `fopen`

fgetc

Purpose

Reads and returns the next character from a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int fgetc(FILE *file_ptr);
```

Explanation

The `fgetc` function reads the next character from the file pointed to by `file_ptr`. The function then moves the file-position indicator ahead one character in the file. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

The `fgetc` function is the function equivalent of the `getc` macro. However, `fgetc` always evaluates its argument exactly once because `fgetc` is always implemented as a function. In contrast, the `getc` macro may evaluate its argument more than once or not at all.

If the file is at end-of-file, `fgetc` sets the end-of-file indicator associated with `file_ptr`. If a read error occurs, the function sets the error indicator associated with `file_ptr`. You can use the `feof` and `ferror` functions to determine if end-of-file was encountered or if an error occurred.

If the file is at end-of-file or a read error occurs, the `fgetc` function sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `read` function for the possible values that `fgetc` uses when setting these variables.

Return Value

The `fgetc` function returns an integer value representing the next character from the specified file.

If the file is at end-of-file or a read error occurs, `fgetc` returns `EOF`.

Examples

The following program opens files named `in_file` and `out_file`. It uses `fgetc` to read each character from `in_file`. It uses `fputc` to write each character to `out_file`.

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

main()
{
   int c;
   FILE *file_ptr1;            /*  For in_file   */
   FILE *file_ptr2;            /*  For out_file  */

   if ( (file_ptr1 = fopen("in_file", "r")) == NULL )
      {
      printf("Could not open in_file.\n");
      exit(1);
      }

   if ( (file_ptr2 = fopen("out_file", "w")) == NULL )
      {
      printf("Could not open out_file.\n");
      exit(1);
      }

   while ( (c = fgetc(file_ptr1)) != EOF )
      fputc(c, file_ptr2);
}
```

Related Functions

`fputc`, `getc`, `getchar`, `putc`, `putchar`, `ungetc`

fgetpos

Purpose

Gets and stores the current value of the file-position indicator for a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int fgetpos(FILE *file_ptr, fpos_t *pos);
```

Explanation

The `fgetpos` function gets and stores the current value of the file-position indicator for the file specified by `file_ptr`. The function stores the current value in the variable pointed to by `pos`. The `pos` argument is a pointer to a variable of the type `fpos_t`, which is a type defined in the `stdio.h` header file. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

The `fgetpos` and `fsetpos` functions are used together. First, you call `fgetpos` to get and store the current position of a file. At a later time, you call `fsetpos` to return to that position. The value stored by `fgetpos` contains unspecified information that is useable by the `fsetpos` function for returning the file-position indicator for the specified file to its position at the time of the `fgetpos` call.

There are certain limitations when using `fgetpos` and `fsetpos` with a sequential file. Seeking on a sequential file is limited to a file that has been opened for reading only: either input mode or dirty-input mode. Seeking on a sequential file can result in inferior performance because the file-positioning function often must read many records to position to the correct offset.

If an error occurs, `fgetpos` sets the external variables `errno` and `os_errno` to the error code number returned by the operating system. The most commonly returned error code is `e$invalid_io_operation` (`1040`).

Return Value

If the operation succeeds, the `fgetpos` function returns the value 0.

If the operation fails, `fgetpos` returns a positive value corresponding to the operating-system error code number.

Examples

The following program uses `fgetpos` to get and store the current value of the file-position indicator. After data has been written to the file, the program then uses `fsetpos` to return to the earlier position so that the written data can be displayed.

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

fpos_t pos;
int return_value, c;
FILE *file_ptr;

main()
{
   if ( (file_ptr = fopen("ex_file", "a+")) == NULL )
      {
      printf("Could not open ex_file\n");
      exit(1);
      }

   if ( return_value = fgetpos(file_ptr, &pos) )
      printf("Error %d occurred in fgetpos\n", return_value);

   fprintf(file_ptr, "%s %d\n", "abcd", 4321);

   if ( return_value = fsetpos(file_ptr, &pos) )
      printf("Error %d occurred in fsetpos\n", return_value);

   while ( (c = fgetc(file_ptr)) != '\n')
      putchar(c);
   putchar('\n');
}
```

Related Functions

`fseek`, `fsetpos`, `ftell`

fgets

Purpose

Reads a line or a specified number of characters from a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

char *fgets(char *s, int n, FILE *file_ptr);
```

Explanation

The `fgets` function reads characters from the file associated with `file_ptr` until one of the following occurs.

- `fgets` reads **one less** than the number of characters specified in the `n` argument.
- `fgets` encounters the newline character (`\n`) or end-of-file.

The `s` argument is the address of an array of `char` of sufficient size to store the characters read plus a terminating null character. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

The characters that `fgets` reads are copied into the array pointed to by `s`. A null character is written immediately after the last character copied into the array. Unlike the `gets` function, `fgets` retains the newline character, if one is read.

If the file is at end-of-file, `fgets` sets the end-of-file indicator associated with `file_ptr`. If a read error occurs, the function sets the error indicator associated with `file_ptr`. You can use the `feof` and `ferror` functions to determine if end-of-file was encountered or if an error occurred.

When the file is at end-of-file or a read error occurs, the `fgets` function sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `read` function for the possible values that `fgets` uses when setting these variables.

Return Value

The `fgets` function returns `s`, a pointer to the array's first character if the characters are read successfully.

If end-of-file is encountered before any characters are read into the array, `fgets` returns the `NULL` pointer constant, and the contents of the array remain unchanged.

If a read error occurs, `fgets` returns the `NULL` pointer constant, and the contents of the array have unpredictable values.

Examples

The following program opens files named `in_file` and `out_file`. It uses `fgets` to read up to `n - 1` characters from `in_file`. It uses `fputs` to write the characters to `out_file`.

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

char s[100];
int n;
FILE *file_ptr1;            /*  For in_file   */
FILE *file_ptr2;            /*  For out_file  */

main()
{
   if ( (file_ptr1 = fopen("in_file", "r")) == NULL )
      {
      printf("Could not open in_file.\n");
      exit(1);
      }

   if ( (file_ptr2 = fopen("out_file", "w")) == NULL )
      {
      printf("Could not open out_file.\n");
      exit(1);
      }

   n = 10;

   if ( fgets(s, n, file_ptr1) == NULL )
      puts("File is empty, or a read error occurred.");
   else
      fputs(s, file_ptr2);
}
```

Related Functions

`fputs`, `gets`, `puts`

fileno

Purpose

Gets the file descriptor for the file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int fileno(FILE *file_ptr);
```

Explanation

The `fileno` macro gets the integer file descriptor for the file associated with `file_ptr`. The `file_ptr` argument is a valid file pointer returned, for example, by a previous call to `fopen` or `freopen`, or it can be one of the preopened standard files, such as `stdin`.

You can use the file descriptor returned by `fileno` with the UNIX I/O functions, such as `read` and `write`, found in the `c_utilities.h` header file.

The `fileno` macro assumes that the `file_ptr` argument is valid. If the value passed to `fileno` is not a valid file pointer, the behavior is unpredictable.

Return Value

The `fileno` macro returns the non-negative integer descriptor for the file associated with `file_ptr`.

If the value passed to `fileno` is not a valid file pointer, the macro returns an unpredictable value when an error does not occur.

Examples

The following program fragment uses `fileno` to get the file descriptor for the file associated with a file pointer.

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

FILE *file_ptr;
int file_des;
```

(Continued on next page)

```
main()
{
   errno = 0;
   if ( (file_ptr = fopen("test_file", "r")) == NULL )
      {
      printf("Error opening test_file:  errno = %d\n", errno);
      exit(1);
      }

   file_des = fileno(file_ptr);

   /*  file_des can be used with the UNIX I/O functions.  */
      .
      .
      .
}
```

creat, fopen, freopen, open

floor

Purpose

Computes the largest integer value that is less than or equal to a specified floating-point number.

Syntax

```
#include <math.h>

double floor(double x);
```

Explanation

The `floor` function calculates the largest integer value that is less than or equal to `x`. The `x` argument is a value of the type `double`. You can use `floor` to round a floating-point value down to an integer.

If the value of `x` is too large (greater than approximately 1.84 multiplied by 10^{19}), the `floor` function sets `errno` to `e$big_arg_causes_error` (`3583`).

Return Value

The `floor` function returns the floor of `x`: the largest integer value that is less than or equal to `x`. If `x` is 0, `floor` returns 0.

If the value of `x` is too large, `floor` returns `x`.

Examples

The following program uses `floor` to calculate the floor of a floating-point value entered at the terminal's keyboard.

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

short e$big_arg_causes_error;      /*  error code 3583  */

main()
{
   double x, flr;

  printf("Enter the number for which you want to find the floor.\n");
   scanf("%lf", &x);
```

(Continued on next page)

```
     errno = 0;
     flr = floor(x);

     if (errno == e$big_arg_causes_error)
        puts("Argument produces an invalid result.");
     else
        printf("The floor of %.3lf = %.3lf\n", x, flr);
  }
```

If `-123.456` were entered at the terminal's keyboard, the output would be as follows:

```
The floor of -123.456 = -124.000
```

Related Functions

`ceil`, `fabs`

fmod

Purpose

Computes the remainder after dividing one floating-point value by another floating-point value.

Syntax

```
#include <math.h>

double fmod(double x, double y);
```

Explanation

The `fmod` function calculates the floating-point remainder of `x / y`. The returned value has the same sign as `x`. Both the `x` and `y` arguments are values of the type `double`.

Return Value

The `fmod` function returns the floating-point remainder of `x / y`, with the returned value having the same sign as `x`. If `y` is the value 0, `fmod` returns 0.

When `x / y` creates an overflow condition, `fmod` returns the value 0. When `x / y` creates an underflow condition, the function returns `x`.

Examples

The following program uses `fmod` to calculate the floating-point remainder of `x / y`.

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

main()
{
   double x, y, remainder;

   printf("Enter a number for x.\n");
   scanf("%lf", &x);
   printf("Enter a number for y.\n");
   scanf("%lf", &y);

   remainder = fmod(x, y);

   printf("%.3lf divided by %.3lf = a remainder of %.3lf\n", x, y,
remainder);
}
```

If two values, `-10.0` for `x` and `-2.99` for `y`, were entered at the terminal's keyboard, the output would be as follows:

```
-10.000 divided by -2.990 = a remainder of -1.030
```

Related Functions

`ceil`, `fabs`, `floor`

fopen

Purpose

Opens a file and associates a stream with the file.

Syntax

```
#include <stdio.h>

FILE *fopen(const char *path_name, const char *mode);
```

Explanation

The `fopen` function opens the file identified by `path_name`, associates a stream with the file, and initializes the stream. The `mode` argument indicates how `fopen` is to open the file. For example, `mode` indicates the type of I/O for which the file will be opened. The `path_name` and `mode` arguments are both character strings. The values allowed for `path_name` and `mode` are explained in the sections that follow.

You use the file pointer returned by `fopen` to identify the file with many of the standard I/O functions such as `fgetc`, `fprintf`, and `fscanf`.

A file opened with `fopen` is fully buffered only if it can be determined not to point to an interactive device. When `fopen` is successfully called, the function clears the error and end-of-file indicators for the file.

If an error occurs during the file-opening operation, `fopen` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `open` function for the possible values that `fopen` uses when setting these variables.

> **Note:** If the values specified in the `path_name` argument conflict with the values specified in the `mode` argument, or if erroneous values are specified in either argument, the results are unpredictable.

Path Name Argument. In VOS C, the `path_name` argument is a character string that has the following form:

"*file_path_name* [option ...]"

Notice that a space is required between the *`file_path_name`* and the first option.

Within the `path_name` argument, the required *`file_path_name`* is a full or relative path name specifying the name of the file or device to open. The `path_name` can also include one or more *`option`* values that indicate how the file is to be opened, including the following:

- file organization
- locking mode
- raw input and output modes (for a terminal device only)
- bulk raw input mode (for a terminal device only)
- associated port.

See "Path Name Options," later in the Explanation, for more information on these optional parts of the `path_name` argument.

Mode Argument. In VOS C, the `mode` argument is a character string that has the following form:

```
"value[option ...]"
```

The `mode` argument must have the required *`value`* part and can include one or more *`option`* parts.

Within `mode`, *`value`* gives information about how the file is to be opened. For example, the *`value`* part of `mode` determines whether the file will be opened for input, output, or update, and whether the file will be opened in text or binary mode.

In addition, `mode` can include certain optional values, which are explained in "VOS C Mode Options" later in the Explanation.

The required *`value`* part of the `mode` argument can have one of the values shown in Table 11-25.

Table 11-25. Mode Values for the `fopen` Function *(Page 1 of 2)*

Mode Value	Description
`r`	Opens a text file for reading.
`w`	Opens a text file for writing. If the file specified by `path_name` exists, it is truncated. If the file does not exist, it is created.
`a`	Opens a text file for writing at end-of-file (that is, append mode). If the file specified by `path_name` does not exist, it is created.
`d`	Opens a text file for reading using dirty-input mode. With dirty input, normal locking rules are ignored.
`rb`	Opens a binary file for reading.
`wb`	Opens a binary file for writing. If the file specified by `path_name` exists, it is truncated. If the file does not exist, it is created.
`ab`	Opens a binary file for writing at end-of-file. If the file specified by `path_name` does not exist, it is created.
`r+`	Opens a text file for update (reading and writing).
`w+`	Opens a text file for update. If the file specified by `path_name` exists, it is truncated. If the file does not exist, it is created.

Table 11-25. Mode Values for the `fopen` Function *(Page 2 of 2)*

Mode Value	Description
`a+`	Opens a text file for update. All writing takes place at end-of-file. If the file specified by `path_name` does not exist, it is created.
`rb+` or `r+b`	Opens a binary file for update.
`wb+` or `w+b`	Opens a binary file for update. If the file specified by `path_name` exists, it is truncated. If the file does not exist, it is created.
`ab+` or `a+b`	Opens a binary file for update. All writing takes place at end-of-file. If the file specified by `path_name` does not exist, it is created.

Input Mode. If the first character in the `mode` argument is `r`, `fopen` opens a file for input. In input mode, the file must exist. If the file does not exist, the `fopen` function returns the `NULL` pointer constant and sets the external variables `errno` and `os_errno` to `e$object_not_found` (`1032`).

Output Mode. If the first character in the `mode` argument is `w`, `fopen` opens a file for output. In output mode, `fopen` creates the file if the file does not exist, or deletes the file's contents if the file does exist.

Append Mode. If the first character in the `mode` argument is `a`, `fopen` opens a file in append mode. By default, with append mode, it is impossible to overwrite information already in the file. The file-positioning functions, such as `fseek`, can be used to change the file-position indicator to any position in the file. However, with append I/O, the file-position indicator is, by default, ignored for write operations. All output is written to the end of the file, and the file-position indicator is repositioned to the end of the output.

When you specify an `a` character **without** a `+` character in the `mode` argument, you can change the behavior of append mode by assigning values to the `_TRADITIONAL_` external variable, which is declared in the `stdio.h` header file.

- If you assign the value 0 to `_TRADITIONAL_`, append mode works as specified in the ANSI C Standard. All writing takes place at end-of-file **regardless** of intervening calls to a file-positioning function, such as `fseek`. If you do not assign a value to `_TRADITIONAL_`, the preceding behavior is the default.

- If you assign a nonzero value to `_TRADITIONAL_`, append mode works as specified in many UNIX implementations. For stream, fixed, and relative files, all writing takes place at the file's current position as defined by the file-position indicator. Thus, calls to `fseek` change the current position for both read and write operations.

For a sequential file, the `_TRADITIONAL_` external variable does **not** affect the behavior of append mode. All writing takes place at end-of-file regardless of calls to a file-positioning function.

In VOS C, when two separate processes open the same file for append I/O, each process can write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

Update Mode. If the second or third character in the `mode` argument is `+`, `fopen` opens the file for update. In update mode, you can perform both input and output on the file. However, output **cannot** be directly followed by input without an intervening call to the `fflush`, `fseek`, `fsetpos`, or `rewind` function. Input **cannot** be directly followed by output without an intervening call to `fseek`, `fsetpos`, or `rewind`, or an input operation that encounters end-of-file.

Dirty-Input Mode. If you use `d` in the `mode` argument, `fopen` opens the file for reading using dirty input I/O type. The program can read from the file even though another program might be modifying the file. However, it is not guaranteed that a dirty-input reader will get a consistent view of the file's data. The `d` value is used in place of the `r`, `w`, or `a` value. The `d` value and the `r`, `w`, `a`, `b`, and `+` values are mutually exclusive.

VOS C Mode Options. In VOS C, the `mode` argument to `fopen` can include optional characters specifying the file organization of a file that `fopen` creates, locking mode, and other information. You append these VOS C extensions following the required *`value`* part of `mode`.

The allowed values for these optional characters are shown in Table 11-26, grouped by category. You can select only one value from each category. For example, you can select only one file organization in an `fopen` call. More information on each category of option appears later in the Explanation.

Table 11-26. Mode Options in `fopen` *(Page 1 of 2)*

Type of Option	Mode Value	Description
File Organization	`s`	When specified with `w` or `a`, creates and opens a file with stream file organization if no file of the specified name exists. When you do not specify a file organization option when creating a file, stream file organization is the default.
	`q`	When specified with `w` or `a`, creates and opens a sequential file if no file of the specified name exists.
	`f` *`size`*	When specified with `w` or `a`, creates and opens a fixed file if no file of the specified name exists. You give the file's record size in *`size`*.
	`v` *`size`*	When specified with `w` or `a`, creates and opens a relative file if no file of the specified name exists. You give the file's maximum record size in *`size`*.
Locking Mode	`i`	Opens the file using implicit-locking mode.
	`y`	Opens the file using set-lock-don't-wait locking mode.
	`z`	Opens the file using wait-for-lock locking mode.
	`g`	Opens a file with stream file organization using region-locking locking mode, which allows the `lockf` function to be used for region locking.

Table 11-26. Mode Options in `fopen` *(Page 2 of 2)*

Type of Option	Mode Value	Description
	`c`	Opens the file using record-locking locking mode.
Raw Input and Output Modes	`u`	Causes all terminal access to be performed using raw input mode and raw output mode. If you do not specify raw mode (`u`), the default input mode is command-line input mode, and the default output mode is generic (or normal) output.
Bulk Raw Input Mode	`k`	Causes all terminal input to be performed using bulk raw input mode. If you do not specify bulk raw input mode (`k`), the default input mode is normal raw input mode.
Associated Port	`p` *`name`*	Attaches a port of the name indicated in *`name`* to the specified file.

File Organization. You indicate a file organization **only when** you are creating a file. You can create a file by specifying `w` or `a` in the `mode` argument. If you specify one of the file organization options and a file of the name specified in *`file_path_name`* does not exist, the `fopen` function creates a file of the given type. If a file of the name specified in *`file_path_name`* does exist, the `fopen` function ignores the file organization option.

See the *VOS C User's Guide (R141)* for information on file organizations.

Locking Mode. Within the `mode` argument, you can indicate one of five locking modes. If you do not explicitly specify a locking mode in `fopen` and the file is not opened in input mode (`r`) or dirty-input mode (`d`), implicit-locking mode is the default. If the file is opened in input mode or dirty-input mode, normal locking rules are ignored, and the locking mode that you specify in `fopen` has no effect on I/O operations.

For information on region-locking locking mode, see the description of the `lockf` function in Chapter 11. For information on the other locking modes shown in Table 11-26, see the *VOS C User's Guide (R141)*.

Raw Input and Output Modes. Within `mode`, you can append two optional characters to control input and output with a terminal:

- `u` specifies raw input mode and raw output mode.
- `k` specifies bulk raw input mode.

If you use `fopen` to set the terminal into raw mode or bulk raw input mode, the input-output mode for the terminal is automatically reset when the terminal is closed. See the *VOS Communications Software: Asynchronous Communications (R025)* manual for detailed information on input and output modes with a terminal device.

Raw Mode. The `u` option tells `fopen` to set the input-output mode for the terminal, specified by *`file_path_name`*, to raw mode. In raw mode, the terminal is accessed using raw input mode and raw output mode. Raw mode can be specified for terminal devices only.

For input from a terminal, the `u` option sets the device in normal raw input mode. With a terminal, normal raw input mode does not return to the program until the buffer is full. For this reason, normal raw input mode is rarely used. Bulk raw input mode (`k`), discussed later in the Explanation, provides a useful alternative. For output to a terminal, the `u` option causes `s$write_raw` calls to be issued rather than `s$seq_write` calls.

Bulk Raw Input Mode. The `k` option tells `fopen` to set the input mode for the terminal, specified by *`file_path_name`*, to bulk raw input mode. Bulk raw input mode can be specified for terminal devices only. In bulk raw input mode, the `s$read_raw` subroutine returns any available characters, even if it has not read enough characters to fill the input buffer.

Associated Port. If you append `p` and *`name`* to the `mode` argument, the `fopen` function attaches a port named *`name`* to the file specified in *`file_path_name`*. The *`name`* can be 1 to 32 characters long. If the port specified in *`name`* is already attached, `fopen` accesses the file attached to the port and ignores *`file_path_name`*.

Path Name Options. The syntax for the `path_name` argument to `fopen` is as follows:

```
"file_path_name [option ...]"
```

In VOS C, the `path_name` argument can include one or more *`option`* values specifying the file organization of a file that `fopen` creates, locking mode, raw input and output modes, bulk raw input mode, and associated port. Notice that a space is required between *`file_path_name`* and the first option.

If the `path_name` argument includes a conflicting or duplicate path name option, `fopen` sets the external variables `errno` and `os_errno` to `e$title_inconsistent` (`1310`). If the options specified contain a syntax error, `fopen` sets `errno` and `os_errno` to `e$bad_title_syntax` (`1308`).

Each option available in `path_name` can alternatively be specified by a corresponding option in the `mode` argument. That is, you can specify similar information with a `mode` option **or** with a `path_name` option. The `path_name` options are included so that `fopen` will be compatible with previous VOS C implementations of that function.

The allowed values for the path name options are shown in Table 11-27, grouped by category. You can select only one value from each category. For example, you can select only one file organization in an `fopen` call.

Table 11-27. Path Name Options in `fopen`

Type of Option	Path Name Option	Description
File Organization	`-stream`	When specified with `w` or `a`, creates and opens a file with stream file organization if no file of the given name exists. When you do not specify a file organization option when creating a file, stream file organization is the default.
	`-sequential`	When specified with `w` or `a`, creates and opens a sequential file if no file of the given name exists.
	`-fixed` *`size`*	When specified with `w` or `a`, creates and opens a fixed file if no file of the given name exists. You give the file's record size in *`size`*.
	`-relative` *`size`*	When specified with `w` or `a`, creates and opens a relative file if no file of the given name exists. You give the file's maximum record size in *`size`*.
Locking Mode	`-implicit_locking` or `-implicitlock`	Opens the file using implicit locking mode. If you do not specify a locking mode option, implicit locking is the default.
	`-nowait`	Opens the file using set-lock-don't-wait locking mode.
	`-wait`	Opens the file using wait-for-lock locking mode.
Raw Input and Output Modes	`-raw_mode`	Causes all terminal access to be performed using raw input mode and raw output mode. If you do not specify raw mode, the default input mode is command-line input mode, and the default output mode is generic (or normal) output.
Bulk Raw Input Mode	`-bulk_raw_input_mode`	Causes all terminal input to be performed using bulk raw input mode. If you do not specify bulk raw input mode, the default input mode is normal raw input mode.
Associated Port	`-port_name` *`name`*	Attaches a port of the name indicated in *`name`* to the specified file.

As an alternative, you can specify each of the path name options through the use of an optional value in the `mode` argument. Because `fopen` checks that the specified `mode` values are allowed, it is **recommended** that you use the `mode` argument, rather than the `path_name` argument, to indicate these options. Table 11-28 shows the correspondences between the values of the `path_name` argument and the values of the `mode` argument.

Table 11-28. Correspondences between the Path Name and Mode Options

Path Name Option	Corresponding Mode Option
`-fixed` *size*	`f` *size*
`-relative` *size*	`v` *size*
`-sequential`	`q`
`-stream`	`s`
`-implicit_locking` or `-implicitlock`	`i`
`-nowait`	`y`
`-wait`	`z`
`-raw_mode`	`u`
`-bulk_raw_input_mode`	`k`
`-port_name` *name*	`p` *name*

For information on the action performed by `fopen` for each `path_name` option, see the section earlier in the Explanation that explains the corresponding `mode` option.

Return Value

If the `fopen` function opens the file successfully, it returns a pointer to the `FILE` structure associated with the file.

If `fopen` fails to open the file, it returns the `NULL` pointer constant.

Examples

The following program fragment contains three examples of how `fopen` is used to open a file.

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

main()
{
   FILE *file_ptr1, *file_ptr2, *file_ptr3;

   errno = 0;
   if ((file_ptr1 = fopen("test_file1", "r+")) == NULL)       /*
Example 1  */
      {
      printf("Error opening test_file1:  errno = %d\n", errno);
      exit(1);
      }
```

(Continued on next page)

```
   errno = 0;
   if ((file_ptr2 = fopen("test_file2", "wif 40")) == NULL)   /*
Example 2  */
      {
      printf("Error opening test_file2:  errno = %d\n", errno);
      exit(1);
      }

   errno = 0;
   if ((file_ptr3 = fopen("test_file3", "rp port_3")) == NULL)/*
Example 3  */
      {
      printf("Error opening test_file3:  errno = %d\n", errno);
      exit(1);
      }
      .
      .
      .
}
```

In example 1, `fopen` opens a file named `test_file1` for update (reading and writing) because `mode` is `r+`.

In example 2, `fopen` opens a file named `test_file2` for output because the first character of `mode` is `w`. If `test_file2` exists, `fopen` deletes its contents. If `test_file2` does not exist, `fopen` creates a fixed file with a record size of 40 bytes because `mode` specifies `f 40`. In addition, `fopen` opens the file for implicit locking because `mode` specifies `i`.

In example 3, `fopen` opens a file named `test_file3` for input because the first character of `mode` is `r`. In addition, `fopen` attaches a port named `port_3` to the specified file because `mode` specifies `p port_3`. If `port_3` is already attached, `fopen` returns `NULL` and sets `errno` to `e$invalid_access_mode` (`1071`).

Related Functions

`creat`, `fclose`, `freopen`, `open`

`fprintf`

Purpose

Writes a formatted string of characters to a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int fprintf(FILE *file_ptr, const char *format, ...);
```

Explanation

The `fprintf` function writes output to the file pointed to by `file_ptr`. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`. The output written is under the control of the string pointed to by `format`.

> **Note:** The `fprintf` function does not support generic output sequences embedded within the format string. To write generic output to a terminal, use any of the write subroutines except `s$write_raw`. See the *VOS Communications Software: Asynchronous Communications (R025)* manual for information on generic output.

The string pointed to by `format` can consist of zero or more directives. The `fprintf` function executes each directive in turn. It returns when it encounters the end of the format string. The directives consist of the following:

- characters, including escape sequences
- conversion specifications.

The `fprintf` function writes ordinary characters (not `%`) unchanged to the file. Each *conversion specification* begins with a `%` and indicates how the value of a subsequent argument will be converted and written. The number and types of the arguments in an `fprintf` call depend on the conversion specifications given in the format string. Each conversion specification results in fetching zero or more arguments. In an `fprintf` call, the syntax for the argument list is as follows:

```
fprintf(file_ptr, format, [argument]...)
```

Each *`argument`* must be an expression yielding an object type appropriate for the conversion specification. If there are insufficient arguments for the format, the conversions have unpredictable results. If the format is exhausted while arguments remain, the extra arguments are evaluated but otherwise ignored.

Conversion Specifications. The format for a conversion specification is as follows:

```
%[flags][width] .[precision][modifier] conversion_specifier
```

For information on each element of a conversion specification, see the `printf` function.

The following table provides a summary of the formatted output conversion specifiers. The table shows some of the argument types typically used with each specifier, and a description and example of the output generated by each specifier. The `fp` argument is a file pointer that specifies a file into which output will be written.

(Page 1 of 2)

Conversion Specifier	Argument Type	Output
`c`	`int`	Writes a single character. For example,
	`char`	`fprintf(fp, "%c", 'A')` writes `A` to the file.
`d, i`	`int`	Writes a signed decimal. For example,
		`fprintf(fp, "%d", -123)` writes `-123` to the file.
`e, E`	`double`	Writes a floating-point value using exponential
	`float`	notation. For example, `fprintf(fp, "%e", -987.654)` writes `-9.876540e+002` to the file. The `E` conversion specifier writes a number with an `E` instead of an `e` introducing the exponent.
`f`	`double`	Writes a floating-point value as a decimal
	`float`	fraction. For example, `fprintf(fp, "%f", -987.345)` writes `-987.345000` to the file.
`g, G`	`double`	Writes a floating-point value using `e` or `f`
	`float`	format, whichever is more compact. For example, `fprintf(fp, "%g", -987.3456)` writes `-987.346` to the file. The `G` conversion specifier writes the number with an `E` instead of an `e` introducing the exponent.
`n`	`int *`	Writes, into the pointed-to `int` variable, the number of characters output so far by this `fprintf` call.
`o`	`unsigned int`	Writes an unsigned integer in octal format. For example, `fprintf(fp, "%o", 0177)` writes `177` to the file.

(Page 2 of 2)

Conversion Specifier	Argument Type	Output
`p`	`void *`	Writes a pointer as an absolute address in hexadecimal format. For example, `fprintf(fp, "%p", NULL)` writes `00000000` to the file.
`s`	`char *`	Writes a C string up to (but not including) the null character. For example, `fprintf(fp, "%s", "abcd")` writes `abcd` to the file.
`v`	`char_varying (`*`n`*`) *`	Writes a `char_varying` string up to the current length. For example, if `cv` is a `char_varying` object storing `"efgh"`, the call `fprintf(fp, "%v", &cv)` writes `efgh` to the file.
`u`	`unsigned int`	Writes an unsigned integer in decimal format. For example, `fprintf(fp, "%u", 1234)` writes `1234` to the file.
`x, X`	`unsigned int`	Writes an unsigned integer in hexadecimal format. For example, `fprintf(fp, "%x", 0x7F)` writes `7f` to the file. The `X` conversion specifier writes the number with the uppercase letters `ABCDEF` instead of the lowercase letters `abcdef`.
`%%`	none	Writes a percent sign to the file.

Return Value

The `fprintf` function returns the number of characters written to the file if the formatted output specified is successfully written.

If an output error occurs, `fprintf` returns a negative value.

Examples

The following program uses `fprintf` to write four lines of formatted output to the file `out_file`.

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

int i_num;
double d_num;
int num;
int return_value;

FILE *file_ptr;

main()
{
   if ( (file_ptr = fopen("out_file", "w")) == NULL )
      {
      printf("Could not open out_file\n");
      exit(1);
      }

   /*  This first conversion specification uses the 0 flag and a field
*/
   /*  width of 5 with the d conversion specifier.                    */

   i_num = 99;
   fprintf(file_ptr, "i_num = |%05d|\n", i_num);

   /*  The next conversion specification uses the # flag, a field        */
   /*  width of 10, and precision of 5 with the g conversion specifier.
*/

   d_num = (double)999;
   fprintf(file_ptr, "d_num = |%#10.5g|\n", d_num);

   /*  The last three conversion specifications use:  the n specifier to
*/
   /*  get the number of characters written, and the d specifier to print
*/
   /*  that number and the return value of the previous fprintf call.
*/

   return_value = fprintf(file_ptr, "12345%n\n", &num);
   fprintf(file_ptr, "number of characters written = %d; return_value =
%d\n",
          num, return_value);
}
```

The output written to the file is as follows:

```
i_num = |00099|
d_num = |    999.00|
12345
number of characters written = 5; return_value = 6
```

Related Functions

`printf`, `sprintf`, `vfprintf`, `vprintf`, `vsprintf`

`fputc`

Purpose

Writes one character to a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int fputc(int c, FILE *file_ptr);
```

Explanation

The `fputc` function writes one character, `c`, to the current position of the file associated with `file_ptr`. The function then moves the file-position indicator ahead one character in the stream. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

The `fputc` function is the function equivalent of the `putc` macro. However, `fputc` always evaluates its argument exactly once because `fputc` is always implemented as a function. In contrast, the `putc` macro may evaluate its argument more than once or not at all.

If a write error occurs, the `fputc` function sets the error indicator associated with `file_ptr`. You can use the `ferror` function to determine if an error occurred.

When a write error occurs, `fputc` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `write` function for the possible values that `fputc` uses when setting these variables.

Return Value

The `fputc` function returns an integer value representing the character written to the file.

If a write error occurs, `fputc` returns `EOF`.

Examples

The following program opens one file, `in_file`, and reads each character from the first line of `in_file`. It uses `fputc` to write each character from the line to the preopened file `stdout`.

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

main()
{
   int c;
   FILE *file_ptr;              /*  For in_file   */

   if ( (file_ptr = fopen("in_file", "r")) == NULL )
      {
      printf("Could not open in_file\n");
      exit(1);
      }

   while ( (c = fgetc(file_ptr)) != EOF )
      {
      if (fputc(c, stdout) == EOF)
         printf("Write error occurred in fputc.\n");
      if (c == '\n')
         break;
      }
}
```

Related Functions

`fgetc`, `getc`, `getchar`, `putc`, `putchar`

fputs

Purpose

Writes a string to a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int fputs(const char *s, FILE *file_ptr);
```

Explanation

The `fputs` function writes a string, `s`, to the current position of the file associated with `file_ptr`. The `s` argument specifies the address of a null-terminated array of `char`. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

The string's null character is not written to the file. In contrast to the `puts` function, `fputs` does **not** append a newline character (`\n`) to the string.

If a write error occurs, the `fputs` function sets the error indicator associated with `file_ptr`. You can use the `ferror` function to determine if an error occurred. When a write error occurs, `fputs` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `write` function for the possible values that `fputs` uses when setting these variables.

Return Value

The `fputs` function returns the value 0 if the string is successfully written.

If a write error occurs, `fputs` returns a nonzero value, specifying the appropriate operating-system error code.

Examples

The following program opens a file, `out_file`, and uses `fputs` to write a string to the file.

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

char *s = "data to write to out_file";

main()
{
   FILE *file_ptr;             /*  For out_file   */
   int return_value;

   if ( (file_ptr = fopen("out_file", "w")) == NULL )
      {
      printf("Could not open out_file.\n");
      exit(1);
      }

   return_value = fputs(s, file_ptr);

   if (return_value != 0)
      puts("Write error occurred in fputs.");
}
```

Related Functions

`fgets`, `gets`, `puts`

fread

Purpose

Reads a specified number of data items, each of an indicated size, from a file associated with a given file pointer.

Syntax

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t number, FILE *file_ptr);
```

Explanation

The `fread` function reads up to `number` items of data from the file specified by `file_ptr` into the array pointed to by `ptr`. The `size` argument indicates the number of bytes in each data item to be read. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

When `size` or `number` is 0, the contents of the array pointed to by `ptr` and the state of the stream remain unchanged.

The data is read starting at the file's current position. After `fread` reads data from the file, it increments the file-position indicator, if defined, by the number of bytes successfully read. The file-position indicator's resulting value is unpredictable if an error occurs or if a partial data item is read.

The `fread` function is typically used to read data from a file that is opened in binary mode. In contrast to the formatted I/O functions such as `fscanf`, `fread` performs no conversions before storing the data in the array located by `ptr`.

If an error occurs, `fread` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `read` function for the possible values that `fread` uses when setting these variables.

You can tell when end-of-file has been reached by examining `errno` after the return value of `fread` is less than the number of data items specified in the `number` argument. In such a case, if `errno` equals `e$end_of_file` (`1025`), end-of-file has been reached before `number` data items could be read.

Return Value

The `fread` function returns the number of data items successfully read from the file. The number of data items read can be less than `number` if a read error occurs or if the end of the file is encountered before the specified number of items is read.

If `size` or `number` is 0, `fread` returns 0.

Examples

The following program fragment uses `fread` to read a specified number of `struct rec` size data items from an existing file opened in binary mode. The `fread` function stores the data items in an array of structures.

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

#define NUM_RECS 10

FILE *file_ptr;

struct rec
   {
   char name[30];
   int number;
   } rec_array[NUM_RECS];

size_t items_read;
int i;

main()
{
   if ( (file_ptr = fopen("binary_file", "rb")) == NULL )
      {
      puts("Error opening binary_file.");
      exit(1);
      }

   for (i = 0; i < NUM_RECS; i++)
      {
      items_read = fread(&rec_array[i], sizeof(struct rec), 1,
file_ptr);

      if (items_read < 1)
         if (ferror(file_ptr))
            {
            puts("Error reading binary_file.");
            abort();
            }
```

(Continued on next page)

```
            else
               break;          /*  Only other reason is EOF  */
         }
            .
            .
            .
      fclose(file_ptr);
   }
```

Related Functions

`fwrite`, `read`

`free`

Purpose

Deallocates a specified block of memory previously allocated with `calloc`, `malloc`, or `realloc`.

Syntax

```
#include <stdlib.h>

void free(void *ptr);
```

Explanation

The `free` function deallocates, or frees for reuse, a block of memory pointed to by `ptr`. The operating system can use the freed memory for further allocation if a subsequent call to `calloc`, `malloc`, or `realloc` occurs. The `ptr` argument must be a pointer returned by a previous call to `calloc`, `malloc`, or `realloc`.

The `free` function does nothing if one of the following is true.

- `ptr` is `NULL`.
- `ptr` is not a pointer previously returned by `calloc`, `malloc`, or `realloc`.
- `ptr` is a pointer that specifies memory freed in an earlier call to `free` or `realloc`.

All of the preceding are incorrect programming practices that can produce unpredictable results on operating systems other than VOS.

The `free` function allows you to free storage dynamically at run time. The amount of memory a program allocates or frees can be determined by run-time events, such as user input or the amount of data in a file.

A pointer to an object becomes invalid if the memory containing the object is deallocated. Referencing an object whose memory has been deallocated can yield unpredictable results.

Return Value

The `free` function returns no value.

Examples

The following program uses `free` to deallocate a block of memory used to store a structure. The memory was previously allocated by a call to `malloc`.

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

struct item
   {
   char name[100];
   double amount;
   };

struct item *ptr;

main()
{
   ptr = (struct item *) malloc( sizeof(struct item) );

   if (ptr == NULL)
      {
      puts("Memory was not successfully allocated.");
      exit(1);
      }
   else
      puts("Memory was successfully allocated.");

   free(ptr);
}
```

Related Functions

`alloca`, `calloc`, `malloc`, `realloc`

freopen

Purpose

Closes the file associated with a specified file pointer, opens another file, and associates the file pointer and stream with that file.

Syntax

```
#include <stdio.h>

FILE *freopen(const char *path_name, const char *mode, FILE
*file_ptr);
```

Explanation

The `freopen` function closes an open file specified by `file_ptr`. Then, in the same manner as `fopen`, `freopen` opens the file identified by `path_name` and associates `file_ptr` with the file. The `mode` argument indicates how `freopen` is to open the file. For instance, `mode` indicates the type of I/O for which the file will be opened. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

The allowed values and uses of the `path_name` and `mode` arguments for `freopen` are identical to the values and uses of the same arguments in the `fopen` function. See the `fopen` function for information on these arguments.

Typically, `freopen` is used to change the file associated with one of the standard streams, `stdin`, `stdout`, or `stderr`.

When `freopen`'s attempt to close the file specified by `file_ptr` fails, the function, nevertheless, associates the file pointer with the file indicated by `path_name`.

If an error occurs during the file-opening operation, `freopen` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `open` function for the possible values that `freopen` uses when setting these variables.

Return Value

If the `freopen` function opens the file successfully, it returns a pointer to the `FILE` structure associated with the file.

If `freopen` fails to open the file, the function returns the `NULL` pointer constant.

Examples

The following program uses `freopen` to close `stdout` and to open a file for output. All output specified for `stdout` (the terminal's screen) will be redirected to this file.

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

main()
{
   FILE *file_ptr;

   errno = 0;
   if ((freopen("output_file", "w", stdout)) == NULL)
      {
      printf("Error opening output_file:  errno = %d\n", errno);
      exit(1);
      }

   puts("This output would normally be written to the terminal's
screen.");

   puts("Since freopen has associated stdout with output_file, ");
   puts("this output will instead be written to output_file.");
}
```

Related Functions

`fclose`, `fopen`

frexp

Purpose

Breaks down a floating-point value into a mantissa (range: 0.5 to less than 1.0) and an integer exponent.

Syntax

```
#include <math.h>

double frexp(double value, int *exp_ptr);
```

Explanation

The `frexp` function breaks down a floating-point value, `value`, into a normalized mantissa and an integer exponent. The returned mantissa ranges from 0.5 to less than 1.0, or the mantissa equals 0 if `value` is 0. The `frexp` function returns the mantissa and, as a side effect, stores the exponent in the `int` variable pointed to by `exp_ptr`. The formula used to compute the floating-point value is as follows:

$$\texttt{value = mantissa * 2}^{|\text{exponent}|}$$

The `value` argument, a `double` value, contains the floating-point number to break down. The `exp_ptr` argument is the address of an `int` variable. The `frexp` function uses the variable pointed to by `exp_ptr` to store the integer exponent that the function calculates. If `value` is 0, the function stores 0 in the variable pointed to by `exp_ptr`.

See Appendix A for more information on the format of floating-point values.

Return Value

The `frexp` function returns a mantissa for the floating-point value `value`, using the procedure described in the Explanation. If `value` is not equal to 0, `frexp` returns a value in the range 0.5 to less than 1.0. If `value` is equal to 0, `frexp` returns 0.

Examples

The following program uses `frexp` to break down a floating-point value, entered at the terminal's keyboard, into a mantissa and an exponent.

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

main()
{
   double value, mantissa;
   int exponent;

   printf("Enter a number that you want to break down ");
   printf("into a mantissa and an exponent.\n");
   scanf("%lf", &value);

   mantissa = frexp(value, &exponent);

   printf("The floating-point value %.3lf = ", value);
   printf("(the mantissa %.3lf * 2 to the power %d).\n", mantissa,
exponent);
}
```

If `10.0` were entered at the terminal's keyboard, the output would be as follows:

```
The floating-point value 10.000 = (the mantissa 0.625 * 2 to the power
4).
```

Related Functions

`atof`, `fabs`, `ldexp`, `modf`

fscanf

Purpose

Using a specified format, reads a sequence of characters from the file associated with a given file pointer.

Syntax

```
#include <stdio.h>

int fscanf(FILE * file_ptr, const char *format, ...);
```

Explanation

The `fscanf` function reads input from the file pointed to by `file_ptr` and converts the input sequence of characters using the directives specified in the string pointed to by `format`. The `fscanf` function assigns each converted result to an object pointed to by the corresponding argument in the argument list. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

The string pointed to by `format` consists of zero or more directives. The directives consist of the following:

- One or more white-space characters cause `fscanf` to read up to the next non-white-space character, which remains unread, or until it can read no more characters. (A newline character is a white-space character.)
- Ordinary non-white-space characters (not `%`) cause `fscanf` to read the next characters from the file. When `fscanf` encounters a nonmatching character, the directive fails, and the nonmatching and subsequent characters remain unread.
- A *conversion specification* defines a set of input sequences that `fscanf` attempts to match.

The `fscanf` function executes each directive in turn. If a directive fails, the `fscanf` function returns. A directive can fail, for example, if no input characters are available or if the input characters do not match those specified by the directive.

Typically, each conversion specification causes `fscanf` to read an input item from the file, to convert the input item to the type specified by the conversion specifier, and to assign the converted result to the object pointed to by the corresponding argument. The following example shows an `fscanf` call consisting of a `file_ptr` argument, a `%d` conversion specification, and a corresponding argument having the type pointer to `int`.

```
int i_num;

fscanf(file_ptr, "%d", &i_num);
```

In the preceding `fscanf` call, `fscanf` executes the `%d` directive by reading characters from the file specified by `file_ptr`, converting the characters that match the `%d` format to an `int` value, and assigning the converted result to the object located by the expression `&i_num`. If the input sequence of characters were `99abcde`, `fscanf` would assign the value 99 to `i_num`.

In an `fscanf` call, the syntax for the argument list is as follows:

```
fscanf(file_ptr, format[, argument]...)
```

Each *`argument`* must be an expression yielding a pointer to an object type appropriate for the conversion specification. The number and types of the arguments in an `fscanf` call depend on the conversion specifications given in the format string.

- If the pointed-to object does not have the appropriate type, or if the result of the conversion cannot be represented in the pointed-to object, `fscanf` yields unpredictable results.
- If there are not enough arguments for the format, the behavior is unpredictable.
- If the format is exhausted while arguments remain, the excess arguments are evaluated, as are all function arguments, but are otherwise ignored.

Conversion Specifications. The format string typically includes one or more conversion specifications. Each conversion specification describes how `fscanf` is to interpret a part of the input sequence of characters. The format for a conversion specification is as follows:

```
%[*][width][size]conversion_specifier
```

For information on each element of a conversion specification, see the `scanf` function.

The following table provides a summary of the formatted input conversion specifiers. For each specifier, the table shows the corresponding argument's type and a description of the expected input item.

Conversion Specifier	**Argument Type**	**Expected Input Item**
`c`	`char *`	A single character or the number of characters specified in *`width`*. White-space characters are not skipped.
`d`	`int *`	An optionally signed decimal integer.
`e, E, f, g, G`	`float *`	An optionally signed floating-point number.
`i`	`int *`	An optionally signed integer in decimal, hexadecimal, or octal format.
`n`	`int *`	The number of characters read so far by this `fscanf` call is written into the pointed-to `int` variable.
`o`	`unsigned int *`	An optionally signed octal integer.
`p`	`void **`	An eight-digit absolute address in hexadecimal format, such as `0000000a`.
`s`	`char *`	A sequence of non-white-space characters delimited by the first white-space character. A terminating null character is appended to the sequence of characters.
`u`	`unsigned int *`	An optionally signed decimal integer.
`v`	`char_varying(`*`n`*`) *`	A varying-length character string.
`x, X`	`unsigned int *`	An optionally signed hexadecimal integer.
`%%`	none	Matches a percent sign.
`[`*`scanlist`*`]`	`char *`	A sequence of characters delimited by the first character not in the *`scanlist`*. A terminating null character is appended to the sequence of characters.

The following table provides a sample invocation of `fscanf` with each conversion specifier. The `fp` argument is a file pointer that specifies a file from which the input will be read. For each specifier, the table shows an input sequence and describes how `fscanf` stores the input item.

Function Invocation	**Argument Declaration**	**Input Sequence**	**Effect**
`fscanf(fp, "%c", &ch)`	`char ch;`	`128e2`	Stores the character `'1'` in `ch`.
`fscanf(fp, "%d", &i)`	`int i;`	`128e2`	Stores the value 128 in `i`.
`fscanf(fp, "%e", &f)`	`float f;`	`128e2`	Stores the value 12800.0 in `f`.
`fscanf(fp, "%i", &i)`	`int i;`	`128e2`	Stores the value 128 in `i`.
`fscanf(fp, "%o", &u)`	`unsigned int u;`	`128e2`	Stores the value 12 octal in `u`.
`fscanf(fp, "%p", &p)`	`void *p;`	`0000000a`	Stores the value 0000000a hex. in `p`.
`fscanf(fp, "%s", a)`	`char a[10];`	`abcd efg`	Stores the string `"abcd"` in `a`.
`fscanf(fp, "%u", &u)`	`unsigned int u;`	`128e2`	Stores the value 128 in `u`.
`fscanf(fp, "%v", &cv)`	`char_varying(5) cv;`	`abcd efg`	Stores the characters `abcd` in `cv`.
`fscanf(fp, "%X", &u)`	`unsigned int u;`	`00ff`	Stores the value ff hex. in `u`.
`fscanf(fp, "%%")`	`none`	`%`	Matches a `%` character.
`fscanf(fp, "%[12345]", &a)`	`char a[10];`	`128e2`	Stores the string `"12"` in `a`.

Return Value

The `fscanf` function returns the number of input items assigned to arguments. The number of input items assigned can be fewer than expected, or zero, depending on the number of input characters that match those specified by the format string's directives.

If end of input occurs before any successful conversion, including those with suppressed assignments, `fscanf` returns `EOF`.

Examples

The following program uses `fscanf` to read two lines of input from a file pointed to by `file_ptr`. The `fscanf` function converts each input item and assigns the result to the corresponding argument.

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

int items;
double d_num;
short s_num;

FILE *file_ptr;

main()
{
   errno = 0;
   if ((file_ptr = fopen("sample_file", "r+")) == NULL)
     {
     printf("Error opening sample_file:  errno = %d\n", errno);
     exit(1);
     }

  items = fscanf(file_ptr, "%lf", &d_num);           /*  Example 1  */
   printf("Example 1:  items = %d\n", items);
   printf("d_num = %lf\n", d_num);

  items = fscanf(file_ptr, "%3hd", &s_num);          /*  Example 2  */
   printf("\nExample 2:  items = %d\n", items);
   printf("s_num = %hd\n", s_num);

   fclose(file_ptr);
}
```

Assume that `sample_file` contains the following two lines of input.

```
987.654
1234
```

With this input, the output from the preceding program would be as follows:

```
Example 1:  items = 1
d_num = 987.654000

Example 2:  items = 1
s_num = 123
```

Related Functions

`scanf`, `sscanf`

`fseek`

Purpose

Sets the file-position indicator for a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int fseek(FILE *file_ptr, long int offset, int whence);
```

Explanation

The `fseek` function sets the file-position indicator for the file specified by `file_ptr`. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

The `offset` argument indicates how many bytes the new position is from `whence`, the starting point. A positive value indicates bytes forward. A negative value indicates bytes backward.

- For a file with stream or sequential file organization, `offset` can be any appropriate number of bytes.
- For a file with relative or fixed file organization, `offset` must be a number of bytes that is a **multiple** of the record size.

The `whence` argument indicates the starting point from where the file-positioning operation begins. The `whence` argument can be one of the values shown in the following table.

whence	Starting Point
`SEEK_SET`	beginning-of-file
`SEEK_CUR`	current position
`SEEK_END`	end-of-file

Each of the `whence` values is a constant defined in the `stdio.h` header file.

There are certain limitations when using `fseek` with a sequential file. Seeking on a sequential file is limited to a file that has been opened for reading only: either input mode or dirty-input mode. When seeking from the beginning of a sequential file, `offset` **must** be the value returned by a previous call to the `ftell` function. Seeking on a sequential file can result in inferior performance because `fseek` often must read many records to position to the correct offset.

When you open a file with stream, fixed, or relative file organization and specify an `a` as the first character in the `mode` argument of `fopen`, the file is opened in append mode. In append mode, the file-position indicator is, by default, ignored for write operations, and all output is written to the end of the file.

If you open a file by specifying an `a` character **without** a `+` in the `mode` argument of `fopen`, you can change the behavior of append mode by assigning a nonzero value to the `_TRADITIONAL_` external variable. When `_TRADITIONAL_` has been assigned a nonzero value, write operations occur at the file's current position. For more information on append mode and file-positioning operations, see the description of the `fopen` function.

A successful call to `fseek` clears the file's end-of-file indicator and undoes any effects of a previous call to `ungetc` for the same file. After a successful call to `fseek`, the next operation on a stream, fixed, or relative file that has been opened in update mode can be either input or output. (The `fseek` function cannot be used on a sequential file that has been opened in update mode.)

If an error occurs during the file-positioning operation, `fseek` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `lseek` function for the possible values that `fseek` uses when setting these variables.

Return Value

If the file-positioning operation succeeds, `fseek` returns the value 0.

If the file-positioning operation fails, `fseek` returns the value -1. For example, the operation can fail if you specify a combination of `offset` and `whence` arguments that would result in a negative current position (that is, a position before the beginning of the file).

Examples

The following program uses `fseek` to set the file-position indicator to byte 10 of a fixed file that has 10-byte records.

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

char buffer[11];
int position;

main()
{
   FILE *file_ptr;
   int num;

   errno = 0;
   if ((file_ptr = fopen("test_file", "w+f 10")) == NULL)
     {
     printf("Error opening test_file:  errno = %d\n", errno);
     exit(1);
     }
```

(Continued on next page)

```
      fputs("1111111111", file_ptr);
      fputs("2222222222", file_ptr);

      errno = 0;
      if ( (position = fseek(file_ptr, 10, SEEK_SET)) != 0)
        {
        printf("Error in seek operation:  errno = %d\n", errno);
        exit(1);
        }

      num = 11;
      fgets(buffer, num, file_ptr);
      printf("The record is %s\n", buffer);

      fclose(file_ptr);
   }
```

The output from the preceding program is as follows:

```
The record is 2222222222
```

Related Functions

`fgetpos`, `fsetpos`, `ftell`, `lseek`, `rewind`

fsetpos

Purpose

Sets the file-position indicator, for a file associated with a specified file pointer, to an earlier position obtained by the `fgetpos` function.

Syntax

```
#include <stdio.h>

int fsetpos(FILE *file_ptr, const fpos_t *pos);
```

Explanation

The `fsetpos` function sets the file-position indicator for the file specified by `file_ptr` according to the value in the variable pointed to by `pos`. That value is obtained from an earlier call to `fgetpos` specifying the value of `file_ptr`. The `pos` argument is a pointer to a variable of the type `fpos_t`, which is a type defined in the `stdio.h` header file. The `file_ptr` argument is a pointer returned, for example, by a previous call to the `fdopen`, `fopen`, or `freopen` function.

The `fgetpos` and `fsetpos` functions are used together. First, you call `fgetpos` to get and store the current position of a file. At a later time, you call `fsetpos` to return to that position.

When `fsetpos` is successfully called for a specified file, the function clears the file's end-of-file indicator and undoes any effects of the `ungetc` function on the associated stream. After an `fsetpos` call, the next operation on a file opened for update can be either input or output.

There are certain limitations when using `fgetpos` and `fsetpos` with a sequential file. Seeking on a sequential file is limited to a file that has been opened for reading only: either input mode or dirty-input mode. Seeking on a sequential file can result in inferior performance because the file-positioning function often must read many records to position to the correct offset.

If an error occurs, `fsetpos` sets the external variables `errno` and `os_errno` to the error code number returned by the operating system. See the `lseek` function for the possible values that `fsetpos` uses when setting these variables.

Return Value

If the operation succeeds, the `fsetpos` function returns the value 0.

If the operation fails, `fsetpos` returns a positive value corresponding to the operating-system error code number.

Examples

See the description of the `fgetpos` function for an example of how to use `fsetpos`.

Related Functions

`fgetpos`, `fseek`, `ftell`

ftell

Purpose

Gets the current value of the file-position indicator for a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

long ftell(FILE *file_ptr);
```

Explanation

The `ftell` function gets the current value of the file-position indicator for the file specified by `file_ptr`. The `file_ptr` argument is a pointer returned, for example, by a previous call to the `fdopen`, `fopen`, or `freopen` function.

There are certain limitations when using `ftell` with a sequential file. Seeking on a sequential file is limited to a file that has been opened for reading only: either input mode or dirty-input mode.

If an error occurs, `ftell` sets the external variables `errno` and `os_errno` to the error code number returned by the operating system. The most commonly returned error code is `e$invalid_io_operation` (`1040`).

Return Value

If the operation is successful, `ftell` returns the current value of the file-position indicator for the file specified by `file_ptr`. If the operation fails, `ftell` returns the value -1.

When `ftell` succeeds, the value that the function returns depends on the file organization of the file associated with `file_ptr`.

For a file with stream file organization, `ftell` returns the current position as the number of bytes from the beginning of the file.

For a file with relative or fixed file organization, `ftell` returns the current position as the number of bytes from the beginning of the file calculated as follows:

```
current_record_number * record_size
```

For a file with sequential file organization, `ftell` returns the current position as unspecified information that is useable by the `fseek` function for returning the file-position indicator for the specified file to its position at the time of the `ftell` call. With a sequential file, the difference between the return values of two `ftell` calls is not necessarily a meaningful measure of the number of characters written or read.

Examples

The following program uses `ftell` to get the current value of the file-position indicator for `out_file`, a file with stream organization. The program writes to the file and then returns to that position.

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

main()
{
   long int current_position;
   int c;
   FILE *file_ptr;

   if ( (file_ptr = fopen("out_file", "a+")) == NULL )
      {
      printf("Could not open out_file\n");
      exit(1);
      }

   current_position = ftell(file_ptr);          /*  Get the position
before  */
                                       /*  the write operation     */
   fprintf(file_ptr, "%s %d\n", "aaaa", 1234);

   fseek(file_ptr, current_position, SEEK_SET); /*  Return to that
position  */

   while ( (c = fgetc(file_ptr)) != '\n')
      putchar(c);
   putchar('\n');
}
```

Related Functions

`fgetpos`, `fseek`, `fsetpos`

fwrite

Purpose

Writes a specified number of data items, each of an indicated size, into a file associated with a given file pointer.

Syntax

```
#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t number, FILE
*file_ptr);
```

Explanation

The `fwrite` function writes up to `number` items of data from the array pointed to by `ptr` into the file specified by `file_ptr`. The `size` argument indicates the number of bytes in each data item to be written. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

The data is written starting at the file's current position. After `fwrite` writes data to the file, it increments the file-position indicator, if defined, by the number of bytes successfully written. The file-position indicator's resulting value is unpredictable if an error occurs.

The `fwrite` function is typically used to write data to a file that is opened in binary mode. In contrast to the formatted I/O functions such as `fprintf`, `fwrite` performs no conversions before storing the data in the array located by `ptr`.

If an error occurs, `fwrite` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `write` function for the possible values that `fwrite` uses when setting these variables.

Return Value

The `fwrite` function returns the number of data items successfully written to the file. The number of data items written will be less than `number` only if an error occurs before the specified number of items is written.

Examples

The following program fragment uses `fwrite` to write a specified number of `struct rec` size data items into a file opened in binary mode.

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

#define NUM_RECS 10

FILE *file_ptr;

struct rec
   {
   char name[30];
   int number;
   } rec_array[NUM_RECS];

size_t items_written;
int i;

main()
{
   if ( (file_ptr = fopen("binary_file", "wb")) == NULL )
      {
      puts("Error opening binary_file.");
      exit(1);
      }
      .
      .
      .
   for (i = 0; i < NUM_RECS; i++)
     if ( fwrite(&rec_array[i], sizeof(struct rec), 1, file_ptr) !=
1 )
         {
         puts("Error writing to binary_file.");
         abort();
         }
      .
      .
      .
   fclose(file_ptr);
}
```

Related Functions

`fread`, `write`

getc

Purpose

Reads and returns the next character from a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int getc(FILE *file_ptr);
```

Explanation

The `getc` macro reads the next character from the file associated with `file_ptr`. The macro then moves the file-position indicator ahead one character in the stream. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

If the file is at end-of-file, `getc` sets the end-of-file indicator associated with `file_ptr`. If a read error occurs, the macro sets the error indicator associated with `file_ptr`. You can use the `feof` and `ferror` functions to determine if end-of-file was encountered or if an error occurred.

If the file is at end-of-file or a read error occurs, `getc` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `read` function for the possible values that `getc` uses when setting these variables.

The `getc` macro can evaluate its argument more than once or not at all. Therefore, `file_ptr` should never be an expression with side effects. If a function is required, use `fgetc`.

Return Value

The `getc` macro returns an integer value representing the next character from the specified file. If a read error occurs or the file is at end-of-file, `getc` returns `EOF`.

Examples

The following program opens a file named `text_file`. It uses `getc` to read each character from the file. It uses `putchar` to display each character on the terminal's screen.

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

main()
{
   int c;
   FILE *file_ptr;

   if ( (file_ptr = fopen("text_file", "r")) == NULL )
      {
      printf("Could not open text_file.\n");
      exit(1);
      }

   while ( (c = getc(file_ptr)) != EOF)
      putchar(c);
}
```

Related Functions

`fgetc`, `fputc`, `getchar`, `putc`, `putchar`, `ungetc`

getchar

Purpose

Reads and returns the next character from the standard input file, `stdin`.

Syntax

```
#include <stdio.h>

int getchar(void);
```

Explanation

The `getchar` macro reads the next character from the preopened file pointed to by `stdin`. The file pointed to by `stdin` is usually associated with the terminal's keyboard. Notice that `getchar` is equivalent to the following:

```
getc(stdin);
```

If the file is at end-of-file, `getchar` sets the end-of-file indicator for `stdin`. If a read error occurs, the macro sets the error indicator associated with `stdin`. You can use the `feof` and `ferror` functions to determine if end-of-file was encountered or if an error occurred.

If the file is at end-of-file or a read error occurs, `getchar` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `read` function for the possible values that `getchar` uses when setting these variables.

In VOS C, `getchar` is implemented as a macro. If you need a function that performs the same operation, use `fgetc(stdin)`.

Return Value

The `getchar` macro returns an integer value representing the next character from `stdin`. If a read error occurs or the file is at end-of-file, `getchar` returns `EOF`.

Examples

The following program uses `getchar` to copy the first word from a line of input entered at the terminal's keyboard. Then, it displays the word on the terminal's screen.

```
#include <stdio.h>

int c;

main()
{
   puts("Enter some words on a line.");

   while ((c != ' ') && (c != '\t') && (c != '\n'))  /* Look for white
space */
      {
      c = getchar();
      putchar(c);
      }
   putchar('\n');
}
```

Related Functions

`fgetc`, `fputc`, `getc`, `putc`, `putchar`, `ungetc`

getenv

Purpose

Returns the `NULL` pointer constant because VOS C does not maintain a list of environment variables.

> **Note:** The `getenv` function is included in the VOS C library only for compatibility. A program that uses the function can be compiled without error.

Syntax

```
#include <stdlib.h>

char *getenv(const char *name);
```

Explanation

In VOS C, the `getenv` function always returns the `NULL` pointer constant, indicating that the specified environment name was not found in an environment list. Currently, VOS C does not maintain an environment list containing environment variables.

The `name` argument is a pointer to `char` and specifies the name of an environment variable for which, in some C implementations, `getenv` searches.

Return Value

The `getenv` function **always** returns the `NULL` pointer constant.

Examples

None

Related Functions

None

`gets`

Purpose

Reads a line from the standard input file, `stdin`.

Syntax

```
#include <stdio.h>

char *gets(char *s);
```

Explanation

Until end-of-file or a newline character is encountered, the `gets` function reads characters from the preopened file pointed to by `stdin`. The file pointed to by `stdin` is usually associated with the terminal's keyboard. The characters read are written into the array pointed to by `s`.

The `gets` function discards the newline character, if it is encountered. It then appends a null character immediately after the last character that is read into the array.

If the file is at end-of-file, `gets` sets the end-of-file indicator for `stdin`. If a read error occurs, the function sets the error indicator associated with `stdin`. You can use the `feof` and `ferror` functions to determine if end-of-file was encountered or if an error occurred.

When the file is at end-of-file or a read error occurs, `gets` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `read` function for the possible values that `gets` uses when setting these variables.

Return Value

The `gets` function returns `s`, a pointer to the array's first character, if the characters are read successfully.

If end-of-file is encountered before any characters are read into the array, `gets` returns the `NULL` pointer constant, and the contents of the array remain unchanged.

If a read error occurs, `gets` returns the `NULL` pointer constant, and the contents of the array have unpredictable values.

Examples

The following program uses gets to read a line from stdin. The program then writes the line into a file, out_file.

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

char *ptr;
char s[80];

main()
{
   FILE *file_ptr;             /*  For out_file   */

   if ( (file_ptr = fopen("out_file", "w")) == NULL )
      {
      printf("Could not open out_file.\n");
      exit(1);
      }

   puts("Enter some characters.");

   if ( (ptr = gets(s)) != NULL )
      fputs(ptr, file_ptr);
   else
      {
      puts("No characters were entered, ");
      puts("or a read error occurred in gets.");
      }
}
```

Related Functions

fgets, fputs, puts

getw

Purpose

Reads and returns the next four bytes of data from a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int getw(FILE *file_ptr);
```

Explanation

The `getw` function reads the next four bytes of data from the file specified by `file_ptr`. The function then moves the file-position indicator ahead four bytes in the file. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

The `getw` function is most often used with files that are opened in binary mode. The size of the "word" that `getw` reads is the size of an `int`, which can vary from machine to machine. In VOS C, `getw` reads four bytes, highest order byte first.

> **Note:** Because of differences in byte ordering and integer size, the data read by `getw` or written by `putw` may not be the same on an operating system other than VOS.

If `getw` returns `EOF` (a valid integer value), you can use `feof` or `ferror` to determine if the end of the file has been encountered or an error has occurred.

Return Value

The `getw` function returns the four bytes it has read as an `int` value.

If an error occurs or the file is at end-of-file, `getw` returns `EOF`.

Examples

The following program uses `getw` to read an `int` value from a file opened in binary mode.

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

FILE *file_ptr;
int number, i_num = 0xFFFFFFFF;

int ret_value;

main()
{
   errno = 0;
   if ( (file_ptr = fopen("ex_file", "r+b")) == NULL )
      {
      printf("Error opening ex_file:  errno = %d\n", errno);
      exit(1);
      }

   if ( putw(i_num, file_ptr) )
      puts("Write error occurred.");

   rewind(file_ptr);

   if ( (number = getw(file_ptr)) == EOF )
      {
      if ( feof(file_ptr) )
         puts("End of file has been encountered.");

      if ( ferror(file_ptr) )
         puts("Read error occurred.");
      }

   printf("The 4-byte value is %8X\n", number);
}
```

Related Functions

`putw`

gmtime

Purpose

Converts calendar time into broken-down time, expressed as Greenwich Mean Time.

Syntax

```
#include <time.h>

struct tm *gmtime(const time_t *timer);
```

Explanation

The `gmtime` function converts the calendar time pointed to by `timer` into broken-down time, expressed as Greenwich Mean Time (GMT). The `timer` argument is a pointer to a variable of the type `time_t`, which is a type defined in the `time.h` header file.

Broken-down time is the calendar time converted into different components, such as seconds, minutes, and hours. When the conversion is complete, broken-down time is stored in a `static` (permanent) structure of the type `struct tm`. This structure object is declared in and used by both `gmtime` and `localtime`. Subsequent calls to `gmtime` or `localtime` will overwrite the date-time values stored in this structure.

See the "Date and Time" section earlier in this chapter for information on how to use the functions found in the `time.h` header file.

Return Value

The `gmtime` function returns a pointer to the structure containing the broken-down time, expressed as Greenwich Mean Time.

If the conversion operation is not successful or Greenwich Mean Time is not available, `gmtime` returns the `NULL` pointer constant.

Examples

The following program uses `gmtime` to convert calendar time into broken-down time, expressed as Greenwich Mean Time.

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

time_t timer;
struct tm *time_ptr;
char *time_string;

main()
{
/*  Use the time function to get the calendar  */
/*  time and assign that value to timer        */

   if (time(&timer) == -1)
      puts("Error occurred:  time function call failed.");

/* Use the gmtime function to convert calendar time into broken-down
*/
/* time, which will be stored in the structure pointed to by time_ptr
*/

   if ( (time_ptr = gmtime(&timer)) == NULL )
      puts("Error occurred:  gmtime function call failed.");

/*  Use the asctime function to convert broken-down time in the
structure   */
/* pointed to by time_ptr into the date-time string
*/

   time_string = asctime(time_ptr);

   printf("The Greenwich Mean Time is %s", time_string);
}
```

The output from the preceding program might be as follows:

```
The Greenwich Mean Time is Thu Aug 02 19:40:24 1999
```

Related Functions

`asctime`, `ctime`, `localtime`, `time`

isalnum

Purpose

Tests whether an ASCII character is a letter or digit.

Syntax

```
#include <ctype.h>

int isalnum(int c);
```

Explanation

The `isalnum` function determines whether the integer argument, `c`, is the ASCII code for one of the lowercase letters `a` through `z`, one of the uppercase letters `A` through `Z`, or one of the decimal digits `0` through `9`. The `c` argument must be an integer value in the range 0 through 255, or the constant `EOF`.

Return Value

If `c` is the ASCII code for a letter or digit, the `isalnum` function returns a nonzero value. Otherwise, `isalnum` returns the value 0.

Examples

The following program uses `isalnum` and the other character-handling functions to classify ASCII codes that the user enters at the terminal's keyboard.

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

void classify(int c);

main()
{
   int code;

   printf("Enter the decimal ASCII code for a character.\n");
   printf("To exit, enter 999.\n");
```

(Continued on next page)

```
while ( scanf("%d", &code) == 1 )
      {
      if ( (code >= 0) && (code <= 255) )
         classify(code);
      else
         if (code == 999)
            exit(1);
         else
            printf("\nValue entered was not in the range 0 through
255.\n");

      printf("\nEnter another decimal ASCII code.\n");
      printf("To exit, enter 999.\n");
      }
}

void classify(int c)
{
   printf("\nThe character represented by the ASCII code %d\n", c);
   printf("is classified as:\n\n");

   if ( isalnum(c) != 0 )
      printf("  - a letter or digit\n");

   if ( isalpha(c) != 0 )
      printf("  - a letter\n");

   if ( iscntrl(c) != 0 )
      printf("  - a control character\n");

   if ( isdigit(c) != 0 )
      printf("  - a decimal digit\n");

   if ( isgraph(c) != 0 )
      printf("  - a printable character, but not a space\n");

   if ( islower(c) != 0 )
      printf("  - a lowercase letter\n");

   if ( isoctal(c) == 1 )         /*  isoctal returns 1 if the test
is true  */
      printf("  - an octal digit\n");

   if ( isprint(c) != 0 )
      printf("  - a printable character, possibly a space\n");


   if ( ispunct(c) != 0 )
      printf("  - a punctuation character\n");
```

(Continued on next page)

```
         if ( isspace(c) != 0 )
            printf("  - a white-space character\n");

         if ( isupper(c) != 0 )
            printf("  - an uppercase character\n");

         if ( isxdigit(c) != 0 )
            printf("  - a hexadecimal digit\n");
      }
```

Related Functions

`isalpha`, `isascii`, `isdigit`, `isoctal`, `isxdigit`

isalpha

Purpose

Tests whether an ASCII character is a letter.

Syntax

```
#include <ctype.h>

int isalpha(int c);
```

Explanation

The `isalpha` function determines whether the integer argument, `c`, is the ASCII code for one of the lowercase letters `a` through `z` or one of the uppercase letters `A` through `Z`. The `c` argument must be an integer value in the range 0 through 255, or the constant `EOF`.

Return Value

If `c` is the ASCII code for a letter, the `isalpha` function returns a nonzero value. Otherwise, `isalpha` returns the value 0.

Examples

See the description of the `isalnum` function for an example of how to use `isalpha`.

Related Functions

`isalnum`, `isascii`, `islower`, `isupper`

isascii

Purpose

Tests whether an integer value is a valid ASCII character.

Syntax

```
#include <ctype.h>

int isascii(int c);
```

Explanation

The `isascii` function determines whether the integer argument, `c`, is the code for a valid ASCII character. The codes for valid ASCII characters are in the range 0 through 127. The `c` argument can be any integer value, including the constant `EOF`.

Return Value

If `c` is the code for a valid ASCII character, the `isascii` function returns a nonzero value. Otherwise, `isascii` returns the value 0.

Examples

The following program uses `isascii` to test whether an integer value, entered at the terminal's keyboard, is a valid ASCII character.

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

main()
{
   int i;

   printf("Enter an integer value.\n");
   printf("To exit, enter 999.\n");

   while ( scanf("%d", &i) == 1 )
      {
      if (i == 999)
          exit(1);
```

(Continued on next page)

```
      if ( isascii(i) != 0 )
        printf("The value %d represents a valid ASCII character.\n",
i);
      else
         printf("The value %d does NOT represent a valid ASCII
character.\n",
                    i);

      printf("\n");
      printf("Enter another integer value.\n");
      printf("To exit, enter 999.\n");
      }
}
```

Related Functions

isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, toascii

isatty

Purpose

Tests whether a specified file descriptor is associated with a terminal device.

Syntax

```
#include <c_utilities.h>

int isatty(int file_des);
```

Explanation

The `isatty` function determines whether a file descriptor, `file_des`, is a terminal device. The `file_des` argument is an integer file descriptor returned by the `creat`, `fileno`, or `open` function.

Return Value

If `file_des` is associated with a terminal device, the `isatty` function returns the value 1. Otherwise, `isatty` returns the value 0.

Examples

The following program uses `isatty` to determine if the file descriptor `1` is currently associated with a terminal device. The file descriptor `1` is normally associated with `stdout`.

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

main()
{
   int file_des = 1;

   if (isatty(file_des) == 1)
      printf("stdout is associated with a terminal.\n");
   else
      printf("stdout is not associated with a terminal.\n");
}
```

Related Functions

`open`

iscntrl

Purpose

Tests whether an ASCII character is a control character.

Syntax

```
#include <ctype.h>

int iscntrl(int c);
```

Explanation

The `iscntrl` function determines whether the integer argument, `c`, is the ASCII code for a control character. The `c` argument must be an integer value in the range 0 through 255, or the constant `EOF`.

A *control character* is any character that cannot be printed. In VOS C, the control characters consist of the ASCII characters with codes in the range 0 through 31, and the code 127.

Return Value

If `c` is the ASCII code for a control character, the `iscntrl` function returns a nonzero value. Otherwise, `iscntrl` returns the value 0.

Examples

See the description of the `isalnum` function for an example of how to use `iscntrl`.

Related Functions

`isgraph`, `isprint`, `ispunct`, `isspace`

isdigit

Purpose

Tests whether an ASCII character is a decimal digit.

Syntax

```
#include <ctype.h>

int isdigit(int c);
```

Explanation

The `isdigit` function determines whether the integer argument, `c`, is the ASCII code for one of the decimal digits `0` through `9`. The `c` argument must be an integer value in the range 0 through 255, or the constant `EOF`.

Return Value

If `c` is the ASCII code for a decimal digit, the `isdigit` function returns a nonzero value. Otherwise, `isdigit` returns the value 0.

Examples

See the description of the `isalnum` function for an example of how to use `isdigit`.

Related Functions

`isalnum`, `isoctal`, `isxdigit`

isgraph

Purpose

Tests whether an ASCII character is a printable character other than a space.

Syntax

```
#include <ctype.h>

int isgraph(int c);
```

Explanation

The `isgraph` function determines whether the integer argument, `c`, is the ASCII code for one of the printable characters excluding the space character. The `c` argument must be an integer value in the range 0 through 255, or the constant `EOF`.

The `isgraph` function classifies ASCII codes in the range 33 through 126 as representing the printable characters. The `isgraph` function does **not** classify the space character (32 decimal) as a printable character.

Return Value

If `c` is the ASCII code for a printable character other than a space, the `isgraph` function returns a nonzero value. Otherwise, `isgraph` returns the value 0.

Examples

See the description of the `isalnum` function for an example of how to use `isgraph`.

Related Functions

`iscntrl`, `isprint`, `ispunct`, `isspace`

`islower`

Purpose

Tests whether an ASCII character is a lowercase letter.

Syntax

```
#include <ctype.h>

int islower(int c);
```

Explanation

The `islower` function determines whether the integer argument, `c`, is the ASCII code for one of the lowercase letters `a` through `z`. The `c` argument must be an integer value in the range 0 through 255, or the constant `EOF`.

Return Value

If `c` is the ASCII code for a lowercase letter, the `islower` function returns a nonzero value. Otherwise, `islower` returns the value 0.

Examples

See the description of the `isalnum` function for an example of how to use `islower`.

Related Functions

`isalnum`, `isalpha`, `isupper`, `toupper`

isoctal

Purpose

Tests whether an ASCII character is an octal digit.

Syntax

```
#include <ctype.h>

int isoctal(int c);
```

Explanation

The `isoctal` function determines whether the integer argument, `c`, is the ASCII code for one of the octal digits `0` through `7`. The `c` argument must be an integer value in the range 0 through 255, or the constant `EOF`.

Return Value

If `c` is the ASCII code for an octal digit, the `isoctal` function returns the value 1. Otherwise, `isoctal` returns the value 0.

> **Note:** The return value of `isoctal` is either 1 or 0. Other character-testing functions return a nonzero value (not necessarily the value 1) if the test is true.

Examples

See the description of the `isalnum` function for an example of how to use `isoctal`.

Related Functions

`isdigit`, `isxdigit`

isprint

Purpose

Tests whether an ASCII character is a printable character, including the space character.

Syntax

```
#include <ctype.h>

int isprint(int c);
```

Explanation

The `isprint` function determines whether the integer argument, `c`, is the ASCII code for one of the printable characters, including the space character. The `c` argument must be an integer value in the range 0 through 255, or the constant `EOF`.

The `isprint` function classifies ASCII codes in the range 32 through 126 as representing the printable characters. The `isprint` function classifies the space character (32 decimal) as a printable character.

Return Value

If `c` is the ASCII code for a printable character including the space character, the `isprint` function returns a nonzero value. Otherwise, `isprint` returns the value 0.

Examples

See the description of the `isalnum` function for an example of how to use `isprint`.

Related Functions

`iscntrl`, `isgraph`, `ispunct`, `isspace`

ispunct

Purpose

Tests whether an ASCII character is a punctuation character.

Syntax

```
#include <ctype.h>

int ispunct(int c);
```

Explanation

The `ispunct` function determines whether the integer argument, `c`, is the ASCII code for a punctuation character. The `c` argument must be an integer value in the range 0 through 255, or the constant `EOF`.

A punctuation character is any printable character except a space, a digit, or a letter.

Return Value

If `c` is the ASCII code for a punctuation character, the `ispunct` function returns a nonzero value. Otherwise, `ispunct` returns the value 0.

Examples

See the description of the `isalnum` function for an example of how to use `ispunct`.

Related Functions

`iscntrl`, `isgraph`, `isprint`, `isspace`

isspace

Purpose

Tests whether an ASCII character is a white-space character.

Syntax

```
#include <ctype.h>

int isspace(int c);
```

Explanation

The `isspace` function determines whether the integer argument, `c`, is the ASCII code for one of the white-space characters. The `c` argument must be an integer value in the range 0 through 255, or the constant `EOF`.

The *white-space characters* consist of the characters shown in the following table:

White-Space Character	ASCII Code
horizontal tab	9
newline	10
vertical tab	11
form feed	12
carriage return	13
space	20

Return Value

If `c` is the ASCII code for a white-space character, the `isspace` function returns a nonzero value. Otherwise, `isspace` returns the value 0.

Examples

See the description of the `isalnum` function for an example of how to use `isspace`.

Related Functions

`iscntrl`, `isgraph`, `isprint`, `ispunct`

isupper

Purpose

Tests whether an ASCII character is an uppercase letter.

Syntax

```
#include <ctype.h>

int isupper(int c);
```

Explanation

The `isupper` function determines whether the integer argument, `c`, is the ASCII code for one of the uppercase letters `A` through `Z`. The `c` argument must be an integer value in the range 0 through 255, or the constant `EOF`.

Return Value

If `c` is the ASCII code for an uppercase letter, the `isupper` function returns a nonzero value. Otherwise, `isupper` returns the value 0.

Examples

See the description of the `isalnum` function for an example of how to use `isupper`.

Related Functions

`isalnum`, `isalpha`, `islower`, `tolower`

isxdigit

Purpose

Tests whether an ASCII character is a hexadecimal digit.

Syntax

```
#include <ctype.h>

int isxdigit(int c);
```

Explanation

The `isxdigit` function determines whether the integer argument, `c`, is the ASCII code for one of the hexadecimal digits: `0` through `9`, `a` through `f`, and `A` through `F`. The `c` argument must be an integer value in the range 0 through 255, or the constant `EOF`.

Return Value

If `c` is the ASCII code for a hexadecimal digit, the `isxdigit` function returns a nonzero value. Otherwise, `isxdigit` returns the value 0.

Examples

See the description of the `isalnum` function for an example of how to use `isxdigit`.

Related Functions

`isdigit`, `isoctal`

`kill`

Purpose

Sends a specified signal to the calling program.

Syntax

```
#include <signal.h>

int kill(int process_id, int sig);
```

Explanation

The `kill` function sends the signal `sig` to the calling program. The `process_id` argument is an integer value specifying the calling program's process identifier. **In VOS C, the** `process_id` argument must be 0.

Signal numbers range from 1 through 8. The `signal.h` header file contains a defined constant (macro) for each value. The following table lists each constant and the corresponding value and exception condition.

Value	**Signal**	**Exception Condition**
1	`SIGABRT`	Abnormal termination of the program, as is signaled by the `abort` function. Also, the `SIGABRT` signal is sent for many operating system errors.
2	`SIGFPE`	Floating-point exception (computational condition): the VOS zero divide, floating-point underflow, and floating-point overflow errors.
3	`SIGILL`	Illegal instruction.
4	`SIGINT`	Interrupt (break condition), which is signaled by pressing the terminal's CTRL and BREAK keys simultaneously.
5	`SIGSEGV`	Segmentation violation (illegal memory access).
6	`SIGTERM`	Program termination, which is signaled only by the program through the function invocation `kill(0, SIGTERM)`.
7	`SIGUSR1`	User-defined signal 1.
8	`SIGUSR2`	User-defined signal 2.

The `kill` function signals the exception condition corresponding to `sig`. When a signal is raised by a call to `kill(0, `*`sig`*`)`, the function passes the error code 0 to the signal-handling function, if one is set up.

You can use the `signal` function to specify how a particular signal is handled. See the `signal` function for information on setting up signal-handling functions.

If the signal `sig` is out of range, the `kill` function sets the external variable `errno` to `e$invalid_arg` (`1371`). If `process_id` is a value other than 0, the function sets `errno` to `e$bad_process_id` (`1210`).

See the "Signal Handling" section earlier in this chapter for more information on signal handling and the signals defined in the `signal.h` header file.

Return Value

If `kill` successfully sends the signal, the function returns the value 0. Otherwise, `kill` returns a nonzero value.

Examples

The following program uses `kill` to send the `SIGUSR1` signal. In the example program, `kill` raises the signal when the user enters a divisor of 0 **prior to** a division operation.

Notice that the signal-handling function `if_sigusr1` does not use `signal` to "reset" the handling for `SIGUSR1`. That is, the `if_sigusr1` function does not contain a call `signal(SIGUSR1, if_sigusr1)`. Therefore, the `if_sigusr1` function is invoked only the first time that a user enters a divisor of 0. The second time a divisor of 0 is entered, the default handler for the floating-point error signal (`SIGFPE`) is invoked when the division operation is performed.

```
#include <signal.h>
#include <stdio.h>
#include <c_utilities.h>
#include <stdlib.h>

int if_sigusr1(int error_code);
int dividend, divisor, result;

main()
{
   if (signal(SIGUSR1, if_sigusr1) == SIG_ERR)
      puts("Default handler for SIGUSR1 still in effect.");

   puts("Enter the dividend.");
   scanf("%d", &dividend);

get_divisor:
   puts("Enter the divisor.");
   scanf("%d", &divisor);
```

(Continued on next page)

```
   if (divisor == 0)
      {
      if ( kill(0, SIGUSR1) != 0 )       /*  Send SIGUSR1 signal  */
         {
         puts("Error sending signal SIGUSR1.");
         exit(1);
         }
      goto get_divisor;
      }

   result = dividend/divisor;
   printf("The result is %d.\n", result);
}

                             /*  Signal handler for the following:  */
int if_sigusr1(int error_code)    /*  before a divide, user entered
*/
                               /*  a divisor of 0                   */
   {
   puts("\nYou entered a divisor of 0.  Try again.");
   }
```

Related Functions

`abort`, `signal`

ldexp

Purpose

Computes a floating-point value by multiplying a specified mantissa by a specified power of 2.

Syntax

```
#include <math.h>

double ldexp(double x, int exp);
```

Explanation

The `ldexp` function calculates a floating-point value by multiplying a mantissa, `x`, by an integer power of 2, `exp`. The formula used to compute the floating-point value is as follows:

`floating_point_value = x * 2`$^{|exp|}$

The `x` argument, a `double` value, contains the mantissa. The `exp` argument, an `int` value, is the power to which 2 will be raised. See Appendix A for more information on the format of floating-point values.

If the resulting floating-point value cannot be represented by a `double` or the specified exponent is too large or too small, `ldexp` sets `errno` to `ERANGE`.

Return Value

The `ldexp` function returns a floating-point value, using the procedure described in the Explanation.

When the result cannot be represented by a `double` or when the specified exponent would create an overflow condition, `ldexp` returns positive or negative `HUGE_VAL`, depending on the sign of `x`. When the specified exponent would create an underflow condition, `ldexp` returns 0.0.

Examples

The following program uses `ldexp` to calculate a floating-point value from a mantissa and exponent entered at the terminal's keyboard.

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

main()
{
   double x, d_num;
   int exp;

   printf("Enter a mantissa.\n");
   scanf("%lf", &x);
   printf("Enter an exponent.\n");
   scanf("%d", &exp);

   d_num = ldexp(x, exp);

   printf("(The mantissa %.3lf * 2 to the power %d)", x, exp);
   printf(" = the floating-point value %.3lf.\n", d_num);
}
```

If two values, `0.625` for `x` and `4` for `exp`, were entered at the terminal's keyboard, the output would be as follows:

```
(The mantissa 0.625 * 2 to the power 4) = the floating-point
value 10.000.
```

Related Functions

`atof`, `fabs`, `frexp`, `modf`

`localtime`

Purpose

Converts calendar time into broken-down time, expressed as local time.

Syntax

```
#include <time.h>

struct tm *localtime(const time_t *timer);
```

Explanation

The `localtime` function converts the calendar time pointed to by `timer` into broken-down time, expressed as local time. In VOS C, *local time* is the calendar time expressed using the current time zone defined for the calling process. The `timer` argument is a pointer to a variable of the type `time_t`, which is a type defined in the `time.h` header file.

Broken-down time is the calendar time converted into different components, such as seconds, minutes, and hours. When the conversion is complete, broken-down time is stored in a `static` (permanent) structure of the type `struct tm`. This structure object is declared in and used by both `gmtime` and `localtime`. Subsequent calls to `gmtime` or `localtime` will overwrite the date-time values stored in this structure.

See the "Date and Time" section earlier in this chapter for information on how to use the functions found in the `time.h` header file.

Return Value

The `localtime` function returns a pointer to the structure containing the broken-down time, expressed as local time.

If the conversion operation is not successful, `localtime` returns the `NULL` pointer constant.

Examples

See the description of the `asctime` function for an example of how to use `localtime`.

Related Functions

`asctime`, `ctime`, `gmtime`, `time`

`lockf`

Purpose

Performs various region-locking actions on a stream file associated with a specified file descriptor.

Syntax

```
#include <c_utilities.h>

int lockf(int file_des, int command, long size);
```

Explanation

For a file specified by `file_des`, the `lockf` function performs the region-locking action given in `command` on the region of the file given in `size`. The file must have stream file organization and be open in region-locking lock mode. The `file_des` argument is a file descriptor returned by a previous call to `creat`, `fileno`, or `open`.

The `lockf` function performs the region-locking action on the region of the file starting at the current position in the file and continuing for the number of contiguous bytes specified in `size`.

- If `size` is a positive value, `lockf` operates on the region of the file beginning with the current position and ending with the byte that is `size` bytes minus 1 after the current position. The region can extend beyond the current end of the file. In this case, the lock encompasses anything added to the file until the file is extended beyond the specified region.
- If `size` is a negative value, `lockf` operates on the region of the file beginning `size` bytes before the current position and ending with the byte before the current position. The region **cannot** extend back beyond the beginning of the file.
- If `size` is the value 0, `lockf` operates on the region from the current position in the file to the maximum possible file size. This region automatically encompasses anything added to the end of the file if the file is extended.

The `command` argument specifies the action to perform on the given region of the file. The allowed values for `command` are constants defined in the `c_utilities.h` header file. The `command` argument can have one of the values shown in the following table.

Command	**Description**
`F_TEST`	Tests the region to determine if it is locked by another process.
`F_LOCK`	Attempts to lock the region for writing. Causes the program's process to sleep until the lock becomes available.
`F_TLOCK`	Attempts to lock the region for writing. Returns if the lock is not immediately available.
`F_RLOCK`	Attempts to lock the region for reading. Causes the program's process to sleep until the lock becomes available.
`F_TRLOCK`	Attempts to lock the region for reading. Returns if the lock is not immediately available.
`F_ULOCK`	Depending on the value of `size`, unlocks the calling program's lock on all or part of a region that the program has previously locked.

Multiple processes can have read locks on a region at the same time, but only one process can have a write lock on a region. It is not possible to get a read lock on a region that another process has locked for writing, nor is it possible to get a write lock on a region that another process has locked for reading.

The calling program can change a read lock to a write lock on a region if no other process has a lock on the region. If another process is waiting for a write lock, it continues to wait. The calling program can change a write lock to a read lock without restriction.

If an error occurs in the lock testing, locking, or unlocking operation, `lockf` sets the external variables `errno` and `os_errno` to the error code returned by the operating system. For example, when all or part of the region is locked by another process, `lockf` sets these variables to `e$region_in_use` (`3543`). The following table lists some other error code values that are possible.

Error Code Name	**Number**
`e$invalid_file_pointer`	`3929`
`e$invalid_io_operation`	`1040`
`e$lock_out_of_range`	`3547`
`e$not_locked_to_this_process`	`1035`
`e$port_not_attached`	`1021`

Return Value

For a locking or unlocking operation, the `lockf` function returns the value 0 if the operation is successful. If `lockf` is not successful in the locking or unlocking operation, the function returns the value -1.

For the lock testing operation (`F_TEST`), if the region is not locked by another process, `lockf` returns the value 0. If the region is locked by another process, `lockf` returns the value -1.

Examples

The following program fragment uses `lockf` to attempt to get a write lock on the first 4096 bytes of a stream file.

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

FILE *file_ptr;

main()
{
   int file_des;
   long int size;

   errno = 0;
   if ( (file_ptr = fopen("stream_file", "r+g") ) == NULL )
      {
      printf("Error opening stream_file:  errno = %d\n", errno);
      exit(1);
      }

   file_des = fileno(file_ptr);

   size = 4096;

   errno = 0;
   if ( lockf(file_des, F_TLOCK, size) == -1 )
     printf("Error occurred in lockf call:  errno = %d\n", errno);
      .
      .
      .
   close(file_des);
}
```

The `lockf` function requires that the specified file have stream file organization (the default) and be opened in region-locking locking mode. The `g` in the second argument to `fopen` specifies that `stream_file` is to be opened in region-locking locking mode.

Related Functions

`close`, `creat`, `open`

log

Purpose

Computes the natural logarithm of a floating-point value.

Syntax

```
#include <math.h>

double log(double x);
```

Explanation

The `log` function calculates the natural logarithm of `x`. The `x` argument is a value of the type `double`. The natural logarithm of `x` is the power to which `e`, the base, would have to be raised to produce `x`. The value `e` is approximately 2.718, the base of the natural logarithm.

If the value of `x` is not positive, a domain error occurs, and `log` sets `errno` to `EDOM`.

Return Value

The `log` function returns the natural logarithm of `x`. When the value of `x` is not positive, `log` returns `-HUGE_VAL`.

Examples

The following program uses `log` to calculate the natural logarithm of a floating-point value entered at the terminal's keyboard.

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

main()
{
   double x, logarithm;

   printf("Enter a number for which you want to find the
logarithm.\n");
   scanf("%le", &x);

   errno = 0;
   logarithm = log(x);
```

(Continued on next page)

```
      if (errno == EDOM)
         puts("A domain error occurred.");
      else
         printf("The logarithm of %lE = %lE.\n", x, logarithm);
   }
```

If `12.0` were entered at the terminal's keyboard, the output would be as follows:

```
   The logarithm of 1.200000E+01 = 2.484907E+00.
```

Related Functions

`exp`, `log10`, `pow`

log10

Purpose

Computes the base-10 logarithm of a floating-point value.

Syntax

```
#include <math.h>

double log10(double x);
```

Explanation

The `log10` function calculates the base-10 logarithm of `x`. The `x` argument is a value of the type `double`. The base-10 logarithm of `x` is the power to which 10 would have to be raised to produce `x`.

If the value of `x` is not positive, a domain error occurs, and `log10` sets `errno` to `EDOM`.

Return Value

The `log10` function returns the base-10 logarithm of `x`. When the value of `x` is not positive, `log10` returns `-HUGE_VAL`.

Examples

The following program uses `log10` to calculate the base-10 logarithm of a floating-point value entered at the terminal's keyboard.

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

main()
{
   double x, b10_log;

   printf("Enter a number for which you want to ");
   printf("find the base-10 logarithm.\n");
   scanf("%le", &x);

   errno = 0;
   b10_log = log10(x);
```

(Continued on next page)

```
      if (errno == EDOM)
         puts("A domain error occurred.");
      else
         printf("The base-10 logarithm of %lE = %lE.\n", x, b10_log);
   }
```

If `100.0` were entered at the terminal's keyboard, the output would be as follows:

```
The base-10 logarithm of 1.000000E+02 = 2.000000E+00.
```

Related Functions

`exp`, `log`, `pow`

longjmp

Purpose

Restores the calling environment saved in the program's most recent invocation of the `setjmp` function.

Syntax

```
#include <setjmp.h>

void longjmp(jmp_buf env, int val);
```

Explanation

The `longjmp` function restores the calling environment that was saved by the most recent invocation of the `setjmp` function with the corresponding `env` argument. The `env` argument, which stores the saved environment, is specified in both the `setjmp` and `longjmp` calls. The `env` argument is of the type `jmp_buf`, which is a type defined in the `setjmp.h` header file.

The `setjmp` and `longjmp` functions are used together to achieve the effect of a nonlocal `goto` statement. First, you call `setjmp` to save the calling environment at the point where `setjmp` was invoked. Second, in some other function, you call `longjmp` to restore that calling environment saved in `env` and to jump to the return address also saved in `env`. In effect, program control is transferred back to the location of the most recent invocation of `setjmp`. When `longjmp` returns, it acts like a return from a call to `setjmp`.

When `longjmp` causes a jump to the return address given in `env`, program execution continues as if `setjmp` had just returned with the return value given in `val`. The `val` argument specifies the value that is returned by `setjmp`. The `longjmp` function cannot cause `setjmp` to return zero. If `val` equals 0, the corresponding invocation of `setjmp` returns the value 1.

The environment saved in `env` must be active. The behavior of `longjmp` is unpredictable if the function containing the `setjmp` call has, in the interim, terminated (as opposed to suspended) execution. For example, the function where `setjmp` appears can terminate execution because it has returned to its caller or because another `longjmp` call has caused a transfer to a `setjmp` invocation in a function earlier in the set of nested calls.

If there has been no previous call to the `setjmp` function, the behavior of `longjmp` is unpredictable.

All accessible objects have values as of the time `longjmp` was called with the following exception. The value of an object having automatic storage duration is **indeterminate** if all of the following conditions exist.

- The object is local to the function containing the invocation of the corresponding `setjmp` call.
- The object does not have `volatile`-qualified type.
- The object has been changed between the `setjmp` call and the `longjmp` call.

The `longjmp` function bypasses the usual function call and return mechanisms and effectively "unwinds" the stack back to the frame associated with the function that saved the environment. While doing this, the `longjmp` function invokes any cleanup handler associated with each active function, starting with the function that called `longjmp` and continuing to, but not including, the function that saved the environment.

The `longjmp` function executes correctly in contexts of interrupts, signals, and any of their associated functions.

However, if the `longjmp` function is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is unpredictable.

Return Value

After `longjmp` is completed, program execution continues as if the corresponding invocation of `setjmp` has just returned the value specified by `val`, or has just returned the value 1 if `val` equals 0.

Examples

See the description of the `setjmp` function for an example of how to use `longjmp`.

Related Functions

`setjmp`

`lseek`

Purpose

Sets the file-position indicator on a file associated with a specified file descriptor.

Syntax

```
#include <c_utilities.h>

int lseek(int file_des, long offset, int whence);
```

Explanation

The `lseek` function sets the file-position indicator for the next input or output operation on the file associated with `file_des` to the position specified by `offset` and `whence`. The `file_des` argument is an integer file descriptor returned by the `creat`, `fileno`, or `open` function.

The `offset` argument specifies how many bytes the new position is from `whence`, the starting point. A positive offset indicates bytes forward. A negative offset indicates bytes backward.

- For a file with stream or sequential file organization, `offset` can be any appropriate number of bytes.
- For a file with relative or fixed file organization, `offset` must be a number of bytes that is a **multiple** of the record size.

The `whence` argument specifies one of the following starting points:

whence	**Starting Point**
0	beginning-of-file
1	current position
2	end-of-file

There are certain limitations when using `lseek` with a sequential file. Seeking on a sequential file is limited to a file that has been opened for reading only: either input mode or dirty-input mode. When seeking from the beginning of a sequential file, `offset` **must** be the value returned by a previous call to the `ftell` function, a standard I/O function. Seeking on a sequential file can result in inferior performance because `lseek` often must read many records to position to the correct offset.

When you open a file with stream, fixed, or relative file organization and specify `O_APPEND` in the `o_flag` argument of `open`, the file is opened in append mode. In append mode, the file-position indicator is, by default, ignored for write operations, and all output is written to the end of the file.

A successful call to `lseek` clears the file's end-of-file indicator and undoes any effects of a previous call to `ungetc` for the same file. After a successful call to `lseek`, the next operation on a stream, fixed, or relative file that has been opened in update mode can be either input or output. (The `lseek` function cannot be used on a sequential file that has been opened in update mode.)

If an error occurs during the file-positioning operation, `lseek` sets the external variables `errno` and `os_errno` to the appropriate error code number. The following table lists some possible error codes.

Error Code Name	Number
`e$beginning_of_file`	`1773`
`e$end_of_file`	`1025`
`e$invalid_file_pointer`	`3929`
`e$invalid_io_operation`	`1040`

Return Value

If the file-positioning operation is successful, the `lseek` function returns the file's new position as a non-negative integer.

If the file-positioning operation is not successful, the function returns the value -1. The following conditions, among others, cause the seek operation to fail:

- if you specify an undefined file descriptor
- if you specify a combination of `offset` and `whence` arguments that would result in a negative current position (that is, a position before the beginning of the file)
- if the file was opened using the `O_RDONLY` flag (input only) and you attempt to seek beyond the end of the file.

Examples

The following program fragment contains two examples of how `lseek` is used to set the file-position indicator for a file.

```
#include <c_utilities.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

char buffer[20];
```

(Continued on next page)

```
int position, current_position;
int file_des;

main()
{
   errno = 0;
   if ( (file_des = open("file1", O_RDONLY)) == -1 )
      {
      printf("Error opening file1:  errno = %d\n", errno);
      exit(1);
      }

   read(file_des, buffer, 5);

   errno = 0;
  if ( (position = lseek(file_des, 0, 0)) == -1 )         /*  Example
1  */
     {
     printf("Error in seek operation: errno = %d\n", errno);
     exit(1);
     }

   read(file_des, buffer, 10);

  current_position = lseek(file_des, 0, 1);              /*  Example
2  */
      .
      .
      .
   if ( (close(file_des)) == -1 )
      {
      printf("Error closing file1\n");
      exit (1);
      }
}
```

In example 1, the `lseek` function rewinds the file, setting the file-position indicator for `file1` to the beginning of the file. The example also tests to determine whether the file-positioning operation was successful.

In example 2, `lseek` finds the current position within `file1`. The value assigned to `current_position` can later be used to seek to this exact position in the file.

Related Functions

`fgetpos`, `fseek`, `fsetpos`, `ftell`, `open`, `rewind`,

`malloc`

Purpose

Allocates memory for an object of a specified size.

Syntax

```
#include <stdlib.h>

void *malloc(size_t size);
```

Explanation

The `malloc` function allocates memory for an object of the size specified in `size`. The `size` argument is a value of the type `size_t`, which is a type definition declared in the `stddef.h` header file. The number of bytes specified in `size` should include any alignment padding that would separate the same objects in an array. You can use the `sizeof` operator to ensure that this padding is included in `size`.

If `malloc` is successful, it returns a pointer to the first byte of allocated memory. This pointer is a "generic pointer" declared as a "pointer to `void`." A pointer to `void` can be assigned, without a cast, to a pointer to any object type. Nevertheless, many programmers cast the return type of `malloc` so that the type of the returned pointer is explicitly converted to the appropriate data type. See the "Pointers to Void" section in Chapter 4 for more information on that pointer type.

The `malloc` function allows you to allocate storage dynamically at run time. The amount of memory a program uses can be determined by run-time events, such as user input or the amount of data in a file.

The only way that you can access the contents of the allocated memory is indirectly through the pointer returned by `malloc`. Therefore, assign the returned pointer to a pointer variable of the appropriate type. Then, you can specify the pointer variable itself when the object's address is needed, or you can use an indirection operation when the contents at that address are needed.

The contents of the memory allocated by `malloc` have unpredictable initial values. If you want the allocated memory to be initialized to binary 0, use the `calloc` function. Alternatively, you can use the `memset` function to write the value 0 or any other value into each byte of allocated memory.

The starting address of the memory allocated with `malloc` begins on a boundary that is suitably aligned for all Stratus processors.

Return Value

If `malloc` successfully allocates memory of the specified size, the function returns a pointer to the first byte of allocated memory.

If `malloc` does not successfully allocate memory of the specified size, the function returns the `NULL` pointer constant. The `malloc` function can fail, for example, if the amount of memory specified in `size` is not free.

If `size` equals 0, `malloc` returns a valid pointer whose pointed-to storage should not be accessed.

Examples

The following program uses `malloc` to allocate memory for a structure. In the call to `malloc`, the `sizeof(struct item)` operation specifies how many bytes of memory to allocate. After memory is allocated, each structure member can be accessed through the `ptr` pointer variable. For example, you can access the structure's `name` member using `ptr->name`.

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

struct item
   {
   char name[100];
   double amount;
   };

struct item *ptr;

main()
{
   ptr = (struct item *) malloc( sizeof(struct item) );

   if (ptr == NULL)
      {
      puts("Memory was not successfully allocated.");
      exit(1);
      }
   else
      puts("Memory was successfully allocated.");
}
```

Related Functions

`alloca`, `calloc`, `free`, `realloc`

`memccpy`

Purpose

Copies bytes from one memory area into another memory area until a specified byte is encountered in the source memory area or until a specified number of bytes are copied, whichever occurs first.

Syntax

```
#include <string.h>

void *memccpy(void *mem1, const void *mem2, int c, size_t n);
```

Explanation

The `memccpy` function copies bytes from the memory area pointed to by `mem2` into the memory area pointed to by `mem1`. The `mem1` and `mem2` arguments specify the beginning address of separate memory areas. The copying starts with the first byte of `mem2` and continues until one of the following occurs.

- `memccpy` encounters the byte specified by `c` in the memory area pointed to by `mem2`. The byte specified by `c` is the last byte copied into the memory area pointed to by `mem1`.
- `memccpy` copies the number of bytes specified in `n`. Because `c` was not encountered, the byte specified by `c` is **not** copied into the memory area pointed to by `mem1`.

The `n` argument is a value of the type `size_t`, which is a type definition declared in the `stddef.h` header file.

If some parts of the memory areas specified by `mem1` and `mem2` overlap, the `memccpy` function can produce **incorrect** results. When the copying takes place between two memory areas that overlap, use the `memmove` function, which is guaranteed to produce correct results.

Return Value

The `memccpy` function returns a pointer to `mem1`, the memory area to which the bytes were copied.

Examples

The following program uses `memccpy` to copy bytes from one memory area into another memory area until a byte having the value of the character `X` is encountered in the source memory area or until five bytes have been copied, whichever occurs first.

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

char mem1[16] = "aaaaaaaaaaaaaaa";
char mem2[16] = "zzzXzzzzzzzzzzz";
char *ptr;
int c;
size_t n;

main()
{
   c = 'X';
   n = 5;

   ptr = memccpy(mem1, mem2, c, n);

   printf("mem1 = %s\n", mem1);
}
```

Because the `X` character is encountered in the memory area pointed to by `mem2` before five bytes are copied, the output from the preceding program is as follows:

```
mem1 = zzzXaaaaaaaaaaa
```

Related Functions

`memcpy`, `memmove`, `memset`, `strcpy`, `strncpy`

memchr

Purpose

Searches a memory area for the first occurrence of a specified byte.

Syntax

```
#include <string.h>

void *memchr(const void *mem, int c, size_t n);
```

Explanation

The `memchr` function searches the first `n` bytes of memory starting at the address pointed to by `mem`. The search ends when the byte `c` is found or when `n` bytes have been searched.

The `memchr` function takes three arguments.

- The `mem` argument specifies the address of the memory area to be searched.
- The `c` argument, an `int` value, indicates the byte for which to search. The `memchr` function converts the `c` argument to an `unsigned int` before it searches the memory area.
- The `n` argument specifies the maximum number of bytes to examine. The `n` argument is a value of the type `size_t`, which is a type definition declared in the `stddef.h` header file.

Return Value

If `memchr` finds `c`, it returns a pointer to the first occurrence of the byte.

If `memchr` does not find `c`, it returns the `NULL` pointer constant.

Examples

The following program uses `memchr` to search a specified memory area for the first occurrence of a byte having the value of the character `t`.

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

char mem[] = "A day in the life";
char *first_occurrence;
int c;
size_t n;

main()
{
   c = 't';
   n = 12;

   first_occurrence = memchr(mem, c, n);

   printf("first occurrence = %s\n", first_occurrence);
}
```

The output from the preceding program is as follows:

```
first occurrence = the life
```

Related Functions

`memcmp`, `strchr`, `strcspn`, `strpbrk`, `strrchr`, `strspn`, `strtok`

memcmp

Purpose

Compares a specified number of bytes in one memory area with the same number of bytes in another memory area.

Syntax

```
#include <string.h>

int memcmp(const void *mem1, const void *mem2, size_t n);
```

Explanation

The `memcmp` function compares the first `n` bytes in the memory area pointed to by `mem1` with the first `n` bytes in the memory area pointed to by `mem2`. The `mem1` and `mem2` arguments specify the beginning address of separate memory areas. The `n` argument specifies the number of bytes to compare. The `n` argument is a value of the type `size_t`, which is a type definition declared in the `stddef.h` header file.

The `memcmp` function uses the following procedure to compare the first `n` bytes in the two memory areas. The two areas of memory are compared, byte by byte, until two bytes are found whose contents differ. The `memcmp` function then compares the two bytes and returns a value based on which of the two bytes' unsigned values is greater.

Notes:

1. If either of the memory areas being compared contains a structure, bytes used for alignment padding within the structure may have unpredictable values unless the contents of the entire object have been explicitly set, for example, with `calloc` or `memset`.

2. If either of the memory areas being compared contains a union or a string shorter than its allocated space, `memcmp` can return unpredictable results.

Return Value

The `memcmp` function returns a value based on the comparison of the objects pointed to by `mem1` and `mem2`. If two bytes that differ are found, `memcmp` returns an integer that is greater than 0, or is less than 0, based on which of the two bytes has the greater numeric value. The following table shows the possible return values of `memcmp` and their meanings.

Return Value	**Result of the Comparison**
Greater than 0	When two bytes are found that are different, the byte in the area located by `mem1` has a greater unsigned value than the byte in the area located by `mem2`.
Less than 0	When two bytes are found that are different, the byte in the area located by `mem1` has a lesser unsigned value than the byte in the area located by `mem2`.
Equal to 0	In the first `n` bytes, every byte in the area located by `mem1` is identical to the corresponding byte in the area located by `mem2`.

Examples

The following program uses `memcmp` to compare the first five bytes in one memory area with the first five bytes in another memory area.

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

char mem1[10] = "ABCDEFGHI";
char mem2[10] = "ABCDeFGHI";
size_t n;
int return_value;

main()
{
   n = 5;

   return_value = memcmp(mem1, mem2, n);

   if (return_value > 0)
      printf("mem1 is greater than mem2.\n");

   if (return_value < 0)
      printf("mem2 is greater than mem1.\n");

   if (return_value == 0)
      printf("mem1 and mem2 are equal.\n");
}
```

The output from the preceding program is as follows:

```
mem2 is greater than mem1.
```

Related Functions

`strcmp`, `strncmp`

memcpy

Purpose

Copies a specified number of bytes from one memory area into another memory area.

Syntax

```
#include <string.h>

void *memcpy(void *mem1, const void *mem2, size_t n);
```

Explanation

The `memcpy` function copies `n` bytes from the memory area pointed to by `mem2` into the memory area pointed to by `mem1`. The `mem1` and `mem2` arguments specify the beginning address of separate memory areas. The `n` argument specifies the number of bytes to copy. The `n` argument is a value of the type `size_t`, which is a type definition declared in the `stddef.h` header file.

If some parts of the memory areas specified by `mem1` and `mem2` overlap, the `memcpy` function can produce **incorrect** results. When the copying takes place between two memory areas that might overlap, use the `memmove` function, which is guaranteed to produce correct results with overlapping memory areas.

Return Value

The `memcpy` function returns a pointer to `mem1`, the memory area to which the bytes were copied.

Examples

The following program uses `memcpy` to copy five bytes from one memory area into another memory area.

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

char mem1[16] = "XXXXXXXXXXXXXXX";
char mem2[6] = "OOOOO";
char *ptr;
size_t n;

main()
{
   n = 5;
```

(Continued on next page)

```
     ptr = memcpy(mem1, mem2, n);

     printf("mem1 = %s\n", mem1);
  }
```

The output from the preceding program is as follows:

```
mem1 = OOOOOXXXXXXXXXX
```

Related Functions

`memccpy`, `memmove`, `memset`, `strcpy`, `strncpy`

memmove

Purpose

Copies a specified number of bytes from one memory area into another memory area. With `memmove`, some parts of the memory areas can overlap.

Syntax

```
#include <string.h>

void *memmove(void *mem1, const void *mem2, size_t n);
```

Explanation

The `memmove` function copies `n` bytes from the memory area pointed to by `mem2` into the memory area pointed to by `mem1`. The `mem1` and `mem2` arguments specify the beginning address of separate memory areas. The `n` argument specifies the number of bytes to copy. The `n` argument is a value of the type `size_t`, which is a type definition declared in the `stddef.h` header file.

Even if some parts of the memory areas specified by `mem1` and `mem2` overlap, the `memmove` function is guaranteed to produce correct results.

Return Value

The `memmove` function returns a pointer to `mem1`, the memory area to which the bytes were copied.

Examples

The following program uses `memmove` to copy seven bytes from one memory area into another, overlapping memory area, which is three bytes to the right of the first area.

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

char mem[20] = "ABCDEF";
char *ptr;
size_t n;

main()
{
   n = 7;

   ptr = memmove(mem + 3, mem, n);
```

(Continued on next page)

```
        printf("mem = %s\n", mem);
    }
```

The output from the preceding program is as follows:

```
mem = ABCABCDEF
```

Related Functions

`memccpy`, `memcpy`, `memset`, `strcpy`, `strncpy`

memset

Purpose

Copies the value of a given byte into a specified number of bytes in a memory area.

Syntax

```
#include <string.h>

void *memset(void *mem, int c, size_t n);
```

Explanation

The `memset` function copies the value of `c` into the first `n` bytes of memory starting at the address pointed to by `mem`. The `memset` function takes three arguments.

- The `mem` argument specifies the address of a memory area.
- The `c` argument indicates the value that will be copied into each byte of the memory area. The `memset` function converts the `c` argument to an `unsigned int` before it copies `c` into the memory area.
- The `n` argument specifies the maximum number of bytes into which `c` will be copied. The `n` argument is a value of the type `size_t`, which is a type definition declared in the `stddef.h` header file.

Typically, the `memset` function is used to initialize large areas of memory to a specific value.

Return Value

The `memset` function returns a pointer to `mem`, the memory area to which the bytes were copied.

Examples

The following program uses `memset` to copy a byte having the value of the character `X` into five bytes of memory starting at the address of an array, `mem`.

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

char mem[5], *ptr;
int c;
size_t n;
```

(Continued on next page)

```
main()
{
   int i;
   c = 'X';
   n = sizeof(mem);

   ptr = memset(mem, (char)c, n);

   for (i = 0; i < 5; i++)
      printf("mem[%d] = %c\n", i, mem[i]);
}
```

The output from the preceding program is as follows:

```
mem[0] = X
mem[1] = X
mem[2] = X
mem[3] = X
mem[4] = X
```

Related Functions

`memccpy`, `memcpy`, `memmove`, `strcpy`, `strncpy`

`modf`

Purpose

Breaks down a floating-point value into its integer and fractional parts.

Syntax

```
#include <math.h>

double modf(double value, double *iptr);
```

Explanation

The `modf` function breaks down a floating-point value, `value`, into its integer and fractional parts. The `modf` function returns the fractional part and, as a side effect, stores the integer part in the `double` variable pointed to by `iptr`. Both the returned fraction and stored integer have the same sign as `value`.

The `value` argument, a `double` value, contains the floating-point number to break down. The `iptr` argument is the address of a `double` variable.

Return Value

The `modf` function returns the fractional part of the floating-point value `value`. The returned value has the same sign as `value`.

Examples

The following program uses `modf` to break down a floating-point value, entered at the terminal's keyboard, into an integer and a fraction.

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

main()
{
   double value, integer, fraction;

   printf("Enter a number that you want to break down ");
   printf("into an integer and a fraction.\n");
   scanf("%lf", &value);

   fraction = modf(value, &integer);
```

(Continued on next page)

```
     printf("The floating-point value %.3lf has\n", value);
    printf("an integer part = %.3lf and a fractional part = %.3lf.\n",
            integer, fraction);
}
```

If `-123.456` were entered at the terminal’s keyboard, the output would be as follows:

```
The floating-point value -123.456 has
an integer part = -123.000 and a fractional part = -0.456.
```

Related Functions

`exp`, `frexp`, `ldexp`

offsetof

Purpose

Gets the offset, in bytes, of a specified member within a structure.

Syntax

```
#include <stddef.h>

size_t offsetof(struct_type, member_name);
```

Explanation

The `offsetof` macro gets the offset, in bytes, of the structure member *`member_name`* from the beginning of a structure of the type *`struct_type`*. The *`struct_type`* can be either a structure tag or a `typedef` name for a structure type. If the *`member_name`* argument is a bit field, the behavior of `offsetof` is unpredictable.

If the specified structure type contains alignment padding preceding *`member_name`*, the returned offset includes those bytes of padding.

Return Value

The `offsetof` macro returns the offset, in bytes, of the given structure member from the beginning of the specified structure type. The return value has the type `size_t`, which is a type defined in the `stddef.h` header file.

Examples

The following program uses `offsetof` to get the offset of the `i` structure member in a structure type `struct s1`, whose members are allocated using the longmap alignment rules.

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

struct $longmap s1
   {
   char c;
   double d;
   int i;
   };

size_t offset;
```

(Continued on next page)

```
main()
{
   offset = offsetof(struct s1, i);

   printf("The i member has an offset of %d\n", offset);
}
```

The output from the preceding program is as follows:

```
The i member has an offset of 16
```

Notice that, in the preceding example, the offset for the `i` member includes the bytes of alignment padding that precede that member. Figure 11-1 shows how a structure object of the type struct `s1` would be allocated. In the figure, each square represents one byte of memory. A square containing an identifier, such as `c`, indicates a byte used for that member. A square containing an asterisk (`*`) indicates a byte used for alignment padding.

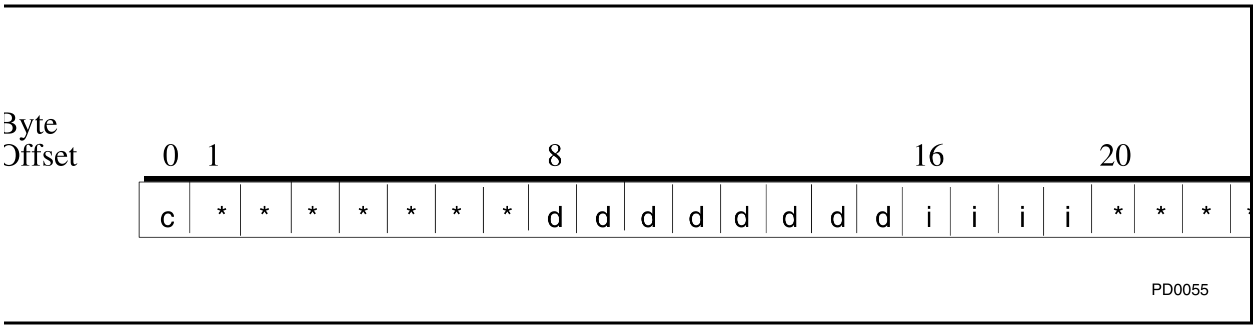


Figure 11-1. Structure Member Offsets

For information on alignment of members within a structure, see the "Data Alignment" section in Chapter 5.

Related Functions

None

onexit

Purpose

Registers a function to be invoked upon normal termination of the program.

Syntax

```
#include <stdlib.h>

onexit_t onexit(onexit_t func_ptr);
```

Explanation

The `onexit` function registers a function, pointed to by `func_ptr`, to be invoked upon normal termination of the program. The pointed-to function cannot have arguments or return a value. The `func_ptr` argument has the type `onexit_t`, which is a type definition declared in the `stdlib.h` header file.

You can register up to 32 functions with `onexit`. The functions are invoked in the reverse order of their registration when the program terminates normally. A program terminates normally if it calls the `exit` function or if `main` returns.

The `onexit` function allows the program to invoke up to 32 functions that perform additional cleanup tasks before the program exits to the operating system.

Return Value

The `onexit` function returns a pointer of the type `onexit_t` if the function is successfully registered. It returns the `NULL` pointer constant if the function is not successfully registered.

Examples

The following program uses `onexit` to register two functions, `func1` and `func2`, to be invoked when the program terminates normally.

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

void func1(void), func2(void);
onexit_t func1_ptr = func1, func2_ptr = func2, test_ptr;
```

(Continued on next page)

```
main()
{
   test_ptr = onexit(func1_ptr);
   if (test_ptr != NULL)
      printf("func1 registered successfully.\n");
   else
      printf("func1 not registered successfully.\n");

   test_ptr = onexit(func2_ptr);
   if (test_ptr != NULL)
      printf("func2 registered successfully.\n");
   else
      printf("func2 not registered successfully.\n");

   printf("Normal program termination.\n");
}

void func1(void)
{
   printf("On exiting, func1 is called SECOND.\n");
}

void func2(void)
{
   printf("On exiting, func2 is called FIRST.\n");
}
```

The output from the preceding program is as follows:

```
func1 registered successfully.
func2 registered successfully.
Normal program termination.
On exiting, func2 is called FIRST.
On exiting, func1 is called SECOND.
```

Related Functions

`exit`

`open`

Purpose

Opens a file and associates a file descriptor with the file.

Syntax

```
#include <c.utilities.h>
#include <fcntl.h>

int open(char *path_name, int o_flag, ...);

int mode;                    /*  mode is an optional argument  */
```

Explanation

The `open` function opens the file identified by `path_name` and associates a file descriptor with the file. The `o_flag` argument indicates how `open` is to open the file. For example, the flags specified in `o_flag` indicate the type of I/O for which the file will be opened. In VOS C, a third argument `mode` is optional. The values allowed for `mode`, `path_name`, and `o_flag` are explained in the sections that follow.

You use the file descriptor returned by `open` to identify the file with many of the UNIX I/O functions, such as `read` and `write`. If the `open` function opens the file successfully, it sets the file-position indicator to the beginning of the file.

Mode Argument. The `mode` argument to `open` is optional. When the `mode` argument is used, a sample invocation of `open` is as follows:

```
file_des = open(path_name, o_flag, mode);
```

In the VOS environment, `open` ignores the `mode` argument.

Path Name Argument. The `path_name` argument is a pointer to a character string that has the following form:

"*file_path_name* [option ...]"

Within the `path_name` argument, the required *`file_path_name`* is a full or relative path name specifying the name of the file or device to open. In VOS C, the `path_name` may also include one or more *`option`* values that indicate the way the file is to be opened, including the following:

- file organization
- locking mode
- raw input and output modes (for a terminal device only)
- bulk raw input mode (for a terminal device only)
- associated port.

If the `path_name` argument contains a conflicting or duplicate option, then `open` sets the external variables `errno` and `os_errno` to `e$title_inconsistent` (`1310`). If the options specified contain a syntax error, `open` sets `errno` to `e$bad_title_syntax` (`1308`).

Note: If the values specified in the `path_name` argument conflict with the values specified in the `o_flag` argument, or if erroneous values are specified in either argument, the results are unpredictable.

See "Path Name Options," later in the Explanation, for more information on the optional parts of the `path_name` argument.

The `o_flag` Argument. The `o_flag` argument indicates what type of I/O will be used when the file is opened, whether the file will be opened in text or binary mode, which file organization will be used when a file is created, and other information. The flags are constants defined in the `fcntl.h` header file. You **must** include the `fcntl.h` header file before you can use any of the `o_flag` defined constants. The `o_flag` argument can have one or more of the values shown in Table 11-29.

Table 11-29. The `o_flag` Values for the `open` Function *(Page 1 of 2)*

o_flag	Description
`O_RDONLY`	Opens a file for reading only.
`O_WRONLY`	Opens a file for writing only.
`O_RDWR`	Opens a file for update (reading and writing).
`O_APPEND`	Sets the file-position indicator to the end of the file before every write operation.
`O_CREAT`	Creates and opens a new file if the file specified by *`file_path_name`* does not exist. The access control list is set to the default access control list for the directory that will contain the file. The author is set to the user name of the process that creates the file. The `O_CREAT` flag is ignored if the file does exist.
`O_EXCL`	Tells the function to return the value -1 if you have also specified `O_CREAT` and the file already exists. Otherwise, the flag is ignored.

Table 11-29. The `o_flag` Values for the `open` Function *(Page 2 of 2)*

o_flag	Description
`O_TRUNC`	Opens and truncates (sets to length 0) an existing file. This truncation deletes the previous contents of the file.
`O_BINARY`	Opens a file in binary mode. If you do not specify `O_BINARY`, the function opens the file in text mode.
`O_SEQL`	Specifies sequential file organization when a file is created using `O_CREAT`.
`O_STRM`	Specifies stream file organization when a file is created using `O_CREAT`. If you do not specify a file organization option when creating a file, stream file organization is the default.
`O_NDELAY`	Sets the locking mode to set-lock-don't-wait.

For `o_flag`, you must specify an I/O type: either `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. In addition, you can specify other flags if needed. Some combinations of flags are not allowed. For example, the `O_RDONLY`, `O_WRONLY`, and `O_RDWR` flags are mutually exclusive. You cannot specify the `O_CREAT` flag without also specifying either `O_WRONLY` or `O_RDWR`. The `O_SEQL` and `O_STRM` flags are VOS C extensions and are not portable.

To specify more than one flag, use the bitwise OR operator (|) between the flags. For example, the following `open` function sets both the `O_WRONLY` and `O_TRUNC` flags, opening a file for writing and truncating the file.

```
    file_des = open(existing_file, O_WRONLY | O_TRUNC);
```

Table 11-30 shows the correspondence between the values of the `o_flag` argument in the `open` function and the values of the `mode` argument in the `fopen` function. The table shows the correspondences for text mode only. Similar correspondences exist for binary mode.

Table 11-30. I/O Modes with the `open` and `fopen` Functions *(Page 1 of 2)*

Flags in open	Mode in fopen	Description
`O_RDONLY`	`"r"`	Opens a text file for reading.
`O_WRONLY | O_TRUNC | O_CREAT`	`"w"`	Opens a text file for writing. If the specified file exists, it is truncated. If the file does not exist, it is created.
`O_WRONLY | O_APPEND | O_CREAT`	`a`	Opens a text file for writing at end-of-file (that is, append mode). If the specified file does not exist, it is created.
`O_RDWR`	`r+`	Opens a text file for update (reading and writing).

Table 11-30. I/O Modes with the `open` and `fopen` Functions *(Page 2 of 2)*

Flags in open	Mode in fopen	Description
`O_RDWR | O_TRUNC | O_CREAT`	`w+`	Opens a text file for update. If the specified file exists, it is truncated. If the file does not exist, it is created.
`O_RDWR | O_APPEND | O_CREAT`	`a+`	Opens a text file for update. All writing takes place at end-of-file. If the specified file does not exist, it is created.

Append Mode. When you specify `O_APPEND` in the `o_flag` argument, `open` opens a file in append mode. By default, with append mode, it is impossible to overwrite information already in the file. The file-positioning functions, such as `lseek`, can be used to change the file-position indicator to any position in the file. However, with append I/O, the file-position indicator is, by default, ignored for write operations. All output is written to the end of the file, and the file-position indicator is repositioned to the end of the output.

In VOS C, when two separate processes open the same file for append I/O, each process can write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

Update Mode. When you specify `O_RDWR` in the `o_flag` argument, `open` opens the file for update. In update mode, you can perform both input and output on the file. However, output **cannot** be directly followed by input without an intervening call to the `fflush`, `fseek`, `fsetpos`, `lseek`, or `rewind` function. Input **cannot** be directly followed by output without an intervening call to `fseek`, `fsetpos`, `lseek`, or `rewind`, or an input operation that encounters end-of-file.

When you specify `O_RDWR` and `O_APPEND` in `o_flag`, `open` opens the file for update and all output is written to the end of the file unless you use `fseek`, `fsetpos`, `lseek`, or `rewind` to change the file position to another location in the file.

Path Name Options. The syntax for the `path_name` argument to `open` is as follows:

```
"file_path_name [option ...]"
```

In VOS C, the `path_name` argument can include one or more *`option`* values specifying the file organization of a file that `open` creates, locking mode, raw input and output modes, bulk raw input mode, and associated port. Notice that a space is required between *`file_path_name`* and the first option.

The allowed values for the options within the `path_name` argument are shown in Table 11-31, grouped by category. You can select only one value from each category. For example, you can select only one file organization in an `open` call.

Table 11-31. Path Name Options in the `open` Function

Type of Option	Path Name Option Value	Description
File Organization	`-stream`	When specified with `O_CREAT`, creates and opens a file with stream file organization if no file of the given name exists. When you do not specify a file organization option when creating a file, stream file organization is the default.
	`-sequential`	When specified with `O_CREAT`, creates and opens a sequential file if no file of the given name exists.
	`-fixed` *`size`*	When specified with `O_CREAT`, creates and opens a fixed file if no file of the given name exists. You give the file's record size in *`size`*.
	`-relative` *`size`*	When specified with `O_CREAT`, creates and opens a relative file if no file of the given name exists. You give the file's maximum record size in *`size`*.
Locking Mode	`-implicit_locking` or `-implicitlock`	Opens the file using implicit locking mode. If you do not specify a locking mode option, implicit locking is the default.
	`-nowait`	Opens the file using set-lock-don't-wait locking mode.
	`-wait`	Opens the file using wait-for-lock locking mode.
Raw Input and Output Modes	`-raw_mode`	Causes all terminal access to be performed using raw input mode and raw output mode. If you do not specify raw mode, the default input mode is command-line input mode, and the default output mode is generic (or normal) output.
Bulk Raw Input Mode	`-bulk_raw_input_mode`	Causes all terminal input to be performed using bulk raw input mode. If you do not specify bulk raw input mode, the default input mode is normal raw input mode.
Associated Port	`-port_name` *`port_name`*	Attaches a port of the name indicated in *`port_name`* to the specified file.

File Organization. You indicate a file organization **only when** you are creating a file. You can create a file by specifying `O_CREAT` in the `o_flag` argument. If you specify one of the file organization options and a file of the name specified in *`file_path_name`* does not exist, the `open` function creates a file of the given type. If a file of the name specified in *`file_path_name`* does exist, the `open` function ignores the file organization option. With the path name options, you can specify one of four file organizations: stream, sequential, fixed, or relative. With the `o_flag` argument, you can specify one of two file organizations: stream or sequential.

See the *VOS C User's Guide (R141)* for information on file organizations.

Locking Mode. Within the `path_name` argument, you can indicate one of three locking modes. If you do not explicitly specify a locking mode in `open` and the file is not opened in input mode (`O_RDONLY`), implicit-locking mode is the default. If the file is opened in input mode, normal locking rules are ignored, and the locking mode that you specify in `open` has no effect on I/O operations.

For information on the locking modes shown in Table 11-31, see the *VOS C User's Guide (R141)*.

Raw Input and Output Modes. Within `path_name`, you can specify two path name options to control input and output with a terminal:

- `-raw_mode` specifies raw input mode and raw output mode.
- `-bulk_raw_input_mode` specifies bulk raw input mode.

If you use `open` to set the terminal into raw mode or bulk raw input mode, the input-output mode for the terminal is automatically reset when the terminal is closed. See the *VOS Communications Software: Asynchronous Communications (R025)* manual for detailed information on input and output modes with a terminal device.

Raw Mode. The `-raw_mode` option tells `open` to set the input-output mode for the terminal, specified by *`file_path_name`*, to raw mode. In raw mode, the terminal is accessed using raw input mode and raw output mode. Raw mode can be specified for terminal devices only.

For input from a terminal, the `-raw_mode` option sets the device in normal raw input mode. With a terminal, normal raw input mode does not return to the program until the buffer is full. For this reason, normal raw input mode is rarely used. Bulk raw input mode, discussed later in the Explanation, provides a useful alternative. For output to a terminal, the `-raw_mode` option causes `s$write_raw` calls to be issued rather than `s$seq_write` calls.

Bulk Raw Input Mode. The `-bulk_raw_input_mode` option tells `open` to set the input mode for the terminal, specified by *`file_path_name`*, to bulk raw input mode. Bulk raw input mode can be specified for terminal devices only. In bulk raw input mode, the `s$read_raw` subroutine returns any available characters, even if it has not read enough characters to fill the input buffer.

Associated Port. The `-port_name` option has the following form:

```
-port_name port_name
```

The *`port_name`* is a name identifying a port. The *`port_name`* can be 1 to 32 characters long. The `open` function attaches a port named *`port_name`* to the file specified in *`file_path_name`*. If the port specified in *`port_name`* is already attached, `open` accesses the file attached to the port and ignores *`file_path_name`*.

Error Codes. If an error occurs, `open` sets the external variables `errno` and `os_errno` to the error code number returned by the operating system. The following table lists some of the error code values that are possible.

Error Code Name	Number
`e$bad_pathname`	`1004`
`e$bad_title_syntax`	`1308`
`e$caller_must_wait`	`1277`
`e$cannot_region_lock`	`3539`
`e$device_already_assigned`	`1155`
`e$device_not_found`	`1220`
`e$directory_exists`	`1045`
`e$file_exists`	`1050`
`e$file_in_use`	`1084`
`e$invalid_file_pointer`	`3929`
`e$invalid_file_type`	`1052`
`e$invalid_io_operation`	`1040`
`e$invalid_record_size`	`1053`
`e$must_lock_implicitly`	`1497`
`e$no_ports_available`	`1006`
`e$not_on_dir`	`1186`
`e$object_not_found`	`1032`
`e$port_already_attached`	`1008`
`e$portname_not_found`	`1028`
`e$title_inconsistent`	`1310`

Return Value

If the `open` function opens the file successfully, it returns a file descriptor in the form of a non-negative integer.

If `open` fails to open the file, the function returns the value -1.

Examples

The following program fragment contains three examples of how `open` is used to open a file.

```
#include <c_utilities.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
```

(Continued on next page)

```
main()
{
   int file_des1, file_des2, file_des3;
   int mode = 0;

                  /*  Example 1  */
   errno = 0;
   if ( (file_des1 = open("file1", O_RDWR, mode)) == -1 )
      {
      printf("Error opening file1:  errno = %d\n", errno);
      exit(1);
      }

                  /*  Example 2  */
   errno = 0;
   if ( (file_des2 = open("file2", O_WRONLY | O_CREAT | O_SEQL)) == -1 )
      {
      printf("Error opening file2:  errno = %d\n", errno);
      exit(1);
      }

                  /*  Example 3  */
   errno = 0;
   if ( (file_des3 = open("file3 -port_name port3", O_RDONLY)) == -1 )
      {
      printf("Error opening file3:  errno = %d\n", errno);
      exit(1);
      }
      .
      .
      .
}
```

In example 1, `open` opens a file named `file1` for update (reading and writing) because `o_flag` is `O_RDWR`. In VOS C, `open` ignores the optional third argument, `mode`.

In example 2, `open` opens a file named `file2` for output because `o_flag` contains `O_WRONLY`. If `file2` does not exist, `open` creates a file having sequential file organization because `o_flag` contains `O_CREAT` and `O_SEQL`. If `file2` exists, `open` ignores `O_CREAT` and `O_SEQL`.

In example 3, `open` opens a file named `file3` for input because `o_flag` is `O_RDONLY`. In addition, `open` attaches a port named `port3` to the specified file because the `path_name` argument specifies `-port_name port3`. If `port3` is already attached, `open` accesses the file attached to the port and ignores `file3` in the `path_name` argument.

Related Functions

`close`, `creat`, `fdopen`, `fileno`, `fopen`, `freopen`

perror

Purpose

Constructs an error message and writes that message to the standard error file, `stderr`.

Syntax

```
#include <stdio.h>

void perror(const char *string);
```

Explanation

The `perror` function constructs an error message and writes that message to the preopened file pointed to by `stderr`. The file pointed to by `stderr` is usually associated with the terminal's screen. Therefore, with most applications, the error message is displayed on the screen.

The `perror` function constructs the error message in the following manner.

1. For the message's first element, the `perror` function uses the string pointed to by `string` if `string` is not equal to the `NULL` pointer constant.
2. The function appends a colon and a space character to the string if `string` is not equal to the `NULL` pointer constant.
3. The function concatenates the operating system error message corresponding to the error code number stored in the external variable `errno`.
4. The function appends a newline character (`\n`).

Certain C library functions store an operating system error code number in `errno` to indicate the reason why an operation fails. If `errno` has not been set to an error code number, `perror` does not include an operating system error message in the message it displays.

Typically, when a function sets `errno` on a failed operation, you could call `perror` immediately after the function returns a value that indicates an error has occurred. The `perror` function displays, on the terminal's screen, the cause of the failure. See the "File I/O Error Handling" section in Chapter 10 for more information on `errno`.

Return Value

The `perror` function returns no value.

Examples

The following program uses `perror` to display the words `ERROR MESSAGE`, followed by a colon and a space and an operating system error message corresponding to error code number `5034`.

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

double d;
char *num_string;

main()
{
   num_string = "  1.23456e-789";

   errno = 0;
   d = atof(num_string);          /*  Out-of-range number causes
underflow  */

   if (d == 0.0)
      {
      perror("ERROR MESSAGE");
      printf("\nerrno = %d\n", errno);
      }
   else
      printf("d = %lf\n", d);
}
```

The output from the preceding program is as follows:

```
ERROR MESSAGE: Math result not representable.

errno = 5034
```

Related Functions

`clearerr`, `feof`, `ferror`

pow

Purpose

Computes the value of one argument raised to the power of another argument.

Syntax

```
#include <math.h>

double pow(double x, double y);
```

Explanation

The `pow` function calculates `x` raised to the power of `y`. Both the `x` and `y` arguments are values of the type `double`.

A domain error occurs, and `pow` sets `errno` to `EDOM` when one of the following conditions is true:

- if `x` is 0 and `y` is less than or equal to 0
- if `x` is negative and `y` is not an integer.

If the result creates an overflow or underflow condition, a range error occurs, and `pow` sets `errno` to `ERANGE`.

Return Value

The `pow` function returns the value of `x` raised to the power of `y`. When the result creates an overflow condition, `pow` returns `HUGE_VAL`. On underflow, it returns the value 0.

Examples

The following program uses `pow` to calculate the value of a floating-point number raised to the power of another floating-point number entered at the terminal's keyboard.

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

main()
{
   double x, y, result;
```

(Continued on next page)

```
          printf("Enter a number for x.\n");
          scanf("%lf", &x);
          printf("Enter the number to which you want to raise x.\n");
          scanf("%lf", &y);

          errno = 0;
          result = pow(x, y);

          if (errno == EDOM)
             puts("A domain error occurred.");
          else
             if (errno == ERANGE)
                puts("A range error occurred.");
             else
                printf("%.3lf raised to the power %.3lf = %.3lf\n", x, y,
       result);
       }
```

If two values, `10.0` for `x` and `2.0` for `y`, were entered at the terminal's keyboard, the output would be as follows:

```
10.000 raised to the power 2.000 = 100.000
```

Related Functions

`exp`, `log`, `log10`, `sqrt`

printf

Purpose

Writes a formatted string of characters to the standard output file, `stdout`.

Syntax

```
#include <stdio.h>

int printf(const char *format, ...);
```

Explanation

The `printf` function writes output to the preopened file pointed to by `stdout`. The file pointed to by `stdout` is usually associated with the terminal's screen. The displayed output is under the control of the string pointed to by `format`.

> **Note:** The `printf` function does not support generic output sequences embedded within the format string. To write generic output to a terminal, use any of the write subroutines except `s$write_raw`. See the *VOS Communications Software: Asynchronous Communications (R025)* manual for information on generic output.

The string pointed to by `format` can consist of zero or more directives. The `printf` function executes each directive in turn. It returns when it encounters the end of the format string. The directives consist of the following:

- characters, including escape sequences
- conversion specifications.

The `printf` function displays ordinary characters (not `%`) unchanged on the terminal's screen. The following example shows a `printf` call consisting only of characters and escape sequences.

```
printf("\aAn incorrect choice.  Try again.\n");
```

In the preceding `printf` call, `\a` is an escape sequence representing an audible alert, and `\n` is an escape sequence representing the newline character. See the "Escape Sequences" section in Chapter 2 for information on escape sequences.

The format string can include conversion specifications. Each *conversion specification* begins with a `%` and indicates how the value of a subsequent argument will be converted and written. The number and types of the arguments in a `printf` call depend on the conversion specifications given in the format string. Each conversion specification results in fetching zero or more arguments. In a `printf` call, the syntax for the argument list is as follows:

```
printf(format,[argument]...)
```

Each *`argument`* must be an expression yielding an object type appropriate for the conversion specification. If there are insufficient arguments for the format, the conversions have unpredictable results. If the format is exhausted while arguments remain, the extra arguments are evaluated but otherwise ignored.

The following example shows a `printf` call consisting of characters, a `%d` conversion specification, and a corresponding `int` argument.

```
int i_num = 99;

printf("i_num = %d\n", i_num);     /*  i_num = 99  */
```

In the preceding `printf` call, the value of `i_num` is converted using the `%d` format and the resulting characters are displayed in the same position as `%d` within the output.

Conversion Specifications. The format for a conversion specification is as follows:

```
%[flags][width] .[precision][modifier] conversion_specifier
```

After the beginning `%` character, the parts of the conversion specification must appear in the sequence shown in the preceding format.

Zero or more *`flags`* indicate how the output will be justified, use of plus and/or minus signs, use of a decimal point with numeric values, and other formatting options. Table 11-32 explains the characters allowed as flags.

Table 11-32. Formatted Output: Flags in a Conversion Specification *(Page 1 of 2)*

Flag	Meaning
`-`	The output is left-justified within its field. If you do not use the `-` flag, the default is right justification.
`+`	For a `d`, `i`, `e`, `E`, `f`, `g`, or `G` conversion, the signed-number result begins with a `+` or `-` sign. If you do not use the `+` flag, the default is to precede only negative numbers with a sign. For all other conversions, the `+` flag has no effect.
space	If the first character of a signed number is not a sign or if a signed conversion results in no characters, a space character is prefixed to the result. If you do not use a space-character flag, no space is prefixed to these conversions. If you use both the `+` flag and the space-character flag, the space-character flag is ignored.

Table 11-32. Formatted Output: Flags in a Conversion Specification *(Page 2 of 2)*

Flag	Meaning
`#`	The result is converted to an *alternate form* as follows: For an `e`, `E`, `f`, `g`, or `G` conversion, the result always contains a decimal point even if no digits follow the decimal point. If you do not use the `#` flag, a decimal point appears in the result only when a digit follows the decimal point. For a `g` or `G` conversion, trailing zeroes are **not** removed from the result. For an `o` conversion, the precision is increased to force the first digit of the result to be a 0. For a `p` conversion, the result is prefixed with `0x`. For an `x` or `X` conversion, nonzero results are prefixed with `0x` and `0X`, respectively. For all other conversions, the `#` flag has no effect.
`0`	For a `d`, `i`, `e`, `E`, `f`, `g`, `G`, `o`, `u`, `x`, or `X` conversion, the result is padded with leading zeroes (after any sign or base prefix) to the length specified by the field width. If you do not use the `0` flag, padding is with spaces. If you use both the `0` and `-` flags, the `0` flag is ignored. If a precision is specified for `d`, `i`, `o`, `u`, `x`, or `X` conversions, the `0` flag is ignored.

Note: A `0` following the `%` character is always interpreted as a `0` flag, never as the beginning of a field width.

Within a conversion specification, an optional *`width`* indicates the field width: the minimum number of characters to output. If the converted value has fewer characters than the field width, it is, by default, padded with spaces on the left. The *`width`* takes the form of a decimal integer or an asterisk (`*`). For information on using an asterisk, see the discussion later in this section.

Two flags affect how a result is padded to the field width specified. If you specify the `0` flag, the result is padded with zeroes to the width indicated. If you specify the `-` flag, the padding appears on the right of the result.

Depending on the *conversion_specifier* that you give, the optional *precision* indicates one of the following:

- for d, i, o, u, x, X, and p conversions, the minimum number of digits to appear in the result
- for e, E, and f conversions, the number of digits to appear after the decimal point
- for the g and G conversions, the maximum number of significant digits
- for the s and v conversions, the maximum number of characters to be written from a string.

The *precision* is ignored for all other conversions. The *precision* takes the form of a period (.) followed by an optional decimal integer or asterisk (*). For information on using an asterisk, see the following discussion. If only a period is specified, the precision is 0.

The *width* or *precision*, or both, can be an asterisk. In this case, an int argument specifies the field width or precision. If you use an * for the field width or precision, the corresponding int argument always appears before the argument to be converted. In the following example, the field width is 9, and the precision is 5.

```
int num = -999;

printf("num = |%*.*d|\n", 9, 5, num);    /*  num = |   -00999|  */
```

When *width* is an *, a negative int argument is interpreted as the - flag followed by a positive field width. When *precision* is an *, a negative int argument is interpreted as if the precision were omitted (that is, the default precision is used).

An optional *modifier* can appear before the conversion specifier, indicating that the type of the argument and, in some cases, the type of resulting output is a short or long integer type. Table 11-33 explains each of the characters allowed for modifier.

Table 11-33. Formatted Output: Modifiers in a Conversion Specification *(Page 1 of 2)*

Modifier	Meaning
h	The expected argument and converted result are as follows: For a d or i conversion, the argument is a short int. The argument is promoted according to the integral promotions, and its value is converted to short int before printing. For an o, u, x, or X conversion, the argument is an unsigned short int. The argument is promoted according to the integral promotions, and its value is converted to unsigned short int before printing. For an n conversion, the argument is a "pointer to a short int."

Table 11-33. Formatted Output: Modifiers in a Conversion Specification *(Page 2 of 2)*

Modifier	Meaning
`l`	The expected argument is as follows: For a `d` or `i` conversion, the argument is a `long int`. For an `o`, `u`, `x`, or `X` conversion, the argument is an `unsigned long int`. For an `n` conversion, the argument is a “pointer to a `long int`.”

Conversion Specifiers. The required *`conversion_specifier`* can be one of several characters. The following paragraphs provide a full explanation of the arguments expected by and the output resulting from each of the conversion specifiers. Each conversion specifier can be used with the following formatted I/O functions: `fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`, and `vsprintf`.

With these conversion specifiers, a nonexistent or small field width does **not** cause truncation of the field. If the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

`c` The `int` argument is converted to an `unsigned char`, and the resulting character is written.

`d`, `i` The `int` argument is converted to a signed decimal in the format [`-`]`dddd`. The precision specifies the number of digits to appear. If the value being converted can be displayed with fewer digits than specified in the precision, the value is padded with leading zeroes. The output for a value of 0 with a precision of 0 is no characters. The default precision is 1.

`e`, `E` The `double` argument is converted in the format [`-`]*`d.ddd`*`e`±*`dd`*, where there is one digit before the decimal point and the number of digits after the decimal point is equal to the precision. If the precision is 0 and the `#` flag is not specified, no decimal point appears. The value is rounded (not truncated) to the appropriate number of digits. The `E` conversion specifier produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. If the value is 0, the exponent is 0. The default precision is 6.

`f` The `double` argument is converted to decimal notation in the format [`-`]*`ddd.ddd`*, where the number of digits after the decimal point is equal to the precision. If the precision is 0 and the `#` flag is not specified, no decimal point appears. If a decimal point appears, at least one digit appears before it. The value is rounded (not truncated) to the appropriate number of digits. The default precision is 6.

`g`, `G` The `double` argument is printed in format `f` or `e` (or in style `E` in the case of a `G` conversion specifier) with the precision specifying the number of significant digits. If the precision is 0, it is taken as 1. The format used depends on the value converted: style `e` or `E` will be used only if the exponent resulting from the conversion is less than -4 or greater than or equal to the precision. If the `#` flag is not specified, trailing zeroes are removed from the fractional portion of the result. A decimal point appears only if it is followed by a digit or if the # flag is used.

o, u, x, XThe unsigned int argument is converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x and X) in the format *dddd*. The letters abcdef are used for the x conversion specifier. The letters ABCDEF are used for the X conversion specifier. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, the value is padded with leading zeroes. The output for a value of 0 with a precision of 0 is no characters. The default precision is 1.

n The argument is a pointer to an int **into which is written** the number of characters output to the specified file so far by this call to the formatted I/O function. No argument is converted, and no output is written to the file.

p The argument is a pointer to void. The value of the pointer is converted to an absolute address in hexadecimal format. For example, if the argument were ((void *)10), the output would be 0000000a. The default precision is 8.

s The argument is a pointer to an array of char. Characters from the array are written up to (but not including) a terminating null character (\0), or until the number of characters specified by the precision is reached. If the precision is missing, it is taken to be infinite, and all characters up to the first null character are printed. If the pointer argument equals NULL or OS_NULL_PTR, that pointer value is written in the form <<NULL_PTR(0)>> and <<NULL_PTR(1)>>, respectively.

v The argument is a pointer to a char_varying string. Characters from the string are written until the number of characters indicated by the current value of the length field is reached, or until the number of characters specified by the precision is reached. If the pointer argument equals NULL or OS_NULL_PTR, that pointer value is written in the form <<NULL_PTR(0)>> and <<NULL_PTR(1)>>, respectively. This conversion specifier is a VOS C extension and is not portable.

% A percent sign character (%) is written. No argument is converted. To write a % character, the complete conversion specification should be %%. In VOS C, a flag, width, precision, or modifier is accepted and ignored.

When a formatted output function is called, some arguments specified with conversion specifiers undergo the default argument promotions. Specifically, the value of a float argument is promoted to double, and the integral promotions are performed on signed and unsigned char, signed and unsigned short, and bit-field arguments of less than 32 bits.

Although some of the preceding explanations specify one data type for the argument, more than one data type is allowed with many of these conversion specifiers. For example, the c conversion specifier specifies an int argument. However, because the argument undergoes the integral promotions before the printf call, the corresponding argument to the c specifier can also be, for example, a signed or unsigned char or a signed or unsigned short int.

For the e, E, f, g, and G conversions, if the value being converted is positive or negative infinity or is not a number, the value is written, respectively, as positive or negative INFINITY or positive or negative NaN. The sign that is written depends on the sign of the value being converted. If you specify an invalid conversion specification, no argument is converted and no output is written.

Table 11-34 provides a summary of the formatted output conversion specifiers. The table shows some of the argument types typically used with each specifier, and a description and example of the output generated by each specifier.

Table 11-34. Formatted Output: Conversion Specifier Summary *(Page 1 of 2)*

Conversion Specifier	**Argument Type**	**Output**
`c`	`int` `char`	Displays a single character. For example, `printf("%c", 'A')` displays `A`.
`d, i`	`int`	Displays a signed decimal. For example, `printf("%d", -123)` displays `-123`.
`e`, `E`	`double` `float`	Displays a floating-point value using exponential notation. For example, `printf("%e", -987.654)` displays `-9.876540e+002`. The `E` conversion specifier displays a number with an `E` instead of an `e` introducing the exponent.
`f`	`double`	Displays a floating-point value as a decimal
	`float`	fraction. For example, `printf("%f", -987.345)` displays `-987.345000`.
`g`, `G`	`double`	Displays a floating-point value using `e` or `f`
	`float`	format, whichever is more compact. For example, `printf("%g", -987.3456)` displays `-987.346`. The `G` conversion specifier displays the number with an `E` instead of an `e` introducing the exponent.
`n`	`int *`	Writes, into the pointed-to `int` variable, the number of characters output so far by this `printf` call.
`o`	`unsigned int`	Displays an unsigned integer in octal format. For example, `printf("%o", 0177)` displays `177`.
`p`	`void *`	Displays a pointer as an absolute address in hexadecimal format. For example, `printf("%p", NULL)` displays `00000000`.
`s`	`char *`	Displays a C string up to (but not including) the null character. For example, `printf("%s", "abcd")` displays `abcd`.
`v`	`char_varyi ng(`*n*`) *`	Displays a `char_varying` string up to the current length. For example, if `cv` is a `char_varying` object storing `"efgh"`, the call `printf("%v", &cv)` displays `efgh`.

Table 11-34. Formatted Output: Conversion Specifier Summary *(Page 2 of 2)*

Conversion Specifier	Argument Type	Output
`u`	`unsigned int`	Displays an unsigned integer in decimal format. For example, `printf("%u", 1234)` displays `1234`.
`x`, `X`	`unsigned int`	Displays an unsigned integer in hexadecimal format. For example, `printf("%x", 0x7F)` displays `7f`. The `X` conversion specifier displays the number with the uppercase letters `ABCDEF` instead of the lowercase letters `abcdef`.
`%%`	none	Displays a percent sign.

Return Value

The `printf` function returns the number of characters written to `stdout` if the formatted output specified is successfully displayed.

If an output error occurs, `printf` returns a negative value.

Examples

The following program uses `printf` to display four lines of formatted output on the terminal's screen.

```
#include <stdio.h>

short int s_num;
double d_num;
int i_num;
int return_value;

main()
{
   /*  This first conversion specification uses the + and 0 flags, a field
*/
   /*  width of 5, and the h modifier with the d conversion specifier.
*/

    s_num = 99;
    printf("s_num = |%+05hd|\n", s_num);

   /*  The next conversion specification uses the - flag, a field         */
   /*  width of 10, and precision of 5 with the f conversion specifier.
*/
```

(Continued on next page)

```
   d_num = 999.99;

   printf("d_num = |%-10.5f|\n", d_num);

  /*  The last three conversion specifications use:  the n specifier to
*/
  /*  get the number of characters written, and the d specifier to print
*/
  /*  that number and the return value of the previous printf call.
*/

   return_value = printf("123456789%n\n", &i_num);
   printf("number of characters written = %d; return_value = %d\n",
          i_num, return_value);
}
```

The output from the preceding program is as follows:

```
s_num = |+0099|
d_num = |999.99000 |
123456789
number of characters written = 9; return_value = 10
```

Related Functions

`fprintf`, `sprintf`, `vprintf`, `vfprintf`, `vsprintf`

putc

Purpose

Writes one character to a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int putc(int c, FILE *file_ptr);
```

Explanation

The `putc` macro writes one character, `c`, to the current position of the file associated with `file_ptr`. The macro then moves the file-position indicator ahead one character in the stream. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

If a write error occurs, the macro sets the error indicator associated with `file_ptr`. You can use the `ferror` function to determine if an error occurred.

When a write error occurs, `putc` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `write` function for the possible values that `putc` uses when setting these variables.

The `putc` macro can evaluate its argument more than once or not at all. Therefore, `file_ptr` should never be an expression with side effects. If a function is required, use `fputc`.

Return Value

The `putc` macro returns an integer value representing the character written to the file. If a write error occurs, `putc` returns `EOF`.

Examples

The following program opens two stream files named `in_file` and `out_file`. It reads each character from the first line of `in_file`. It uses `putc` to write each character from the line to `out_file`.

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

main()
{
   int c;
   FILE *file_ptr1;            /*  For in_file   */
   FILE *file_ptr2;            /*  For out_file  */

   if ( (file_ptr1 = fopen("in_file", "r")) == NULL )
      {
      printf("Could not open in_file.\n");
      exit(1);
      }

   if ( (file_ptr2 = fopen("out_file", "w")) == NULL )
      {
      printf("Could not open out_file.\n");
      exit(1);
      }

   c = getc(file_ptr1);
   while ( c != EOF )
      {
      putc(c, file_ptr2);
      if (c == '\n')
         break;
      else
         c = getc(file_ptr1);
      }
}
```

Related Functions

`fgetc`, `fputc`, `getc`, `getchar`, `putchar`

putchar

Purpose

Writes one character to the standard output file, `stdout`.

Syntax

```
#include <stdio.h>

int putchar(int c);
```

Explanation

The `putchar` macro writes one character, `c`, to the preopened file pointed to by `stdout`. The file pointed to by `stdout` is usually associated with the terminal's screen. Notice that `putchar` is equivalent to the following:

```
putc(c, stdout);
```

If a write error occurs, `putchar` sets the error indicator for `stdout`. You can use the `ferror` function to determine if an error occurred.

When a write error occurs, `putchar` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `write` function for the possible values that `putchar` uses when setting these variables.

In VOS C, `putchar` is implemented as a macro. If you need a function that performs the same operation, use `fputc(c, stdout)`.

Return Value

The `putchar` macro returns an integer value representing the character written to `stdout`. If a write error occurs, `putchar` returns `EOF`.

Examples

The following program reads each character in the first line of input entered at the terminal's keyboard. It uses `putchar` to display each character from the line on the terminal's screen.

```
#include <stdio.h>

int c;

main()
{
   puts("Enter a line of input.");

   while ( (c = getchar()) != '\n' )      /*  Look for newline  */
      {
      putchar(c);
      }
putchar('\n');
}
```

Related Functions

`fgetc`, `fputc`, `getc`, `getchar`, `putc`

`puts`

Purpose

Writes a string to the standard output file, `stdout`.

Syntax

```
#include <stdio.h>

int puts(const char *s);
```

Explanation

The `puts` function writes a string, `s`, to the preopened file pointed to by `stdout`. The `s` argument specifies the address of a null-terminated array of `char`. The file pointed to by `stdout` is usually associated with the terminal's screen.

Before `puts` writes the string, it appends a newline character (`\n`) to the string. The string's null character is not written.

If a write error occurs, `puts` sets the error indicator for `stdout`. You can use the `ferror` function to determine if an error occurred. When a write error occurs, `puts` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `write` function for the possible values that `puts` uses when setting these variables.

Return Value

The `puts` function returns the value 0 if the string is successfully written.

If a write error occurs, the `puts` function returns a nonzero value, specifying the appropriate operating-system error code.

Examples

The following program uses `puts` to write two strings to the terminal's screen.

```
#include <stdio.h>

char s1[] = "This short program";
char s2[] = "has written two strings.";

main()
{
   puts(s1);
   puts(s2);
}
```

The output from the preceding program is as follows:

```
This short program
has written two strings.
```

Related Functions

`fgets`, `fputs`, `gets`

putw

Purpose

Writes four bytes of data to a file associated with a specified file pointer.

Syntax

```
#include <stdio.h>

int putw(int w, FILE *file_ptr);
```

Explanation

The `putw` function writes the four bytes of data given in `w` to the file specified by `file_ptr`. The function then moves the file-position indicator ahead four bytes in the file. The `file_ptr` is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

The `putw` function is most often used with files that are opened in binary mode. The size of the "word" that `putw` writes is the size of an `int`, which can vary from machine to machine. In VOS C, `putw` writes four bytes, highest order byte first.

> **Note:** Because of differences in byte ordering and integer size, the data read by `getw` or written by `putw` may not be the same on an operating system other than VOS.

Return Value

If an error does not occur during the write operation, `putw` returns the value 0.

If an error does occur during the write operation, `putw` returns a nonzero value.

Examples

The following program uses `putw` to write an `int` value into a file opened in binary mode.

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

FILE *file_ptr;
int number, w = 255;

int ret_value;
```

(Continued on next page)

```
main()
{
   errno = 0;
   if ( (file_ptr = fopen("ex_file", "r+b")) == NULL )
      {
      printf("Error opening ex_file:  errno = %d\n", errno);
      exit(1);
      }

   if ( putw(w, file_ptr) )
      puts("Write error occurred.");

   rewind(file_ptr);

   if ( (number = getw(file_ptr)) == EOF )
      {
      if ( feof(file_ptr) )
         puts("End of file has been encountered.");

      if ( ferror(file_ptr) )
         puts("Read error occurred.");
      }

   printf("The 4-byte value is %d\n", number);
}
```

Related Functions

`getw`

qsort

Purpose

Sorts the elements of an array.

Syntax

```
#include <stdlib.h>

void qsort( void *pbase, size_t n_elem, size_t e_width,
        int (*compare)(const void *element_1, const void *element_2)
);
```

Explanation

The `qsort` function sorts the elements of an array. The `qsort` function's first three arguments are as follows:

- `pbase` is a pointer to the beginning address of the array to be sorted.
- `n_elem`, an argument of the type `size_t`, specifies the number of elements in the array.
- `e_width`, also an argument of the type `size_t`, specifies the size (in bytes) of each element.

The `size_t` type is a type definition declared in the `stddef.h` header file.

The fourth argument, `compare`, is a pointer to a programmer-defined comparison function that returns an `int`. To sort the array's elements, `qsort` calls the function pointed to by `compare` and, in each call, passes two pointers as arguments. The comparison function should accept two parameters, the pointers `element_1` and `element_2`. Each of these parameters points to an element of the array to be sorted. The comparison function can compare the two elements based on a programmer-designed algorithm, or it can call another library function, such as `strcmp`, to perform the comparison.

The comparison function returns a value based on its comparison of the objects pointed to by `element_1` and `element_2`. Based on the results of the comparison, your function should return an integer that is greater than 0, less than 0, or equal to 0. See the following table.

Return Value	**Result of the Comparison**
Greater than 0	The first element is greater than the second element.
Less than 0	The first element is less than the second element.
Equal to 0	The two elements compare equal.

Elements that compare equal can appear in any order in the final array. The comparison function should not sort the elements by a secondary key and then by a primary key. The comparison function should take **all** ordering constraints into consideration in a single pass.

When specifying the `compare` argument in the `qsort` call, be aware that pointers to functions that have different parameter-type information are different types. Therefore, unless the `compare` function pointer points to a function having two parameters of the type `const void *`, the compiler, with the `-type_checking` argument or the corresponding `#pragma` specified as at least `minimum`, generates the following warning:

```
WARNING 2053 SEVERITY 1 BEGINNING ON LINE num
Implicit conversion has occurred from one pointer to another
which locates a different object.
```

In most cases, the implicit conversion is from pointer to some other `const`-qualified type into "pointer to `const void`." This conversion yields correct results. If you want to avoid the preceding warning, use the following cast before the `compare` function-pointer argument in the `qsort` function invocation.

```
(int (*)(const void *, const void *))
```

See the `qsort` function invocation in Examples for an illustration of how to use this cast.

Return Value

The `qsort` function returns no value.

Examples

The following program uses `qsort` to sort an array of three pointers based on the values of the pointed-to character-string literals.

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

char *ptr[3];

size_t num, width;
int i;

int compare_func(const char **p1, const char **p2);

main()
{
   ptr[0] = "zzz";
   ptr[1] = "aaa";
   ptr[2] = "kkk";

   num = 3;
   width = sizeof(char *);

   qsort(ptr, num, width, (int (*)(const void *, const void
*))compare_func);

   for (i = 0; i < 3; i++)
      printf("ptr[%i] points to %s\n", i, ptr[i]);
}

int compare_func(const char **p1, const char **p2)
{
   return ( strcmp(*p1, *p2) );
}
```

The output from the preceding program is as follows:

```
ptr[0] points to aaa
ptr[1] points to kkk
ptr[2] points to zzz
```

Related Functions

`strcmp`, `strncmp`

rand

Purpose

Generates a pseudo-random integer in the range 0 through `RAND_MAX`.

Syntax

```
#include <stdlib.h>

int rand(void);
```

Explanation

The `rand` function generates a pseudo-random integer in the range 0 through `RAND_MAX`. The `RAND_MAX` constant, defined in the `stdlib.h` header file, has the value 32,767.

When `rand` is invoked multiple times, the function generates a reproducible sequence of pseudo-random numbers. After program startup, the `rand` function always returns the same sequence of numbers unless, before calling `rand`, you first call the `srand` function to change the seed used to generate the values. If `rand` is invoked without first calling `srand`, the same sequence of numbers is generated as when `srand` is first called with a seed value of 1.

Note: The sequence of numbers returned by `rand` or the value of `RAND_MAX` or both may change in future releases of the library functions.

Return Value

The `rand` function returns a pseudo-random number.

Examples

The following program uses `rand` to generate a sequence of five pseudo-random numbers. The numbers generated by `rand` are displayed on the terminal's screen.

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

main()
{
   int i, random_number;
```

(Continued on next page)

```
         for (i = 0; i < 5; i++)
            {
            random_number = rand();
            printf("Random number #%d = %d\n", (i + 1), random_number);
            }
      }
```

The output from the preceding program is as follows:

```
Random number #1 = 10523
Random number #2 = 11866
Random number #3 = 690
Random number #4 = 13340
Random number #5 = 30587
```

Related Functions

`srand`

`read`

Purpose

Reads a specified number of bytes of data from a file associated with a given file descriptor.

Syntax

```
#include <c_utilities.h>

int read(int file_des, char *buffer, unsigned int nbytes);
```

Explanation

The `read` function reads `nbytes` of data from the file specified by `file_des` into the memory area pointed to by `buffer`. The `file_des` argument is a file descriptor returned by a previous call to the `fileno` or `open` function.

The data is read starting at the file's current position. After `read` reads data from the file, it increments the file-position indicator, associated with the file, by the number of bytes actually read. The `read` function guarantees that the number of bytes specified will be read as long as end-of-file is not reached before that number of bytes is read.

If an error occurs, `read` sets the external variables `errno` and `os_errno` to the error code number returned by the operating system. The following table lists some of the error code values that are possible.

Error Code Name	**Number**
`e$file_modified_during_reads`	`1067`
`e$form_requested`	`1225`
`e$invalid_file_pointer`	`3929`
`e$invalid_io_operation`	`1040`
`e$no_alloc_user_heap`	`3080`
`e$partial_conversion`	`4180`
`e$record_format_error`	`4265`

The `read` function never sets `errno` or `os_errno` to `e$end_of_file` (`1025`). You can tell when end-of-file has been reached by examining `errno` after the return value of `read` is less than was specified in the `nbytes` argument. In such a case, if `errno` equals the value 0, end-of-file has been reached before `nbytes` could be read.

Return Value

The `read` function returns the number of bytes actually read from the file and written into the memory area. If the file-position indicator is at end-of-file before `read` is called, the function returns the value 0.

If an error occurs, `read` returns the value -1.

Examples

The following program uses `read` to read a specified number of bytes of data from a file associated with `file_des_1`. It stores the characters that it has read in an array, whose address is `buffer`. The program then copies the data from `buffer` into another file, `out_file`.

```
#include <c_utilities.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int file_des_1, file_des_2, mode;
int chars_read, chars_written;
char buffer[10000];

main()
{
   if ( (file_des_1 = open("in_file", O_RDONLY, mode)) == -1)
      {
      printf("Error opening in_file.\n");
      exit(1);
      }

   if ( (file_des_2 = open("out_file", O_WRONLY | O_APPEND, mode)) == -1)
      {
      printf("Error opening out_file.\n");
      exit(1);
      }

   errno = 0;
   chars_read = read(file_des_1, buffer, 10000);
   if (chars_read == -1)
     printf("Error occurred during read operation:  errno = %d\n", errno);
```

(Continued on next page)

```
   errno = 0;
   chars_written = write(file_des_2, buffer, chars_read);
   if (chars_written == -1)
      printf("Write error occurred:  errno = %d\n", errno);
   else
      printf("%d bytes have been written to out_file\n", chars_written);

   close(file_des_1);

   close(file_des_2);
}
```

Related Functions

`close`, `creat`, `fread`, `lseek`, `open`, `write`

realloc

Purpose

Changes the size of a block of memory previously allocated with `calloc`, `malloc`, or `realloc`.

Syntax

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

Explanation

The `realloc` function changes the size of a block of memory pointed to by `ptr` to the size specified by `size`.

The `ptr` argument can be the `NULL` pointer constant, or an address returned by a previous call to `calloc`, `malloc`, or `realloc`. When `ptr` equals `NULL`, `realloc` behaves exactly like `malloc` and returns a pointer to a newly allocated block of memory of the size given in `size`. If `ptr` is not equal to `NULL` or to an address returned by a previous call to `calloc`, `malloc`, or `realloc`, the reallocation operation fails: `realloc` returns the `NULL` pointer constant, and the object pointed to by `ptr` is unchanged. If `ptr` locates the address of memory that has been freed by `free`, the behavior is unpredictable.

The `size` argument specifies the number of bytes of storage to allocate. The `size` argument is a value of the type `size_t`, which is a type definition declared in the `stddef.h` header file.

- If the `size` specified in `realloc` is larger than the existing object's size, the `realloc` function copies the contents of the existing object into the initial portion of the newly allocated object. The remainder of the newly allocated object has unpredictable initial values.

- If the `size` specified in `realloc` is not larger than the existing object's size, the `realloc` function copies only `size` bytes of the initial part of the existing object into the newly allocated object.

When `ptr` is not `NULL` and `size` equals 0, the pointed-to memory is deallocated.

If `realloc` is successful, it returns a pointer to the first byte of allocated memory. This pointer is a "generic pointer" declared as a "pointer to `void`." A pointer to `void` can be assigned, without a cast, to a pointer to any object type. Nevertheless, many programmers cast the return type of `realloc` so that the type of the returned pointer is explicitly converted to the appropriate data type. See the “Pointers to Void” section in Chapter 4 for more information on that pointer type.

The `realloc` function allows you to allocate storage dynamically at run time. The amount of memory a program uses can be determined by run-time events, such as user input or the amount of data in a file.

The only way that you can access the contents of the allocated memory is indirectly through the pointer returned by `realloc`. Therefore, assign the returned pointer to a pointer variable of the appropriate type. Then, you can specify the pointer variable itself when the object’s address is needed, or you can use an indirection operation when the contents at that address are needed.

The space allocated with `realloc` begins on a boundary that is suitably aligned for all Stratus processors.

Return Value

If `realloc` successfully allocates memory of the specified size, the function returns a pointer to the first byte of allocated memory. The newly allocated block of memory may not be located at the same address as the previously allocated block.

If `realloc` does not successfully allocate memory of the specified size, the function returns the `NULL` pointer constant, and the object pointed to by `ptr` is unchanged. The `realloc` function can fail, for example, if the amount of memory specified in `size` is not free.

If `ptr` equals `NULL` and `size` equals 0, `realloc` returns a valid pointer whose pointed-to storage should not be accessed.

Examples

The following program uses `realloc` to change the size of a block of memory previously allocated with `calloc`.

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

struct item
   {
   char name[100];
   double amount;
   };

struct item *ptr;

int orig_size, new_size;
```

(Continued on next page)

```
main()
{
   /*  Use calloc to allocate memory for five structures  */

   orig_size = 5;
   ptr = (struct item *)calloc( orig_size, sizeof(struct item) );

   if (ptr == NULL)
      {
      puts("Memory was not successfully allocated.");
      exit(1);
      }

   /*  Use realloc to change the size of the memory allocated  */
   /*  so that there is space for five additional structures   */

   new_size = 10;

   ptr = (struct item *)realloc( ptr, (10 * sizeof(struct item)) );

   if (ptr == NULL)
      {
      puts("realloc failed to allocate the memory specified.");
      exit(1);
      }
}
```

Related Functions

`alloca`, `calloc`, `free`, `malloc`

`remove`

Purpose

Deletes a specified file.

Syntax

```
#include <stdio.h>

int remove(const char *filename);
```

Explanation

The `remove` function deletes a file whose path name is the string pointed to by `filename`. The string should be the full or relative path name of a file that is closed.

If `remove` cannot delete the file, the function sets the external variables `errno` and `os_errno` to the appropriate error code number. For example, if the file is open, `remove` sets `errno` and `os_errno` to the error code `e$file_in_use` (`1084`).

Return Value

The `remove` function returns the value 0 if it successfully deletes the file.

If `remove` cannot delete the file, `remove` returns a nonzero value, specifying the appropriate operating-system error code.

Examples

The following program uses `remove` to delete a file whose path name is entered on the command line.

```
#include <stdio.h>

main(int argc, char *argv[])
{
   int return_value = 0;

   if (argc == 1)
      puts("Enter a file name on the command line.");
```

(Continued on next page)

```
         else
            {
            errno = 0;
            return_value = remove(argv[1]);
            if (return_value != 0)
               {
               puts("File could not be deleted.");
               printf("errno = %d\n", errno);
               }
            }
      }
```

Related Functions

`rename`, `tmpfile`, `tmpnam`

`rename`

Purpose

Changes the name of a specified file.

Syntax

```
#include <stdio.h>

int rename(const char *old_name, const char *new_name);
```

Explanation

The `rename` function changes the name of a file specified in the string pointed to by `old_name` into the name specified in the string pointed to by `new_name`. You cannot use `rename` to change the file's location.

The string pointed to by `old_name` can be a full or relative path name. The string pointed to by `new_name` **must** be a file name, not a path name. The new file name should not be the name of an existing file, directory, or link in the directory containing the file to be renamed. When the new file name exceeds 32 characters, the rightmost characters beyond 32 are truncated.

If `rename` cannot change the file's name, the function sets the external variables `errno` and `os_errno` to the appropriate error code number. For example, if the name specified by `new_name` names an existing file in the containing directory, `rename` sets `errno` and `os_errno` to the error code `e$file_exists` (`1050`).

Return Value

The `rename` function returns the value 0 if it successfully changes the file's name.

If `rename` cannot change the file's name, it returns a nonzero value, specifying the appropriate operating-system error code.

Examples

The following program uses `rename` to change a file's name. Both the old and new names are entered on the command line.

```
#include <stdio.h>

main(int argc, char *argv[])
{
   int return_value = 0;

   if (argc < 3)
      puts("Enter on the command line:  rename old_name new_name");
   else
      {
      errno = 0;
      return_value = rename(argv[1], argv[2]);
      if (return_value != 0)
         {
         puts("File could not be renamed.");
         printf("errno = %d\n", errno);
         }
      }
}
```

Related Functions

`remove`, `tmpfile`, `tmpnam`

`rewind`

Purpose

Sets the file-position indicator, for a file associated with a specified file pointer, to the beginning of the file.

Syntax

```
#include <stdio.h>

void rewind(FILE *file_ptr);
```

Explanation

The `rewind` function sets the file-position indicator to the beginning of the file. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`. A call to the `rewind` function is similar to the following `fseek` call:

```
fseek(file_ptr, 0, SEEK_SET);
```

Both the `rewind` and `fseek` functions clear the end-of-file indicator for the specified file. However, only the `rewind` function also clears the file's error indicator.

There are certain limitations when using `rewind` with a sequential file. Seeking on a sequential file is limited to a file that has been opened for reading only: either input mode or dirty-input mode.

If an error occurs during the seek operation, `rewind` sets the external variables `errno` and `os_errno` to the appropriate error code number. See the `lseek` function for the possible values that `rewind` uses when setting these variables.

Return Value

The `rewind` function returns no value.

Examples

The following program uses `rewind` to set the file-position indicator for a file, `in_file`, to the beginning of the file after displaying the contents of the file and reaching the end of the file.

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

main()
{
   int c;
   FILE *file_ptr;

   if ( (file_ptr = fopen("in_file", "r")) == NULL )
      {
      printf("Could not open in_file\n");
      exit(1);
      }

   while ( (c = fgetc(file_ptr)) != EOF )
      putchar(c);
   putchar('\n');

   rewind(file_ptr);

   printf("The file-position indicator = %d\n", ftell(file_ptr));
}
```

If `in_file` contained `aaaa 1234`, the output from the preceding program would be as follows:

```
aaaa 1234
The file-position indicator = 0
```

Related Functions

`fseek`

s$c_get_portid

Purpose

Returns the port ID associated with a file pointer.

Syntax

```
#include <stdio.h>

int s$c_get_portid(FILE *file_ptr);
```

Explanation

The `s$c_get_portid` function returns the port ID associated with the file specified by `file_ptr`. The `file_ptr` argument is a pointer returned, for example, from a previous call to `fdopen`, `fopen`, or `freopen`.

In the VOS environment, when a file is opened (for example, with `fopen`), a port is attached to the file. A *port ID* is a 2-byte integer used to identify a port.

If `file_ptr` equals `NULL` or `OS_NULL_PTR`, `s$c_get_portid` sets the external variables `errno` and `os_errno` to `e$invalid_file_ptr` (`3929`).

Return Value

The `s$c_get_portid` function returns a port ID.

If `file_ptr` equals `NULL` or `OS_NULL_PTR`, `s$c_get_portid` returns the value -1.

Examples

The following program fragment uses `s$c_get_portid` to get the port ID associated with the file specified by `file_ptr`.

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

int port_id;

main()
{
   FILE *file_ptr = NULL;
```

(Continued on next page)

```
          if ( (file_ptr = fopen("text_file", "r")) == NULL )
             {
             printf("Error opening text_file.\n");
             exit(1);
             }

          if ( (port_id = s$c_get_portid(file_ptr)) == -1 )
             {
             printf("Error getting port id.\n");
             exit(1);
             }
             .
             .
             .
       }
```

Related Functions

```
s$c_get_portid_from_fildes
```

s$c_get_portid_from_fildes

Purpose

Returns the port ID associated with a file descriptor.

Syntax

```
#include <stdio.h>

int s$c_get_portid_from_fildes(int file_des);
```

Explanation

The `s$c_get_portid_from_fildes` function returns the port ID associated with the file specified by `file_des`. The `file_des` argument is a file descriptor returned, for example, from a previous call to `creat`, `fileno`, or `open`.

In the VOS environment, when a file is opened (for example, with `open`), a port is attached to the file. A *port ID* is a 2-byte integer used to identify a port.

Return Value

The `s$c_get_portid_from_fildes` function returns a port ID.

If `file_des` is invalid, `s$c_get_portid_from_fildes` returns the value -1.

Examples

The following program fragment uses `s$c_get_portid_from_fildes` to get the port ID associated with the file specified by `file_des`.

```
#include <stdio.h>
#include <stdlib.h>
#include <c_utilities.h>
#include <fcntl.h>

int file_des, mode, port_id;

main()
{
   if ( (file_des = open("in_file", O_RDONLY, mode)) == -1 )
      {
      printf("Error opening in_file.\n");
      exit(1);
      }
```

(Continued on next page)

```
     if ( (port_id = s$c_get_portid_from_fildes(file_des)) == -1 )
        {
        printf("Error getting port id.\n");
        exit(1);
        }
        .
        .
        .
  }
```

Related Functions

`s$c_get_portid`

scanf

Purpose

Using a specified format, reads a sequence of characters from the standard input file, `stdin`.

Syntax

```
#include <stdio.h>

int scanf(const char *format, ...);
```

Explanation

The `scanf` function reads input from the preopened file pointed to by `stdin` and converts the input sequence of characters using the directives specified in the string pointed to by `format`. The `scanf` function assigns each converted result to an object pointed to by the corresponding argument in the argument list. The file pointed to by `stdin` is usually associated with the terminal's keyboard.

The string pointed to by `format` consists of zero or more directives. The directives consist of the following:

- One or more white-space characters cause `scanf` to read up to the next non-white-space character, which remains unread, or until it can read no more characters. (A newline character is a white-space character.)
- Ordinary non-white-space characters (not `%`) cause `scanf` to read the next characters from the file. When `scanf` encounters a nonmatching character, the directive fails, and the nonmatching and subsequent characters remain unread.
- A *conversion specification* defines a set of input sequences that `scanf` attempts to match. The conversion specifications are described later in the Explanation.

The `scanf` function executes each directive in turn. If a directive fails, the `scanf` function returns. A directive can fail, for example, if no input characters are available or if the input characters do not match those specified by the directive.

Typically, each conversion specification causes `scanf` to read an input item from the file, to convert the input item to the type specified by the conversion specifier, and to assign the converted result to the object pointed to by the corresponding argument. The following example shows a `scanf` call consisting of a `%d` conversion specification and a corresponding argument having the type pointer to `int`.

```
int i_num;

scanf("%d", &i_num);
```

In the preceding `scanf` call, `scanf` executes the `%d` directive by reading characters from `stdin`, converting the characters that match the `%d` format to an `int` value, and assigning the converted result to the object located by the expression `&i_num`. If the input sequence of characters were `99abcde`, `scanf` would assign the value 99 to `i_num`.

In a `scanf` call, the syntax for the argument list is as follows:

```
scanf(format, [argument]...)
```

Each *`argument`* must be an expression yielding a pointer to an object type appropriate for the conversion specification. The number and types of the arguments in a `scanf` call depend on the conversion specifications given in the format string.

- If the pointed-to object does not have the appropriate type, or if the result of the conversion cannot be represented in the pointed-to object, `scanf` yields unpredictable results.
- If there are not enough arguments for the format, the behavior is unpredictable.
- If the format is exhausted while arguments remain, the excess arguments are evaluated, as are all function arguments, but are otherwise ignored.

Conversion Specifications. The format string typically includes one or more conversion specifications. Each conversion specification describes how `scanf` is to interpret a part of the input sequence of characters. The format for a conversion specification is as follows:

```
%[*][width][size] conversion_specifier
```

After the beginning `%` character, the parts of the conversion specification must appear in the sequence shown in the preceding format.

An optional assignment-suppressing character `*` indicates that the input item is to be read according to the conversion specification, but the result of the conversion is not to be stored. Do **not** give a corresponding argument when you specify assignment suppression. The suppression of assignment provides a way of describing an input item that is to be skipped.

An optional *`width`* is a nonzero decimal integer that indicates the field width: the maximum number of characters to be read for an input item. Except for the `[`, `c`, and `n` specifiers, all conversion specifiers skip any leading white-space characters. These white-space characters are not counted as part of the item's field width.

An optional *`size`* can appear before the conversion specifier, indicating the size of the receiving object. The *`size`* modifies the *`conversion_specifier`* and **must** appear in a conversion specification if the receiving field is one of the following:

- a pointer to `short int` or a pointer to `long int`
- a pointer to `unsigned short int` or a pointer to `unsigned long int`
- a pointer to `double`.

Table 11-35 explains each of the characters allowed for *`size`*. If a size character (`h` or `l`) appears with any conversion specifier other than as shown in the table, the behavior is unpredictable. See Examples, later in this description, for some sample uses of the *`size`* character in a conversion specification.

Table 11-35. Formatted Input: Size in a Conversion Specification

Size	Meaning
`h`	If the conversion specifier is preceded by `h`, the type of the corresponding argument is as follows: For a `d`, `i`, or `n` conversion, the argument is a pointer to `short int` rather than a pointer to `int`. For an `o`, `u`, `x`, or `X` conversion, the argument is a pointer to `unsigned short int` rather than a pointer to `unsigned int`.
`l`	If the conversion specifier is preceded by `l`, the type of the corresponding argument is as follows: For a `d`, `i`, or `n` conversion, the argument is a pointer to `long int` rather than a pointer to `int`. For an `o`, `u`, `x`, or `X` conversion, the argument is a pointer to `unsigned long int` rather than a pointer to `unsigned int`. For an `e`, `E`, `f`, `g`, or `G` conversion, the argument is a pointer to `double` rather than a pointer to `float`.

Conversion Specifiers. The required *`conversion_specifier`* can be one of several characters. The following paragraphs provide a full explanation of the input item expected by and the argument type that corresponds to each of the conversion specifiers. Each conversion specifier can be used with the following formatted input functions: `fscanf`, `scanf`, and `sscanf`.

`c` The expected input item is a sequence of characters of the number specified by *`width`* or one character if *`width`* is not specified. The corresponding argument should be a pointer to `char` or, if the field width is greater than 1, to a character array large enough to store the sequence. Input white-space characters are **not** skipped. No null character is appended. To read the next non-white-space character, use `% 1s`.

d The expected input item is an optionally signed decimal integer. The corresponding argument should be a pointer to int. The input item has the same format as the subject sequence of the strtol function when it is called with the base argument equal to 10.

e, E, f, g, GThe expected input item is an optionally signed floating-point number. The corresponding argument should be a pointer to float. The input item has the same format as the subject sequence of the strtod function.

i The expected input item is an optionally signed integer in decimal, hexadecimal, or octal format. The corresponding argument should be a pointer to int. The input item has the same format as the subject sequence of the strtol function when it is called with the base argument equal to the value 0.

n No input item is expected or read. The corresponding argument is a pointer to int into which is **written** the number of characters read from the input file so far by this call to scanf. Execution of %n does not increment the assignment count returned at the completion of the scanf call.

o The expected input item is an optionally signed octal integer. The corresponding argument is a pointer to unsigned int. The input item has the same format as the subject sequence of the strtoul function when it is called with the base argument equal to the value 8.

p The expected input item is an eight-digit absolute address in hexadecimal format (for example, 00000000 or 0x00000000). The corresponding argument is a pointer to a pointer to void. The input item should be the same as is produced by the %p conversion of the fprintf function.

- If the input item is a value converted earlier during the same program execution, the pointer value that results compares equal to that value.
- If the input item is **not** a value converted earlier during the same program execution, the pointer value that results is unpredictable.

s The expected input item is a sequence of non-white-space characters. The input field is delimited by a white-space character. The corresponding argument is a pointer to the first character of an array of char of sufficient length to store the sequence of characters plus a terminating null character. The scanf function automatically appends a null character to the sequence of characters.

u The expected input item is an optionally signed decimal integer. The corresponding argument is a pointer to unsigned int. The input item has the same format as the subject sequence of the strtoul function when it is called with the base argument equal to the value 10.

v The expected input item is a char_varying string. The corresponding argument is a pointer to char_varying(*n*) where *n* specifies an object of sufficient length to store the input item. The input field is delimited by a white-space character. The sequence of characters read equals the current length of the input string unless the number of characters specified by the field width is read before the string's last character is reached. This conversion specifier is a VOS C extension and is not portable.

`x`, `X` The expected input item is an optionally signed hexadecimal integer. The corresponding argument is a pointer to `unsigned int`. The input item has the same format as the subject sequence of the `strtoul` function when it is called with the `base` argument equal to the value 16.

`%` The expected input item is a single `%`. No conversion or assignment is performed. To read a `%` character, the complete conversion specification should be `%%`. In VOS C, an assignment-suppressing character, width, or size is accepted and ignored.

If a conversion specification is invalid, the behavior is unpredictable.

Scansets. In addition to the preceding conversion specifiers, you can use a scanset to store, in an array, one or more specified characters from the input sequence of characters. A *scanset* is a set of expected characters, the characters from the input sequence that `scanf` is to store in the array. With a scanset, `scanf` stops reading and storing characters when it encounters a character not in the scanset or when it reads the number of characters specified by the field width, whichever occurs first. Input white-space characters are not skipped.

With a scanset, the corresponding argument is a pointer to the first of an array of `char` of sufficient length to store the sequence of characters plus a terminating null character. The `scanf` function automatically appends a null character to the sequence of characters.

A scanset has the following form:

```
[scanlist]
```

A scanset begins with an opening bracket (`[`). The conversion specification includes all subsequent characters in the format string up to and including the matching right bracket (`]`). The characters between the brackets (the *`scanlist`*) comprise the scanset. The following example shows a scanset that includes all of the characters that can appear in a hexadecimal number.

```
[0123456789abcdefABCDEF]
```

If the character after the opening bracket is a circumflex (^), the list of characters to match consists of all characters that do **not** appear in the list between the circumflex and the right bracket. The following example shows a scanset that includes all characters except the lowercase and uppercase vowels.

```
[^aeiouAEIOU]
```

If you want to specify a `]` character in the scanlist, the `]` character must appear immediately after the left bracket (`[]`) or immediately after the circumflex, if present (`[^]`). In these cases, the next right bracket is the matching right bracket that ends the conversion specification. If the `]` character appears in any other position, it is the one that ends the conversion specification.

In VOS C, a range of characters can be specified in the scanlist with the construct *`first-last`*. Thus, you can specify `[0123456789]` with `[0-9]`. Using this convention, *`first`* must be lexically less than or equal to *`last`*; otherwise, the `-` character will stand for itself. The `-` character will also stand for itself whenever it appears in the scanlist as the first character, as the second character where the first character is `^`, or as the last character.

Table 11-36 provides a summary of the formatted input conversion specifiers. For each specifier, the table shows the corresponding argument's type and a description of the expected input item.

Table 11-36. Formatted Input: Conversion Specifier Summary

Conversion Specifier	Argument Type	Expected Input Item
`c`	`char *`	A single character or the number of characters specified in *`width`*. White-space characters are not skipped.
`d`	`int *`	An optionally signed decimal integer.
`e, E, f, g, G`	`float *`	An optionally signed floating-point number.
`i`	`int *`	An optionally signed integer in decimal, hexadecimal, or octal format.
`n`	`int *`	The number of characters read so far by this `scanf` call is written into the pointed-to `int` variable.
`o`	`unsigned int *`	An optionally signed octal integer.
`p`	`void **`	An eight-digit absolute address in hexadecimal format, such as `0000000a`.
`s`	`char *`	A sequence of non-white-space characters delimited by the first white-space character. A terminating null character is appended to the sequence of characters.
`u`	`unsigned int *`	An optionally signed decimal integer.
`v`	`char_varying(`*`n`*`) *`	A varying-length character string.
`x, X`	`unsigned int *`	An optionally signed hexadecimal integer.
`%%`	none	Matches a percent sign.
`[`*`scanlist`*`]`	`char *`	A sequence of characters delimited by the first character not in the *`scanlist`*. A terminating null character is appended to the sequence of characters.

A formatted input function, like `scanf`, executes each conversion specification in the following manner.

First, the `scanf` function skips over input white-space characters unless the conversion specification contains the `[`, `c`, or `n` specifier. These white-space characters are not counted as part of the field width.

Second, the `scanf` function reads an input item from the file. An *input item* is the longest sequence of input characters that match those specified by the conversion specifier, unless the longest matching sequence exceeds the field width. In this case, the input item is the initial subsequence of matching input characters that equals the length indicated in the field width. The first character, if any, after the input item remains unread.

Third, except with the `%` specifier, the `scanf` function converts the input item to the type appropriate to the conversion specifier. With the `n` specifier, the count of input characters is converted to the appropriate type.

Lastly, unless the assignment-suppression character (`*`) is specified, the `scanf` function assigns the converted result to the object pointed to by the first argument following the `format` argument that has not already been assigned a result.

If a formatted input function, like `scanf`, encounters end-of-file during input, it ends the conversion.

- When the function encounters end-of-file **before** any characters matching the current directive have been read (other than leading white-space characters, where permitted), execution of the current directive fails because no matching input characters are available.
- When the function encounters end-of-file **after** characters matching the current directive have been read, execution of the next directive fails because no input characters are available.

If `scanf` terminates conversion because it has read a conflicting input character, the conflicting character is left unread in the input stream. Trailing white-space characters (including newline characters) are left unread unless matched by a directive.

The only method you can use to determine whether a literal match or a suppressed assignment has succeeded or failed is to use the `%n` conversion specification.

Table 11-37 provides a sample invocation of `scanf` with each conversion specifier. For each specifier, the table shows an input sequence and describes how `scanf` stores the input item.

Table 11-37. Formatted Input: Sample Conversion Specifications *(Page 1 of 2)*

Function Invocation	**Argument Declaration**	**Input Sequence**	**Effect**
`scanf("%c", &ch)`	`char ch;`	`128e2`	Stores the character `'1'` in `ch`.
`scanf("%d", &i)`	`int i;`	`128e2`	Stores the value 128 in `i`.
`scanf("%e", &f)`	`float f;`	`128e2`	Stores the value 12800.0 in `f`.

Table 11-37. Formatted Input: Sample Conversion Specifications *(Page 2 of 2)*

Function Invocation	Argument Declaration	Input Sequence	Effect
`scanf("%i", &i)`	`int i;`	`128e2`	Stores the value 128 in `i`.
`scanf("%o", &u)`	`unsigned int u;`	`128e2`	Stores the value 12 octal in `u`.
`scanf("%p", &p)`	`void *p;`	`0000000a`	Stores the value 0000000a hex. in `p`.
`scanf("%s", a)`	`char a[10];`	`abcd efg`	Stores the string `"abcd"` in `a`.
`scanf("%u", &u)`	`unsigned int u;`	`128e2`	Stores the value 128 in `u`.
`scanf("%v", &cv)`	`char_varying(5) cv;`	`abcd efg`	Stores the characters `abcd` in `cv`.
`scanf("%X", &u)`	`unsigned int u;`	`00ff`	Stores the value ff hex. in `u`.
`scanf("%%")`	none	`%`	Matches a `%` character.
`scanf("%[12345]", &a)`	`char a[10];`	`128e2`	Stores the string `"12"` in `a`.

Return Value

The `scanf` function returns the number of input items assigned to arguments. The number of input items assigned can be fewer than expected, or zero, depending on the number of input characters that match those specified by the format string's directives.

If end of input occurs before any successful conversion, including those with suppressed assignments, `scanf` returns `EOF`.

Examples

The following program uses `scanf` to read two lines of input. The `scanf` function converts each input item and assigns the result to the corresponding argument.

```
#include <stdio.h>

int items;

int i_num;
char array_1[15];
double d_num;

short s_num;
int count;
```

(Continued on next page)

```
main()
{
  items = scanf("%d %s %lf", &i_num, array_1, &d_num);     /*  Example
1  */
   printf("Example 1:  items = %d\n", items);
   printf("i_num = %d; array_i = %s; d_num = %lf\n", i_num, array_1,
d_num);

   getchar();  /*  Reads the unread newline  */

  items = scanf("%3hd %n", &s_num, &count);               /*  Example
2  */
   printf("\nExample 2:  items = %d\n", items);
   printf("s_num = %hd; count = %d\n", s_num, count);
}
```

Assume that the following two lines of input were entered at the terminal's keyboard.

```
9876 johnny 543.21
9876
```

With this input, the output from the preceding program would be as follows:

```
Example 1:  items = 3
i_num = 9876; array_i = johnny; d_num = 543.210000

Example 2:  items = 2
s_num = 987; count = 3
```

Related Functions

`fscanf`, `sscanf`

setbuf

Purpose

Causes a specified buffer, instead of a buffer allocated by the system, to be used for buffered I/O on a file associated with a given file pointer.

Syntax

```
#include <stdio.h>

void setbuf(FILE *file_ptr, char *buffer);
```

Explanation

The `setbuf` function causes the buffer pointed to by `buffer` to be used for buffered I/O on the file specified by `file_ptr`. If you do not use `setbuf` or `setvbuf` to specify such a buffer for the file, the system automatically allocates the buffer that the system uses for buffered I/O on the file. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`. You can call `setbuf` only after the file has been opened and before any I/O operation is performed on the file.

When `buffer` is not the `NULL` pointer constant, you should declare the buffer pointed to by `buffer` as an array of `char` having exactly `BUFSIZ` elements. `BUFSIZ` is a constant defined in the `stdio.h` header file.

When `buffer` is not the `NULL` pointer constant, a call to `setbuf` is equivalent to the following `setvbuf` call:

```
setvbuf(file_ptr, buffer, _IOFBF, BUFSIZ);
```

The buffer pointed to by `buffer` **must** have a lifetime at least as long as the associated stream (for example, until the file is closed). If the buffer has automatic storage duration, the file must be closed before the buffer is deallocated upon block exit. **The contents of the array specified by** `buffer` are at any time indeterminate.

When `buffer` is the `NULL` pointer constant, `setbuf` causes I/O on the specified file to be unbuffered.

Return Value

The `setbuf` function returns no value.

Examples

The following program uses `setbuf` to change a file's I/O buffer to a program-specified buffer and then displays data from that buffer.

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

FILE *file_ptr;
char buffer[BUFSIZ];

main()
{
   if ( (file_ptr = fopen("ex_file", "r")) == NULL )
      {
      puts("Error occurred opening ex_file.");
      exit(1);
      }

   setbuf(file_ptr, buffer);

  fgetc(file_ptr);           /*  Perform some I/O to fill the buffer
*/

   printf("%20s\n", buffer);
}
```

Related Functions

`setvbuf`

setjmp

Purpose

Saves its calling environment for later use by the `longjmp` function.

Syntax

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

Explanation

The `setjmp` function saves its calling environment, in the `env` argument, for later use by the `longjmp` function. The `env` argument is of the type `jmp_buf`, which is a type defined in the `setjmp.h` header file.

The `setjmp` and `longjmp` functions are used together to achieve the effect of a nonlocal `goto` statement. First, you call `setjmp` to save the calling environment at the point where `setjmp` was invoked. Second, in some other function, you call `longjmp` to restore the calling environment saved in `env` and to jump to the return address also saved in `env`. In effect, program control is transferred back to the location of the most recent invocation of `setjmp` that used `env` as its argument. When `longjmp` returns, it acts like a return from a call to `setjmp` with a nonzero return value.

An invocation of `setjmp` should appear only as one of the following:

- the entire expression of an expression statement (possibly cast to `void`)
- the entire controlling expression of a `do`, `for`, `if`, `switch`, or `while` statement
- one operand of a relational or equality operator with the other operand an integral expression, with the resulting expression being the entire controlling expression of a `do`, `for`, `if`, `switch`, or `while` statement
- the operand of the logical negation operator (`!`) with the resulting expression being the entire controlling expression of a `do`, `for`, `if`, `switch`, or `while` statement.

If `setjmp` appears in any other context, the behavior is unpredictable. When a `setjmp` call is the entire expression of an expression statement, it is impossible to distinguish between the return from the `setjmp` call and a transfer of program control, caused by the execution of `longjmp`, back to the location of the `setjmp` call.

Return Value

The `setjmp` function returns the value 0 when the function is called to save the calling environment. After the calling environment has been saved, a subsequent call to `longjmp` will appear to be a return from `setjmp` with a nonzero return value.

Examples

The following program fragment uses `setjmp` and `longjmp` to achieve the effect of a nonlocal `goto` statement. The `setjmp` function saves the calling environment at the point of its invocation in the `main` function. When `longjmp` is invoked in `func`, that function restores the environment saved in `env`. In effect, when `longjmp` is called, program control transfers back to the location of the `setjmp` invocation in `main`.

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

int func(void);

int code;
jmp_buf env;

main()
{
   int return_value;

   puts("Saving this calling environment ....");

   switch ( setjmp(env) )
      {

      case (0):   puts("Returning from setjmp call in main.");
                  return_value = 0;
                  break;

      case (99):  puts("Returning from longjmp call in func.");
                  /*  Special action can be taken if needed  */
                  return_value = 99;
                  exit(return_value);
                  break;

      default:    puts("Returning from who knows where.");
                  return_value = -1;
                  exit(return_value);
                  break;
      }
```

(Continued on next page)

```
                  code = func();
                 .
                 .
                 .
              exit(return_value);
        }

        int func(void)
        {
           int val, okay = 0;
              .
              .
              .
           if (okay == 0)
              {
             puts("Calling longjmp to restore environment saved in env ....");
              val = 99;
              longjmp(env, val);
              }

           return(1);
        }
```

Related Functions

`longjmp`

setvbuf

Purpose

For a file associated with a specified file pointer, changes one or more aspects of buffering: the buffer to be used in place of the system-allocated buffer, the mode of buffering, or the size of the buffer.

Syntax

```
#include <stdio.h>

int setvbuf(FILE *file_ptr, char *buffer, int mode, size_t size);
```

Explanation

The `setvbuf` function changes how the file specified by `file_ptr` is buffered. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`. You can call `setvbuf` only after the file has been opened and before any I/O operation is performed on the file. With `setvbuf`, you can change one or more of the following aspects of buffering for the specified file.

- the buffer to use for buffered I/O
- the mode of buffering
- the size of the buffer.

If the `buffer` argument is not the `NULL` pointer constant, `setvbuf` causes the buffer pointed to by `buffer` to be used for buffered I/O on the specified file. If you do not use `setbuf` or `setvbuf` to specify such a buffer for the file, the system automatically allocates the buffer that the system uses for buffered I/O on the file. If the `buffer` argument is `NULL`, the system-allocated buffer is used for buffered I/O on the file.

If `buffer` is not the `NULL` pointer constant, you should declare the buffer pointed to by `buffer` as an array of `char` having at least the number of elements indicated in `size`. **The contents of the array specified by** `buffer` are at any time indeterminate.

The buffer pointed to by `buffer` **must** have a lifetime at least as long as the associated stream (for example, until the file is closed). If the buffer has automatic storage duration, the file must be closed before the buffer is deallocated upon block exit.

The `mode` argument specifies the buffering method to use for the specified file. The following table lists and explains the modes that `setvbuf` allows. Each `mode` value, such as `_IOFBF`, is a constant defined in the `stdio.h` header file.

Mode	**Buffering Method**
`_IOFBF`	The file is fully buffered, if possible.
`_IOLBF`	The file is line buffered, if possible.
`_IONBF`	The file is unbuffered, if possible.

Not all `mode` values are allowed for all file organizations and device types.

- For a stream, sequential, or relative file that has been opened in binary mode, you can choose fully buffered or unbuffered.
- For a fixed file, you can choose only fully buffered.
- For `stdin` and `stdout`, you can choose line buffered or unbuffered if `stdin` and `stdout` are associated with a terminal device. Otherwise, if `stdin` and `stdout` are not associated with a terminal device, you can choose fully buffered, line buffered, or unbuffered.
- For `stderr`, you can choose line buffered or unbuffered.
- For a terminal device opened in text mode, you can choose line buffered or unbuffered.
- For a terminal device opened in binary mode, you can choose only unbuffered.

See the "Buffered Input and Output" section in Chapter 10 for more information on each buffering method.

The `size` argument specifies the size of the buffer that will be used for buffered I/O on the given file. The `size` argument is a value of the type `size_t`, which is a type definition declared in the `stddef.h` header file. If the mode of buffering is unbuffered (`_IONBF`), `size` must equal 0. If the specified file has fixed file organization, `size` must be a multiple of the record size.

The `setvbuf` function checks that the mode of buffering indicated by `mode` and the buffer size indicated by `size` are appropriate for the given file or device. If you specify an **inappropriate** value for either argument or if `file_ptr` is `NULL`, `setvbuf` fails and sets the external variables `errno` and `os_errno` to the appropriate error code number. The following table lists some of the error code values that are possible.

Error Code Name	**Number**
`e$illegal_buffer_size`	3575
`e$illegal_buffer_type`	3574
`e$invalid_io_operation`	1040

Return Value

The `setvbuf` function returns the value 0 if the operation succeeds.

If the operation fails, `setvbuf` returns the value -1.

Examples

The following program uses `setvbuf` to change a file's I/O buffer to a program-specified buffer, to change the mode of buffering to line buffered, and to change the buffer size to 300 bytes.

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

char buffer[300];

main()
{
   FILE *file_ptr;

   if ( (file_ptr = fopen("stream_file", "r")) == NULL )
      {
      puts("Error occurred opening stream_file.");
      exit(1);
      }

   setvbuf(file_ptr, buffer, _IOLBF, 300);

  fgetc(file_ptr);           /*  Perform some I/O to fill the buffer
*/

  printf("%s", buffer);       /*  Write out the contents of buffer
*/
                             /*  up to the first null character      */
}
```

If `stream_file` contains `abcdefghijklmnopqrstuvwxyz\n`, the output from the preceding program is as follows:

```
abcdefghijklmnopqrstuvwxyz
```

The newline character at the end of the data in the file is also written to `stdout` when `printf` is called.

Related Functions

`setbuf`

signal

Purpose

Determines how a specified signal will be handled.

Syntax

```
#include <signal.h>

void (*signal(int sig, void (*func) (int))) (int);
```

Explanation

The `signal` function determines how a specified signal `sig` will be handled. For example, if the calling program uses `signal` to set up a signal-handling function for a floating-point exception (`SIGFPE`), that signal-handling function is invoked when the exception condition occurs. If the calling program has not used `signal` to set up a signal-handling function for a floating-point exception, a default error handler is invoked when the exception condition occurs.

The `sig` argument is a signal in the range 1 through 8. The `signal.h` header file contains a defined constant (macro) for each signal. The following table lists the constant for each signal and the corresponding value and exception condition.

Value	Signal	Exception Condition
1	`SIGABRT`	Abnormal termination of the program, as is signaled by the `abort` function. Also, the `SIGABRT` signal is sent for many operating system errors.
2	`SIGFPE`	Floating-point exception (computational condition): the VOS zero divide, floating-point underflow, and floating-point overflow errors.
3	`SIGILL`	Illegal instruction.
4	`SIGINT`	Interrupt (break condition), which is signaled by pressing the terminal's CTRL and BREAK keys simultaneously.
5	`SIGSEGV`	Segmentation violation (illegal memory access).
6	`SIGTERM`	Program termination, which is signaled only by the program through the function invocation `kill(0, SIGTERM)`.
7	`SIGUSR1`	User-defined signal 1.
8	`SIGUSR2`	User-defined signal 2.

The `func` argument is a pointer to either the signal-handling function that will be called, or to one of two defined constants: `SIG_IGN` or `SIG_DFL`. The value of `func` determines how the signal will be handled.

- If `func` equals `SIG_IGN`, the specified signal will be ignored. You **cannot** specify `SIG_IGN` for the `SIGILL` or `SIGSEGV` signal.
- If `func` equals `SIG_DFL`, the specified signal will be handled in the default manner. See the "Default Signal Handlers" section earlier in this chapter for information on the default handling for each signal.
- If `func` equals a pointer to a signal-handling function defined within your program, the specified signal will be handled by that signal-handling function. A function name acts as a pointer to the function.

When the signal is raised and the signal-handling function is called, the handling function is passed the error code number (oncode) associated with the condition. If the signal is raised by a call to `kill(0, *sig*);`, the error code number is 0.

> **Note:** Passing the error code number to the signal-handling function is unique to VOS C and is not standard. If portability is important, do not use the passed error code number in the called signal-handling function.

If the `signal` function is not successful, it sets `errno` to the appropriate error code number.

See the "Signal Handling" section earlier in this chapter for more information on how to handle signals.

Return Value

If `signal` is successful, it returns a pointer to the previous signal-handling function for the specified signal. If `signal` is not successful, it returns `SIG_ERR`, as defined in the `signal.h` header file. For example, the `signal` function returns `SIG_ERR` when you attempt to specify `SIG_IGN` for either `SIGILL` or `SIGSEGV`.

Examples

The following program fragment uses `signal` to specify that `if_sigint`, a signal-handling function defined within the program, will be used to handle the `SIGINT` signal.

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

void if_sigint(int error_code);
```

(Continued on next page)

```
main()
{
   if ( signal(SIGINT, if_sigint) == SIG_ERR )
      {
      puts("signal returned SIG_ERR.");
      exit(1);
      }
      .
      .
      .
}
                              /*  Signal handler for the following:  */
void if_sigint(int error_code)    /*  user pressed CONTROL-BREAK
*/

   {
   puts("\nDo you really want to go to break level?");
   puts("To pause at break level, press CONTROL-BREAK again.");
   puts("To return to processing, do nothing.\n");
   }
```

Related Functions

`abort`, `kill`

sin

Purpose

Computes the sine of an angle, which is expressed in radians.

Syntax

```
#include <math.h>

double sin(double x);
```

Explanation

The `sin` function calculates the sine of `x`. The `x` argument is an angle expressed in radians.

If the absolute value of `x` exceeds 32,766, `sin` sets `errno` to the error code number returned by the operating system.

Return Value

The `sin` function returns a sine. When the absolute value of `x` exceeds 32,766, `sin` returns the value 0.

Examples

The following program uses `sin` to calculate the sine of an angle, expressed in radians, entered at the terminal's keyboard.

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

main()
{
   double x, sine;

   printf("Enter an angle (in radians) for which ");
   printf("you want to find the sine.\n");
   scanf("%le", &x);

   errno = 0;
   sine = sin(x);
```

(Continued on next page)

```
         if (errno != 0)
            printf("Error condition occurred:  errno = %d\n", errno);
         else
            printf("The sine of %lE radians = %lE.\n", x, sine);
      }
```

If `1.5707` were entered at the terminal's keyboard, the output would be as follows:

```
The sine of 1.570700E+00 radians = 1.000000E+00.
```

Related Functions

`asin`, `cos`, `tan`

sinh

Purpose

Computes the hyperbolic sine of a floating-point value.

Syntax

```
#include <math.h>

double sinh(double x);
```

Explanation

The `sinh` function calculates the hyperbolic sine of `x`. The `x` argument is a value of the type `double`.

If the result cannot be represented by a `double`, a range error occurs, and `sinh` sets `errno` to `ERANGE`.

Return Value

The `sinh` function returns a hyperbolic sine. When the result creates a range error, `sinh` returns positive or negative `HUGE_VAL` depending on the sign of `x`.

Examples

The following program uses `sinh` to calculate the hyperbolic sine of a number entered at the terminal's keyboard.

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

main()
{
   double x, h_sine;

   printf("Enter a number for which you want to find the hyperbolic
sine.\n");
   scanf("%le", &x);

   errno = 0;
   h_sine = sinh(x);
```

(Continued on next page)

```
      if (errno == ERANGE)
         puts("A range error occurred.");
      else
         printf("The hyperbolic sine of %lE = %lE.\n", x, h_sine);
   }
```

If `1.0` were entered at the terminal's keyboard, the output would be as follows:

```
The hyperbolic sine of 1.000000E+00 = 1.175201E+00.
```

Related Functions

`cosh`, `tanh`

sleep

Purpose

Suspends the calling process for a specified number of seconds.

Syntax

```
#include <c_utilities.h>

int sleep(unsigned int seconds);
```

Explanation

The `sleep` function causes the calling process to sleep for the number of seconds specified in `seconds`. Until the specified number of seconds expires, the only way to awaken the process is by pressing the terminal's CTRL and BREAK keys simultaneously.

In VOS C, `sleep` will not suspend the calling process for more than 2,097,152 seconds. If the value of `seconds` exceeds this limit, `sleep` suspends the process for 2,097,152 seconds.

Return Value

If the sleep operation is successful, the `sleep` function returns the value 0.

If the sleep operation is not successful, `sleep` returns the value of `seconds`.

Examples

The following program uses `sleep` to suspend the program's process for the number of seconds entered at the terminal's keyboard.

```
#include <c_utilities.h>

unsigned int seconds;

main()
{
   puts("How many seconds should the program's process be
suspended?");
   scanf("%d", &seconds);

   sleep(seconds);
}
```

Related Functions

`system`

sprintf

Purpose

Writes a formatted string of characters to an array.

Syntax

```
#include <stdio.h>

int sprintf(char *s, const char *format, ...);
```

Explanation

The `sprintf` function writes output to the array of `char` whose address is specified by `s`. After the characters are written, `sprintf` appends a null character so that the specified array is a C string. The output written is under the control of the string pointed to by `format`.

One possible use of `sprintf` is to convert a numeric value stored in a variable into an ASCII character representation of the value. For example, the value 123 stored in an `int` variable can be written to an array as `"123"`, three characters plus the appended null character.

The string pointed to by `format` can consist of zero or more directives. The `sprintf` function executes each directive in turn. It returns when it encounters the end of the format string. The directives consist of the following:

- characters, including escape sequences
- conversion specifications.

The `sprintf` function writes ordinary characters (not `%`) unchanged to the array. Each *conversion specification* begins with a `%` and indicates how the value of a subsequent argument will be converted and written. The number and types of the arguments in a `sprintf` call depend on the conversion specifications given in the format string. Each conversion specification results in fetching zero or more arguments. In a `sprintf` call, the syntax for the argument list is as follows:

```
sprintf(s, format, [argument]...)
```

Each *`argument`* must be an expression yielding an object type appropriate for the conversion specification. If there are insufficient arguments for the format, the conversions have unpredictable results. If the format is exhausted while arguments remain, the extra arguments are evaluated but otherwise ignored.

Conversion Specifications. The format for a conversion specification is as follows:

```
%[flags][width] .[precision][modifier] conversion_specifier
```

For information on each element of a conversion specification, see the `printf` function.

The following table provides a summary of the formatted output conversion specifiers. The table shows some of the argument types typically used with each specifier, and a description and example of the output generated by each specifier. The `s` argument locates an array of `char` into which output will be written. In each example, after the characters are written, `sprintf` appends a null character so that the specified array contains a C string.

(Page 1 of 2)

Conversion Specifier	Argument Type	Output
`c`	`int` `char`	Writes a single character. For example, `sprintf(s, "%c", 'A')` writes `A` to the array.
`d, i`	`int`	Writes a signed decimal. For example, `sprintf(s, "%d", -123)` writes `-123` to the array.
`e, E`	`double` `float`	Writes a floating-point value using exponential notation. For example, `sprintf(s, "%e", -987.654)` writes `-9.876540e+002` to the array. The `E` conversion specifier writes a number with an `E` instead of an `e` introducing the exponent.
`f`	`double` `float`	Writes a floating-point value as a decimal fraction. For example, `sprintf(s, "%f", -987.345)` writes `-987.345000` to the array.
`g, G`	`double` `float`	Writes a floating-point value using `e` or `f` format, whichever is more compact. For example, `sprintf(s, "%g", -987.3456)` writes `-987.346` to the array. The `G` conversion specifier writes the number with an `E` instead of an `e` introducing the exponent.
`n`	`int *`	Writes, into the pointed-to `int` variable, the number of characters output so far by this `sprintf` call.
`o`	`unsigned int`	Writes an unsigned integer in octal format. For example, `sprintf(s, "%o", 0177)` writes `177` to the array.
`p`	`void *`	Writes a pointer as an absolute address in hexadecimal format. For example, `sprintf(s, "%p", NULL)` writes `00000000` to the array.
`s`	`char *`	Writes a C string up to (but not including) the null character. For example, `sprintf(s, "%s", "abcd")` writes `abcd` to the array.
`v`	`char_varying(`*n*`) *`	Writes a `char_varying` string up to the current length. For example, if `cv` is a `char_varying` object storing `"efgh"`, the call `sprintf(s, "%v", &cv)` writes `efgh` to the array.
`u`	`unsigned int`	Writes an unsigned integer in decimal format. For example, `sprintf(s, "%u", 1234)` writes `1234` to the array.

(Page 2 of 2)

Conversio n Specifier	Argument Type	Output
`x, X`	`unsigned int`	Writes an unsigned integer in hexadecimal format. For example, `sprintf(s, "%x", 0x7F)` writes `7f` to the array. The `X` conversion specifier writes the number with the uppercase letters `ABCDEF` instead of the lowercase letters `abcdef`.
`%%`	none	Writes a percent sign to the array.

Return Value

If the formatted output specified is successfully written, the `sprintf` function returns the number of characters, except for the terminating null character, written to the array.

Examples

The following program uses `sprintf` to convert an `int` and a `double` value into ASCII character representations of the values and to write the string to a specified array.

```
#include <stdio.h>

char string[50];

int i_num = 123;
double d_num = 987.654;

main()
{
   sprintf(string, "field 1: |%6d|\nfield 2: |%6.2f|\n", i_num, d_num);

   printf("%s", string);
}
```

The output written to the array `string` is as follows:

```
field 1: |   123|\nfield 2: |987.65|
```

Related Functions

`fprintf`, `printf`, `vfprintf`, `vprintf`, `vsprintf`

sqrt

Purpose

Computes the square root of a non-negative, floating-point value.

Syntax

```
#include <math.h>

double sqrt(double x);
```

Explanation

The `sqrt` function calculates the square root of `x`. The `x` argument is a non-negative value of the type `double`.

If `x` is negative, a domain error occurs, and `sqrt` sets `errno` to `EDOM`.

Return Value

The `sqrt` function returns the square root of the value `x`. If `x` is negative, `sqrt` returns the value 0.

Examples

The following program uses `sqrt` to calculate the square root of a floating-point number entered at the terminal's keyboard.

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

main()
{
   double x, sq_root;

   printf("Enter the number for which you want to find the square
root.\n");
   scanf("%lf", &x);

   errno = 0;
   sq_root = sqrt(x);
```

(Continued on next page)

```
   if (errno == EDOM)
      puts("A domain error occurred.");
   else
      printf("The square root of %.3lf = %.3lf\n", x, sq_root);
}
```

If `144.0` were entered at the terminal's keyboard, the output would be as follows:

```
The square root of 144.000 = 12.000
```

Related Functions

`pow`

srand

Purpose

Initializes the sequence of pseudo-random numbers that are returned by the `rand` function.

Syntax

```
#include <stdlib.h>

void srand(unsigned int seed);
```

Explanation

The `srand` function uses `seed` to initialize the sequence of pseudo-random numbers that are returned by subsequent calls to `rand`.

Different values for `seed` cause the `rand` function to generate different, though reproducible, sequences of random numbers. For example, if you call `srand` and pass the value 1 as the `seed` argument, a sequence of calls to `rand` generates one reproducible set of pseudo-random numbers. Another sequence of calls to `rand` generates the same series of pseudo-random numbers. However, if you call `srand` and pass the value 99 as the `seed` argument, another sequence of calls to `rand` generates a different series of pseudo-random numbers.

If `rand` is invoked without first calling `srand`, the same sequence of numbers is generated as when `srand` is first called with a seed value of 1.

Note: The sequence of numbers returned by `rand` may change in future releases of the library functions.

Return Value

The `srand` function returns no value.

Examples

The following program contains two examples of how `srand` is used to initialize a sequence of pseudo-random numbers that are returned by subsequent calls to `rand`. The numbers returned by `rand` are displayed on the terminal's screen.

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

(Continued on next page)

```
main()
{
   unsigned int seed;
   int i, random_number;

   seed = 1;             /*  Use the value 1 as a seed     */
   srand(seed);
   for (i = 0; i < 3; i++)
      {
      random_number = rand();
      printf("Random number #%d = %d\n", (i + 1), random_number);
      }

   puts("\nGenerate a different sequence of pseudo-random
numbers.\n");

   seed = 99;            /*  Use a new seed, the value 99  */
   srand(seed);
   for (i = 0; i < 3; i++)
      {
      random_number = rand();
      printf("Random number #%d = %d\n", (i + 1), random_number);
      }
}
```

The output from the preceding program is as follows:

```
Random number #1 = 10523
Random number #2 = 11866
Random number #3 = 690

Generate a different sequence of pseudo-random numbers.

Random number #1 = 25957
Random number #2 = 32292
Random number #3 = 8532
```

Related Functions

`rand`

sscanf

Purpose

Using a specified format, reads a sequence of characters from a given string.

Syntax

```
#include <stdio.h>

int sscanf(const char *s, const char *format, ...);
```

Explanation

The `sscanf` function reads input from the array whose address is specified by `s` and converts the input sequence of characters using the directives specified in the string pointed to by `format`. The `sscanf` function assigns each converted result to an object pointed to by the corresponding argument in the argument list.

The `s` argument locates an array of `char` containing a null-terminated string from which the input is to be obtained. Reaching the end of the string is equivalent to encountering end-of-file with the `fscanf` function. If the copying takes place between objects that overlap, `sscanf` produces unpredictable results.

The string pointed to by `format` consists of zero or more directives. The directives consist of the following:

- One or more white-space characters cause `sscanf` to read up to the next non-white-space character, which remains unread, or until it can read no more characters. (A newline character is a white-space character.)
- Ordinary non-white-space characters (not `%`) cause `sscanf` to read the next characters from the array. When `sscanf` encounters a nonmatching character, the directive fails, and the nonmatching and subsequent characters remain unread.
- A *conversion specification* defines a set of input sequences that `sscanf` attempts to match. The conversion specifications are described later in the Explanation.

The `sscanf` function executes each directive in turn. If a directive fails, the `sscanf` function returns. A directive can fail, for example, if no input characters are available or if the input characters do not match those specified by the directive.

Typically, each conversion specification causes `sscanf` to read an input item from the array, to convert the input item to the type specified by the conversion specifier, and to assign the converted result to the object pointed to by the corresponding argument. The following example shows an `sscanf` call consisting of an `s` argument, a `%d` conversion specification, and a corresponding argument having the type pointer to `int`.

```
int i_num;

sscanf(s, "%d", &i_num);
```

In the preceding `sscanf` call, `sscanf` executes the `%d` directive by reading characters from the array specified by `s`, converting the characters that match the `%d` format to an `int` value, and assigning the converted result to the object located by the expression `&i_num`. If the input sequence of characters were `99abcde`, `sscanf` would assign the value 99 to `i_num`.

In an `sscanf` call, the syntax for the argument list is as follows:

```
sscanf(s, format,[argument]...)
```

Each *`argument`* must be an expression yielding a pointer to an object type appropriate for the conversion specification. The number and types of the arguments in an `sscanf` call depend on the conversion specifications given in the format string.

- If the pointed-to object does not have the appropriate type, or if the result of the conversion cannot be represented in the pointed-to object, `sscanf` yields unpredictable results.
- If there are not enough arguments for the format, the behavior is unpredictable.
- If the format is exhausted while arguments remain, the excess arguments are evaluated, as are all function arguments, but are otherwise ignored.

Conversion Specifications. The format string typically includes one or more conversion specifications. Each conversion specification describes how `sscanf` is to interpret a part of the input sequence of characters. The format for a conversion specification is as follows:

```
%[*][width][size] conversion_specifier
```

For information on each element of a conversion specification, see the `scanf` function.

The following table provides a summary of the formatted input conversion specifiers. For each specifier, the table shows the corresponding argument's type and a description of the expected input item.

(Page 1 of 2)

Conversion Specifier	Argument Type	Expected Input Item
`c`	`char *`	A single character or the number of characters specified in *`width`*. White-space characters are not skipped.
`d`	`int *`	An optionally signed decimal integer.
`e, E, f, g, G`	`float *`	An optionally signed floating-point number.

(Page 2 of 2)

Conversion Specifier	Argument Type	Expected Input Item
`i`	`int *`	An optionally signed integer in decimal, hexadecimal, or octal format.
`n`	`int *`	The number of characters read so far by this `sscanf` call is written into the pointed-to `int` variable.
`o`	`unsigned int *`	An optionally signed octal integer.
`p`	`void **`	An eight-digit absolute address in hexadecimal format, such as `0000000a`.
`s`	`char *`	A sequence of non-white-space characters delimited by the first white-space character. A terminating null character is appended to the sequence of characters.
`u`	`unsigned int *`	An optionally signed decimal integer.
`v`	`char_varying(`*`n`*`) *`	A varying-length character string.
`x, X`	`unsigned int *`	An optionally signed hexadecimal integer.
`%%`	none	Matches a percent sign.
`[`*`scanlist`*`]`	`char *`	A sequence of characters delimited by the first character not in the *`scanlist`*. A terminating null character is appended to the sequence of characters.

The following table provides a sample invocation of `sscanf` with each conversion specifier. The `s` argument locates a string from which the input will be read. For each specifier, the table shows an input sequence and describes how `sscanf` stores the input item.

(Page 1 of 2)

Function Invocation	Argument Declaration	Input Sequence	Effect
`sscanf(s, "%c", &ch)`	`char ch;`	`128e2`	Stores the character `'1'` in `ch`.
`sscanf(s, "%d", &i)`	`int i;`	`128e2`	Stores the value 128 in `i`.
`sscanf(s, "%e", &f)`	`float f;`	`128e2`	Stores the value 12800.0 in `f`.
`sscanf(s, "%i", &i)`	`int i;`	`128e2`	Stores the value 128 in `i`.
`sscanf(s, "%o", &u)`	`unsigned int u;`	`128e2`	Stores the value 12 octal in `u`.
`sscanf(s, "%p", &p)`	`void *p;`	`0000000a`	Stores the value 0000000a hex. in `p`.
`sscanf(s, "%s", a)`	`char a[10];`	`abcd efg`	Stores the string `"abcd"` in `a`.

(Page 2 of 2)

Function Invocation	Argument Declaration	Input Sequence	Effect
`sscanf(s, "%u", &u)`	`unsigned int u;`	`128e2`	Stores the value 128 in `u`.
`sscanf(s, "%v", &cv)`	`char_varying(5) cv;`	`abcd efg`	Stores the characters `abcd` in `cv`.
`sscanf(s, "%X", &u)`	`unsigned int u;`	`00ff`	Stores the value ff hex. in `u`.
`sscanf(s, "%%")`	none	`%`	Matches a `%` character.
`sscanf(s, "%[12345]", &a)`	`char a[10];`	`128e2`	Stores the string `"12"` in `a`.

Return Value

The `sscanf` function returns the number of input items assigned to arguments. The number of input items assigned can be fewer than expected, or zero, depending on the number of input characters that match those specified by the format string's directives.

If end of input occurs before any successful conversion, including those with suppressed assignments, `sscanf` returns `EOF`.

Examples

The following program uses `sscanf` to read input from two arrays, `string1` and `string2`. The `sscanf` function converts each input item and assigns the result to the corresponding argument. #include <stdio.h>

```
#include <stdlib.h>

int items;
double d_num;
short s_num;

char string1[] = "987.654";
char string2[] = "abcdefghijkl";
char array[25];

main()
{
  items = sscanf(string1, "%lf", &d_num);          /*  Example 1  */
   printf("Example 1:  items = %d\n", items);
   printf("d_num = %lf\n", d_num);

  items = sscanf(string2, "%[^g]", array);         /*  Example 2  */
   printf("\nExample 2:  items = %d\n", items);
   printf("array = %s\n", array);
}
```

The output from the preceding program is as follows:

```
Example 1:  items = 1
d_num = 987.654000

Example 2:  items = 1
array = abcdef
```

Related Functions

`fscanf`, `scanf`

strcat

Purpose

Appends one string to another string.

Syntax

```
#include <string.h>

char         *strcat(...);    /*  Types of the arguments can vary  */

char          *strcat_nstr_vstr(char *s1, const char_varying *v2);
char_varying *strcat_vstr_nstr(char_varying *v1, const char *s2);
char_varying *strcat_vstr_vstr(char_varying *v1,const char_varying
*v2);
```

Explanation

A sample invocation of the `strcat` function is as follows:

```
return_value = strcat(string1, string2);
```

In VOS C, the `strcat` function is a generic, string-manipulation function that appends the string pointed to by `string2` to the end of the string pointed to by `string1`. The `string1` and `string2` arguments can be either pointers to `char` or pointers to `char_varying` strings, or a combination of the two pointer types. The `return_value` is a pointer to `string1`.

With the generic, string-manipulation functions, the VOS C compiler takes a special action if either `string1` or `string2` contains the address of a `char_varying` string. Based on the data types of the function's arguments, the compiler translates the name of the function (`strcat`) into the name of the actual function that is invoked. The following table lists the actual function names that the compiler uses and the data types of each function's `string1` argument, `string2` argument, and return value.

Actual Function Name	**string1**	**string2**	**Return Value**
`strcat`	`char *`	`char *`	`char *`
`strcat_nstr_vstr`	`char *`	`char_varying(`*`l`*`) *`	`char *`
`strcat_vstr_nstr`	`char_varying(`*`l`*`) *`	`char *`	`char_varying *`
`strcat_vstr_vstr`	`char_varying(`*`l`*`) *`	`char_varying(`*`m`*`) *`	`char_varying *`

When you invoke the function, you can specify `strcat` or the actual function name in your program. For example, if `string1` is a pointer to a `char_varying` string and `string2` is a pointer to `char`, you can invoke the function using `strcat`, or the actual function name, `strcat_vstr_nstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype.

The length of the data appended, `string2`, and the length of the resulting string, `string1`, are determined as follows:

- If `string2` is a pointer to `char`, the length of the string that is appended equals the number of characters preceding the first null character in `string2`.
- If `string2` is a pointer to a `char_varying` string, the length of the string that is appended equals the string's current length.
- If `string1` is a pointer to `char`, its terminating null character is overwritten by the initial character of `string2`, and a null character is appended to the resulting string.
- If `string1` is a pointer to a `char_varying` string, no null character is appended to the resulting string.

Be sure that `string1` has enough space to hold `string2`, including the null character if one is appended. **If** `string1` does not have enough space to hold `string2`, other data in your program can be corrupted. Also, if the copying takes place between two objects that overlap, the `strcat` function can have unpredictable results.

Return Value

The `strcat` function returns a pointer to `string1`. The return value has the type pointer to `char` if `string1` is a pointer to `char`, or has the type pointer to `char_varying` if `string1` is a pointer to a `char_varying` string.

Examples

The following program contains two examples of the `strcat` function. In each example, one string is appended to another string. In the first example, each argument to `strcat` is a pointer to `char`. In the second example, both arguments are pointers to `char_varying` strings.

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

char nstr_1[30] = "Congratulations and ";
char nstr_2[30] = "good luck.";

char_varying(30) vstr_1 = "Good luck and ";
char_varying(30) vstr_2 = "good riddance.";
char_varying *vstr_1_ptr;
```

(Continued on next page)

```
main()
{
   strcat(nstr_1, nstr_2);                            /*  Example 1  */

   printf("%s\n", nstr_1);

   vstr_1_ptr = strcat_vstr_vstr(&vstr_1, &vstr_2);    /*  Example 2
*/

   printf("%v\n", vstr_1_ptr);
}
```

The output from the preceding program is as follows:

```
Congratulations and good luck.
Good luck and good riddance.
```

Related Functions

`strcpy`, `strncat`, `strncpy`

strchr

Purpose

Searches for the first occurrence of a specified character within a given string.

Syntax

```
#include <string.h>

char *strchr(...);        /*  Types of the arguments can vary  */

char *strchr_vstr(const char_varying *v, int c);
```

Explanation

A sample invocation of the `strchr` function is as follows:

```
return_value = strchr(string, c);
```

In VOS C, the `strchr` function is a generic, string-manipulation function that searches for the first occurrence of the character `c` (converted to an `unsigned char`) within the string pointed to by `string`. The `string` argument can be either a pointer to `char` or a pointer to a `char_varying` string. If `string` is a pointer to `char`, the string's null character is part of the sequence of characters searched.

With the generic, string-manipulation functions, the VOS C compiler takes a special action if `string` contains the address of a `char_varying` string. In this case, the compiler translates the name of the function (`strchr`) into the name of the actual function that is invoked. The following table lists the actual function names that the compiler uses and the data types of each function's `string` argument, `c` argument, and return value.

Actual Function Name	string	c	Return Value
`strchr`	`char *`	`int`	`char *`
`strchr_vstr`	`char_varying(`*`l`*`) *`	`int`	`char *`

When you invoke the function, you can specify `strchr` or the actual function name in your program. For example, if `string` is a pointer to a `char_varying` string, you can invoke the function using `strchr`, or the actual function name, `strchr_vstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype.

The length of the string that `strchr` searches is determined as follows:

- If `string` is a pointer to `char`, the length of the string equals the number of characters up to and including the first null character.
- If `string` is a pointer to a `char_varying` string, the length of the string equals its current length.

Return Value

If `strchr` finds the specified character, the function returns a pointer to the located character within `string`. The return value has the type pointer to `char`.

If `strchr` does not find the specified character, it returns the `NULL` pointer constant.

Examples

The following program contains two examples of the `strchr` function. In each example, the function searches for the first occurrence of an `X` within a specified string. In the first example, the `string` argument is a pointer to `char`. In the second example, the `string` argument is a pointer to a `char_varying` string.

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

char nstr[30] = "abcdefghXijklXmn";
char_varying(30) vstr = "abcdXefghijXklmn";
char_varying(30) substring;

int c = 'X';
unsigned int offset;

main()
{
   char *char_ptr;

   char_ptr = strchr(nstr, c);                    /*  Example 1  */
   if (char_ptr != NULL)
      printf("Example 1:  %s\n", char_ptr);

   char_ptr = strchr_vstr(&vstr, c);              /*  Example 2  */
   if (char_ptr != NULL)
      {
      offset = (char_ptr - &vstr) - 1;
      substring = $substr(vstr, offset);
      printf("Example 2:  %v\n", &substring);
      }
}
```

The output from the preceding program is as follows:

```
    Example 1:   XijklXmn
    Example 2:   XefghijXklmn
```

In example 2 of the preceding program, the `$substr` built-in function is used to extract the `char_varying` substring whose initial character is pointed to by the return value of `strchr`. The following expression yields the offset of the first `X` character within the character array part of `vstr`.

```
    (char_ptr - &vstr) - 1;
```

This offset is used in the `$substr` function call. The `$substr` function uses the value 1 (not 0) as the index of the initial character within the character array part of a `char_varying` string. Thus, subtracting 1 from the result of `char_ptr - &vstr` takes into account the 2-byte, current-length field in `vstr` and yields the desired offset for the `$substr` call. As shown in Figure 11-2, `char_ptr - &vstr` yields the value 6. Subtracting 1 from this result yields 5, the offset of the first `X` character within the character-array part of `vstr`.

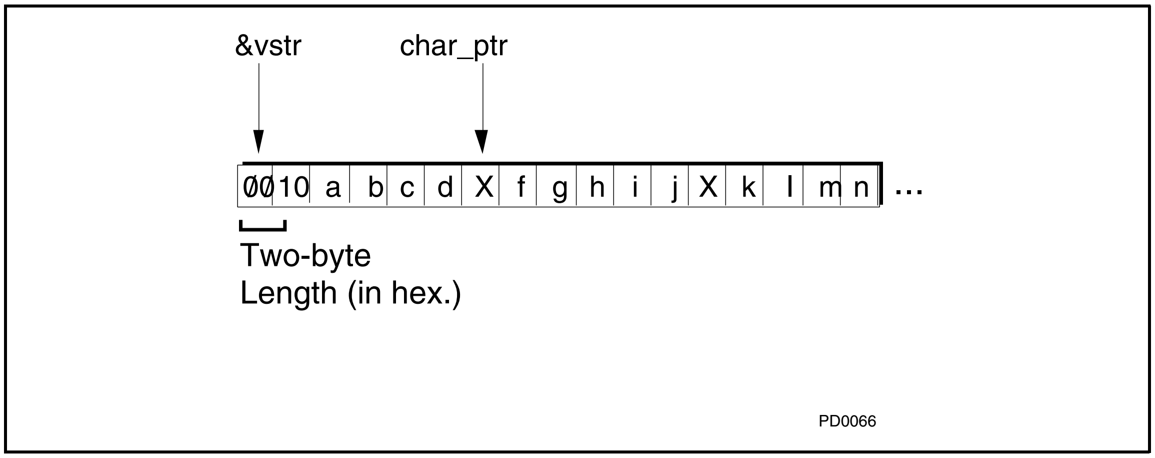


Figure 11-2. Calculating an Offset within `char_varying` Data

See the “Varying-Length Character String Type” section in Chapter 4 for information on that data type.

See the “`$substr`” section in Chapter 12 for information on that built-in function.

Related Functions

`memchr`, `strcspn`, `strpbrk`, `strrchr`, `strspn`, `strstr`, `strtok`

strcmp

Purpose

Compares one string with another string.

Syntax

```
#include <string.h>

int strcmp(...);          /*  Types of the arguments can vary  */

int strcmp_nstr_vstr(const char *s1, const char_varying *v2);
int strcmp_vstr_nstr(const char_varying *v1, const char *s2);
int strcmp_vstr_vstr(const char_varying *v1, const char_varying *v2);
```

Explanation

A sample invocation of the `strcmp` function is as follows:

```
return_value = strcmp(string1, string2);
```

In VOS C, the `strcmp` function is a generic, string-manipulation function that compares the string pointed to by `string1` with the string pointed to by `string2`. The `string1` and `string2` arguments can be either pointers to `char` or pointers to `char_varying` strings, or a combination of the two pointer types.

With the generic, string-manipulation functions, the VOS C compiler takes a special action if either `string1` or `string2` contains the address of a `char_varying` string. Based on the data types of the function's arguments, the compiler translates the name of the function (`strcmp`) into the name of the actual function that is invoked. The following table lists the actual function names that the compiler uses and the data types of each function's `string1` argument, `string2` argument, and return value.

Actual Function Name	**string1**	**string2**	**Return Value**
`strcmp`	`char *`	`char *`	`int`
`strcmp_nstr_vstr`	`char *`	`char_varying(`*`l`*`) *`	`int`
`strcmp_vstr_nstr`	`char_varying(`*`l`*`) *`	`char *`	`int`
`strcmp_vstr_vstr`	`char_varying(`*`l`*`) *`	`char_varying(`*`m`*`) *`	`int`

When you invoke the function, you can specify `strcmp` or the actual function name in your program. For example, if `string1` is a pointer to a `char_varying` string and `string2` is a pointer to `char`, you can invoke the function using `strcmp`, or the actual function name, `strcmp_vstr_nstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype.

The lengths of the strings compared, `string1` and `string2`, are determined as follows:

- If the string is a pointer to `char`, the length of the string equals the number of characters preceding the first null character.
- If the string is a pointer to a `char_varying` string, the length of the string equals its current length.

The `strcmp` function uses the following procedure to compare `string1` and `string2`. The strings are compared, character by character, until two characters are found that are different. The `strcmp` function then compares the two characters and returns a value based on which of the two characters' ASCII codes has the greater numeric value. If the two strings compare equal up to the length of the shorter string, the longer string always compares greater.

Return Value

The `strcmp` function returns a value based on the comparison of the strings pointed to by `string1` and `string2`. Based on the results of the comparison, `strcmp` returns an `int` value that is greater than 0, less than 0, or equal to 0. See the following table.

Return Value	**Result of the Comparison**
Greater than 0	When two characters are found that are different, the character in the string pointed to by `string1` has a greater ASCII code than the character in the string pointed to by `string2`.
Less than 0	When two characters are found that are different, the character in the string pointed to by `string1` has a smaller ASCII code than the character in the string pointed to by `string2`.
Equal to 0	Every character in the string pointed to by `string1` is identical to the corresponding character in the string pointed to by `string2`.

Examples

The following program contains two examples of the `strcmp` function. In each example, one string is compared with another string. In the first example, each argument to `strcmp` is a pointer to `char`. In the second example, one argument is a pointer to `char`, and the other argument is a pointer to a `char_varying` string.

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

char nstr_1[10] = "ABCDEFGHI";
char nstr_2[10] = "ABCDEFGH";

char_varying(10) vstr_2 = "abcdefghi";
```

(Continued on next page)

```
int return_value;

main()
{
  return_value = strcmp(nstr_1, nstr_2);            /*  Example 1  */

   if (return_value > 0)
      printf("nstr_1 is greater than nstr_2.\n");

   if (return_value < 0)
      printf("nstr_2 is greater than nstr_1.\n");

   if (return_value == 0)
      printf("nstr_1 and nstr_2 are equal.\n");

   return_value = strcmp_nstr_vstr(nstr_1, &vstr_2);    /*  Example
2  */

   if (return_value > 0)
      printf("nstr_1 is greater than vstr_2.\n");

   if (return_value < 0)
      printf("vstr_2 is greater than nstr_1.\n");

   if (return_value == 0)
      printf("nstr_1 and vstr_2 are equal.\n");
}
```

The output from the preceding program is as follows:

```
nstr_1 is greater than nstr_2.
vstr_2 is greater than nstr_1.
```

Related Functions

`memcmp`, `strncmp`

strcpy

Purpose

Copies one string into another string.

Syntax

```
#include <string.h>

char         *strcpy(...);    /*  Types of the arguments can vary  */

char          *strcpy_nstr_vstr(char *s1, const char_varying *v2);
char_varying *strcpy_vstr_nstr(char_varying *v1, const char *s2);
char_varying *strcpy_vstr_vstr(char_varying *v1, const char_varying
*v2);
```

Explanation

A sample invocation of the `strcpy` function is as follows:

```
return_value = strcpy(string1, string2);
```

In VOS C, the `strcpy` function is a generic, string-manipulation function that copies the string pointed to by `string2` into the string pointed to by `string1`. The `string1` and `string2` arguments can be either pointers to `char` or pointers to `char_varying` strings, or a combination of the two pointer types. The `return_value` is a pointer to `string1`.

With the generic, string-manipulation functions, the VOS C compiler takes a special action if either `string1` or `string2` contains the address of a `char_varying` string. Based on the data types of the function's arguments, the compiler translates the name of the function (`strcpy`) into the name of the actual function that is invoked. The following table lists the actual function names that the compiler uses and the data types of each function's `string1` argument, `string2` argument, and return value.

Actual Function Name	**string1**	**string2**	**Return Value**
`strcpy`	`char *`	`char *`	`char *`
`strcpy_nstr_vstr`	`char *`	`char_varying(`*`l`*`) *`	`char *`
`strcpy_vstr_nstr`	`char_varying(`*`l`*`) *`	`char *`	`char_varying *`
`strcpy_vstr_vstr`	`char_varying(`*`l`*`) *`	`char_varying(`*`m`*`) *`	`char_varying *`

When you invoke the function, you can specify `strcpy` or the actual function name in your program. For example, if `string1` is a pointer to a `char_varying` string and `string2` is a pointer to `char`, you can invoke the function using `strcpy`, or the actual function name, `strcpy_vstr_nstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype.

The length of the data copied, `string2`, and the length of the resulting string, `string1`, are determined as follows:

- If `string2` is a pointer to `char`, the length of the string that is copied equals the number of characters preceding the first null character in `string2`.
- If `string2` is a pointer to a `char_varying` string, the length of the string that is copied equals the current length of `string2`.
- If `string1` is a pointer to `char`, a terminating null character is appended to the resulting string.
- If `string1` is a pointer to a `char_varying` string, no null character is appended to the resulting string.

Be sure that `string1` has enough space to hold `string2`, including the null character if one is appended. **If** `string1` does not have enough space to hold `string2`, other data in your program can be corrupted. Also, if the copying takes place between two objects that overlap, the `strcpy` function can have unpredictable results.

In VOS C, you can directly assign the value of one `char_varying` string to another `char_varying` string. For example:

```
char_varying(20) vstr_1, vstr_2 = "John Brown";

vstr_1 = vstr_2;
```

The assignment `vstr_1 = vstr_2` can replace a call to the `strcpy_vstr_vstr` function.

Return Value

The `strcpy` function returns a pointer to `string1`. The return value has the type pointer to `char` if `string1` is a pointer to `char`, or has the type pointer to `char_varying` if `string1` is a pointer to a `char_varying` string.

Examples

The following program contains two examples of the `strcpy` function. In each example, one string is copied into another string. In the first example, each argument to `strcpy` is a pointer to `char`. In the second example, one argument is a pointer to a `char_varying` string, and the other argument is a pointer to `char`.

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

char nstr_1[30], nstr_2[30] = "Welcome home.";

char_varying(30) vstr_1;
char_varying *vstr_1_ptr;

main()
{
   strcpy(nstr_1, nstr_2);                            /*  Example 1  */

   printf("%s\n", nstr_1);

  vstr_1_ptr = strcpy_vstr_nstr(&vstr_1, nstr_2);    /*  Example 2  */

   printf("%v\n", vstr_1_ptr);
}
```

The output from the preceding program is as follows:

```
Welcome home.
Welcome home.
```

Related Functions

`memcpy`, `memset`, `strcat`, `strncat`, `strncpy`

strcspn

Purpose

Computes the length of the initial segment of a specified string that consists entirely of characters **not** from another string.

Syntax

```
#include <string.h>

size_t strcspn(...);     /*  Types of the arguments can vary  */

size_t strcspn_nstr_vstr(const char *s1, const char_varying *v2);
size_t strcspn_vstr_nstr(const char_varying *v1, const char *s2);
size_t strcspn_vstr_vstr(const char_varying *v1, const char_varying
*v2);
```

Explanation

A sample invocation of the `strcspn` function is as follows:

```
return_value = strcspn(string1, string2);
```

In VOS C, the `strcspn` function is a generic, string-manipulation function that calculates the length of the initial segment of the string pointed to by `string1` that consists entirely of characters **not** from the string pointed to by `string2`. The `string1` and `string2` arguments can be either pointers to `char` or pointers to `char_varying` strings, or a combination of the two pointer types.

With the generic, string-manipulation functions, the VOS C compiler takes a special action if either `string1` or `string2` contains the address of a `char_varying` string. Based on the data types of the function's arguments, the compiler translates the name of the function (`strcspn`) into the name of the actual function that is invoked. The following table lists the actual function names that the compiler uses and the data types of each function's `string1` argument, `string2` argument, and return value.

Actual Function Name	**string1**	**string2**	**Return Value**
`strcspn`	`char *`	`char *`	`size_t`
`strcspn_nstr_vstr`	`char *`	`char_varying(`*`l`*`) *`	`size_t`
`strcspn_vstr_nstr`	`char_varying(`*`l`*`) *`	`char *`	`size_t`
`strcspn_vstr_vstr`	`char_varying(`*`l`*`) *`	`char_varying(`*`m`*`) *`	`size_t`

When you invoke the function, you can specify `strcspn` or the actual function name in your program. For example, if `string1` is a pointer to a `char_varying` string and `string2` is a pointer to `char`, you can invoke the function using `strcspn`, or the actual function name, `strcspn_vstr_nstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype.

The lengths of the strings, `string1` and `string2`, are determined as follows:

- If the string is a pointer to `char`, the length of the string equals the number of characters preceding the first null character.
- If the string is a pointer to a `char_varying` string, the length of the string equals its current length.

Return Value

The `strcspn` function returns the length of the initial segment of `string1` that consists entirely of characters not from `string2`. The return value has the type `size_t`, which is a type definition declared in the `stddef.h` header file.

Examples

The following program contains two examples of the `strcspn` function. In each example, `strcspn` calculates the length of the initial segment of one string consisting entirely of characters not from another string. In the first example, both arguments to `strcspn` are pointers to `char`. In the second example, one argument is a pointer to a `char_varying` string, and the other argument is a pointer to `char`.

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

char nstr_1[30] = "there's magic in the web of it";
char *nstr_2;

char_varying(30) vstr_1 = "to be or not to be";
char_varying(30) vstr_2;

size_t segment_length;

main()
{
    nstr_2 = "\t \n";
   segment_length = strcspn(nstr_1, nstr_2);              /*  Example
1  */
    printf("Example 1:  segment_length = %u\n", segment_length);

    strcpy_vstr_nstr(&vstr_2, "lmn");
    segment_length = strcspn_vstr_vstr(&vstr_1, &vstr_2);     /*
Example 2  */
    printf("Example 2:  segment_length = %u\n", segment_length);
}
```

The output from the preceding program is as follows:

```
Example 1:  segment_length = 7
Example 2:  segment_length = 9
```

Related Functions

`memchr`, `strchr`, `strpbrk`, `strrchr`, `strspn`, `strstr`, `strtok`

strlen

Purpose

Computes the length of a specified string.

Syntax

```
#include <string.h>

size_t strlen(...);      /*  Types of the arguments can vary  */

size_t strlen_vstr(const char_varying *v);
```

Explanation

A sample invocation of the `strlen` function is as follows:

```
return_value = strlen(string);
```

In VOS C, the `strlen` function is a generic, string-manipulation function that calculates the length of the string pointed to by `string`. The `string` argument can be either a pointer to `char` or a pointer to a `char_varying` string.

With the generic, string-manipulation functions, the VOS C compiler takes a special action if `string` contains the address of a `char_varying` string. In this case, the compiler translates the name of the function (`strlen`) into the name of the actual function that is invoked. The following table lists the actual function names that the compiler uses and the data types of each function's `string` argument and return value.

Actual Function Name	string	Return Value
`strlen`	`char *`	`size_t`
`strlen_vstr`	`char_varying(`*`l`*`) *`	`size_t`

When you invoke the function, you can specify `strlen` or the actual function name in your program. For example, if `string` is a pointer to a `char_varying` string, you can invoke the function using `strlen`, or the actual function name, `strlen_vstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype.

The length of the specified string is determined as follows:

- If the string is a pointer to `char`, the length of the string equals the number of characters preceding the first null character.
- If the string is a pointer to a `char_varying` string, the length of the string equals its current length.

Though it is not shown in the Syntax section, `strlen` is also implemented to generate inline code that returns the current length of a `char_varying` string if the argument is a `char_varying` string (as opposed to the address of a `char_varying` string). In this case only, the return value of `strlen` has the type `short`.

Return Value

The `strlen` function returns the length of the string pointed to by `string`. The return value has the type `size_t`, which is a type definition declared in the `stddef.h` header file.

Examples

The following program contains two examples of the `strlen` function. In each example, the function calculates the length of a specified string. In the first example, the `string` argument is a pointer to `char`. In the second example, the `string` argument is a pointer to a `char_varying` string.

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

char nstr[30] = "123456789";
char_varying(30) vstr = "12345";

main()
{
   size_t length = 0;

   length = strlen(nstr);                           /*  Example 1  */
   printf("Example 1:  length = %u\n", length);

   length = strlen_vstr(&vstr);                     /*  Example 2  */
   printf("Example 2:  length = %u\n", length);
}
```

The output from the preceding program is as follows:

```
Example 1:  length = 9
Example 2:  length = 5
```

Related Functions

None

strncat

Purpose

Appends a specified number of characters from one string to another string.

Syntax

```
#include <string.h>

char          *strncat(...);   /*  Types of the arguments can vary  */

char          *strncat_nstr_vstr(char *s1, const char_varying *v2, size_t
n);
char_varying *strncat_vstr_nstr(char_varying *v1, const char *s2, size_t
n);
char_varying *strncat_vstr_vstr(char_varying *v1, const char_varying *v2,
                                size_t n);
```

Explanation

A sample invocation of the `strncat` function is as follows:

```
return_value = strncat(string1, string2, n);
```

In VOS C, the `strncat` function is a generic, string-manipulation function that appends not more than `n` characters from the string pointed to by `string2` to the end of the string pointed to by `string1`. The `string1` and `string2` arguments can be either pointers to `char` or pointers to `char_varying` strings, or a combination of the two pointer types. The `return_value` is a pointer to `string1`.

The `n` argument specifies the number of characters to append. The `n` argument is a value of the type `size_t`, which is a type definition declared in the `stddef.h` header file.

With the generic, string-manipulation functions, the VOS C compiler takes a special action if either `string1` or `string2` contains the address of a `char_varying` string. Based on the data types of the function's arguments, the compiler translates the name of the function (`strncat`) into the name of the actual function that is invoked. The following table lists the actual function names that the compiler uses and the data types of each function's `string1` argument, `string2` argument, `n` argument, and return value.

Actual Function Name	**string1**	**string2**	**n**	**Return Value**
`strncat`	`char *`	`char *`	`size_t`	`char *`
`strncat_nstr_vstr`	`char *`	`char_varying(`*`l`*`) *`	`size_t`	`char *`
`strncat_vstr_nstr`	`char_varying(`*`l`*`) *`	`char *`	`size_t`	`char_varying *`
`strncat_vstr_vstr`	`char_varying(`*`l`*`) *`	`char_varying(`*`m`*`) *`	`size_t`	`char_varying *`

When you invoke the function, you can specify `strncat` or the actual function name in your program. For example, if `string1` is a pointer to a `char_varying` string and `string2` is a pointer to `char`, you can invoke the function using `strncat`, or the actual function name, `strncat_vstr_nstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype.

The length of `string2` affects what characters are appended. When `string2` is shorter than `n`, the number of characters appended equals the length of `string2`. The length of `string2` is determined as follows:

- If `string2` is a pointer to `char`, the length of the string equals the number of characters preceding the first null character in `string2`.
- If `string2` is a pointer to a `char_varying` string, the length of the string equals its current length.

The length of the resulting string, `string1`, is determined as follows:

- If `string1` is a pointer to `char`, `strncat` overwrites `string1`'s terminating null character with the initial character of `string2` as characters from `string2` are appended. The `strncat` function then appends a null character to the resulting string. After the append operation, the maximum length of `string1` will be as follows:
- strlen(string1) + n + 1
- If `string1` is a pointer to a `char_varying` string, `strncat` appends the characters from `string2`, but does not add a null character to the resulting string.

Be sure that `string1` has enough space to hold `string2`, including the null character if one is appended. **If** `string1` does not have enough space to hold `string2`, other data in your program can be corrupted. Also, if the copying takes place between two objects that overlap, the `strncat` function can have unpredictable results.

Return Value

The `strncat` function returns a pointer to `string1`. The return value has the type pointer to `char` if `string1` is a pointer to `char`, or has the type pointer to `char_varying` if `string1` is a pointer to a `char_varying` string.

Examples

The following program contains two examples of the `strncat` function. In each example, a specified number of characters from one string is appended to another string. In the first example, each argument to `strncat` is a pointer to `char`. In the second example, each argument is a pointer to a `char_varying` string.

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

char nstr_1[20] = "XXXXXXXXXX";
char nstr_2[10] = "AAAAA";

char_varying(20) vstr_1 = "XXXXXXXXXX";
char_varying(10) vstr_2 = "BBBBB";
char_varying *vstr_1_ptr;

size_t n;

main()
{
    n = 5;
   strncat(nstr_1, nstr_2, n);                         /*  Example 1  */
    printf("%s\n", nstr_1);

    n = 5;
   vstr_1_ptr = strncat_vstr_vstr(&vstr_1, &vstr_2, n);    /*  Example
2  */
    printf("%v\n", vstr_1_ptr);
}
```

The output from the preceding program is as follows:

```
XXXXXXXXXXAAAAA
XXXXXXXXXXBBBBB
```

Related Functions

`strcat`, `strcpy`, `strncpy`

strncmp

Purpose

Compares a specified number of characters from one string with characters from another string.

Syntax

```
#include <string.h>

int strncmp(...);         /*  Types of the arguments can vary  */

int strncmp_nstr_vstr(const char *s1, const char_varying *v2, size_t
n);
int strncmp_vstr_nstr(const char_varying *v1, const char *s2, size_t
n);
int strncmp_vstr_vstr(const char_varying *v1, const char_varying *v2,
                      size_t n);
```

Explanation

A sample invocation of the `strncmp` function is as follows:

```
return_value = strncmp(string1, string2, n);
```

In VOS C, the `strncmp` function is a generic, string-manipulation function that compares not more than `n` characters from the string pointed to by `string1` with characters from the string pointed to by `string2`. The `string1` and `string2` arguments can be either pointers to `char` or pointers to `char_varying` strings, or a combination of the two pointer types.

The `n` argument specifies the number of characters to compare. The `n` argument is a value of the type `size_t`, which is a type definition declared in the `stddef.h` header file.

With the generic, string-manipulation functions, the VOS C compiler takes a special action if either `string1` or `string2` contains the address of a `char_varying` string. Based on the data types of the function's arguments, the compiler translates the name of the function (`strncmp`) into the name of the actual function that is invoked. The following table lists the actual function names that the compiler uses and the data types of each function's `string1` argument, `string2` argument, `n` argument, and return value.

Actual Function Name	**string1**	**string2**	**n**	**Return Value**
`strncmp`	`char *`	`char *`	`size_t`	`int`
`strncmp_nstr_vstr`	`char *`	`char_varying(`*`l`*`) *`	`size_t`	`int`
`strncmp_vstr_nstr`	`char_varying(`*`l`*`) *`	`char *`	`size_t`	`int`
`strncmp_vstr_vstr`	`char_varying(`*`l`*`) *`	`char_varying(`*`m`*`) *`	`size_t`	`int`

When you invoke the function, you can specify `strncmp` or the actual function name in your program. For example, if `string1` is a pointer to a `char_varying` string and `string2` is a pointer to `char`, you can invoke the function using `strncmp`, or the actual function name, `strncmp_vstr_nstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype.

The lengths of the strings, `string1` and `string2`, are determined as follows:

- If the string is a pointer to `char`, the length of the string equals the number of characters preceding the first null character in `string2`.
- If the string is a pointer to a `char_varying` string, the length of the string equals its current length.

The `strncmp` function uses the following procedure to compare `string1` and `string2`. The strings are compared, character by character, until not more than `n` characters have been compared or until two characters are found that are different. If `strncmp` finds two characters that are different, it compares the two characters and returns a value based on which of the two characters' ASCII codes has the greater numeric value. If the two strings compare equal up to the length of the shorter string, the longer string always compares greater.

Return Value

The `strncmp` function returns a value based on the comparison of the first `n` characters in the strings pointed to by `string1` and `string2`. Based on the results of the comparison, `strncmp` returns an `int` value that is greater than 0, less than 0, or equal to 0. See the following table.

Return Value	**Result of the Comparison**
Greater than 0	When two characters are found that are different, the character in the string pointed to by `string1` has a greater ASCII code than the character in the string pointed to by `string2`.
Less than 0	When two characters are found that are different, the character in the string pointed to by `string1` has a smaller ASCII code than the character in the string pointed to by `string2`.
Equal to 0	Every character in the string pointed to by `string1` is identical to the corresponding character in the string pointed to by `string2`.

Examples

The following program contains two examples of the `strncmp` function. In each example, `n` characters from one string are compared with characters from another string. In the first example, each argument to `strncmp` is a pointer to `char`. In the second example, one argument is a pointer to a `char_varying` string, and the other argument is a pointer to `char`.

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

char nstr_1[10] = "ABCDEFGhi";
char nstr_2[10] = "ABCDEFGHI";

char_varying(10) vstr_1 = "ABCDEFGHi";

size_t n = 8;
int return_value;

main()
{
   return_value = strncmp(nstr_1, nstr_2, n);                 /*
Example 1  */

   if (return_value > 0)
      printf("nstr_1 is greater than nstr_2.\n");

   if (return_value < 0)
      printf("nstr_2 is greater than nstr_1.\n");

   if (return_value == 0)
      printf("nstr_1 and nstr_2 are equal.\n");
```

(Continued on next page)

```
   return_value = strncmp_vstr_nstr(&vstr_1, nstr_2, n);      /*
Example 2  */

   if (return_value > 0)
      printf("vstr_1 is greater than nstr_2.\n");

   if (return_value < 0)
      printf("nstr_2 is greater than vstr_1.\n");

   if (return_value == 0)
      printf("vstr_1 and nstr_2 are equal.\n");
}
```

The output from the preceding program is as follows:

```
nstr_1 is greater than nstr_2.
vstr_1 and nstr_2 are equal.
```

Notice that, in the preceding program, only the first eight characters of the strings are compared because in both `strncmp` calls `n` equals the value 8.

Related Functions

`memcmp`, `strcmp`

strncpy

Purpose

Copies a specified number of characters from one string into another string.

Syntax

```
#include <string.h>

char        *strncpy(...);   /*  Types of the arguments can vary  */

char         *strncpy_nstr_vstr(char *s1, const char_varying *v2,
size_t n);
char_varying *strncpy_vstr_nstr(char_varying *v1, const char *s2,
size_t n);
char_varying *strncpy_vstr_vstr(char_varying *v1, const char_varying
*v2,
                                size_t n);
```

Explanation

A sample invocation of the `strncpy` function is as follows:

```
return_value = strncpy(string1, string2, n);
```

In VOS C, the `strncpy` function is a generic, string-manipulation function that copies not more than `n` characters from the string pointed to by `string2` into the string pointed to by `string1`. The `string1` and `string2` arguments can be either pointers to `char` or pointers to `char_varying` strings, or a combination of the two pointer types. The `return_value` is a pointer to `string1`.

The `n` argument specifies the number of characters to copy. The `n` argument is a value of the type `size_t`, which is a type definition declared in the `stddef.h` header file.

With the generic, string-manipulation functions, the VOS C compiler takes a special action if either `string1` or `string2` contains the address of a `char_varying` string. Based on the data types of the function's arguments, the compiler translates the name of the function (`strncpy`) into the name of the actual function that is invoked. The following table lists the actual function names that the compiler uses and the data types of each function's `string1` argument, `string2` argument, `n` argument, and return value.

Actual Function Name	**string1**	**string2**	**n**	**Return Value**
`strncpy`	`char *`	`char *`	`size_t`	`char *`
`strncpy_nstr_vstr`	`char *`	`char_varying(`*`l`*`) *`	`size_t`	`char *`
`strncpy_vstr_nstr`	`char_varying(`*`l`*`) *`	`char *`	`size_t`	`char_varying *`
`strncpy_vstr_vstr`	`char_varying(`*`l`*`) *`	`char_varying(`*`m`*`) *`	`size_t`	`char_varying *`

When you invoke the function, you can specify `strncpy` or the actual function name in your program. For example, if `string1` is a pointer to a `char_varying` string and `string2` is a pointer to `char`, you can invoke the function using `strncpy`, or the actual function name, `strncpy_vstr_nstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype.

The `strncpy` function copies `n` characters if the length of `string2` is at least `n` characters long. The length of the source string, `string2`, is determined as follows:

- If `string2` is a pointer to `char`, the length of the string equals the number of characters preceding the first null character. Characters after the null character are not copied.
- If `string2` is a pointer to a `char_varying` string, the length of the string equals its current length.

The length of the resulting string, `string1`, is determined as follows:

- If `string1` is a pointer to `char`, the first character of `string2` overwrites the first character in `string1`. If the length of `string2` is shorter than `n`, null characters are copied into `string1` until at least `n` characters have been written. A terminating null character is copied into the resulting string **except when** one of the following conditions is true.
  - `string2` is a pointer to `char` and no null character appears in the first `n` characters of `string2`.
  - `string2` is a pointer to a `char_varying` string and `n` is less than or equal to the current length of `string2`.
- If `string1` is a pointer to a `char_varying` string, the first character of `string2` overwrites the first character in `string1`. The length of the string that is copied into `string1` equals either `n` characters or the length of `string2`, whichever is less. No null character is copied into the resulting string.

Be sure that `string1` has enough space to hold `string2`, including the null character if one is appended. **If** `string1` does not have enough space to hold `string2`, other data in your program can be corrupted. Also, if the copying takes place between two objects that overlap, the `strncpy` function can have unpredictable results.

Return Value

The `strncpy` function returns a pointer to `string1`. The return value has the type pointer to `char` if `string1` is a pointer to `char`, or has the type pointer to `char_varying` if `string1` is a pointer to a `char_varying` string.

Examples

The following program contains two examples of the `strncpy` function. In each example, a specified number of characters from one string is copied into another string. In the first example, each argument to `strncpy` is a pointer to `char`. In the second example, one argument is a pointer to `char`, and the other argument is a pointer to a `char_varying` string.

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

char nstr_1[20], nstr_2[20] = "AAAAAAAAAA";
char *nstr_1_ptr;

char_varying(20) vstr_2 = "BBBBBBBBBB";

size_t n;

main()
{
   strcpy(nstr_1, "XXXXXXXXXXXXXXXXXXXX");
   n = 11;
   strncpy(nstr_1, nstr_2, n);                     /*  Example 1  */
   printf("Example 1:  %s\n", nstr_1);

   strcpy(nstr_1, "XXXXXXXXXXXXXXXXXXXX");
   n = 10;
   nstr_1_ptr = strncpy_nstr_vstr(nstr_1, &vstr_2, n);  /*  Example
2  */
   printf("Example 2:  %s\n", nstr_1);
}
```

The output from the preceding program is as follows:

```
Example 1:  AAAAAAAAAA
Example 2:  BBBBBBBBBBXXXXXXXXXX
```

Notice that, in example 2, `strncpy` does not append a null character to `nstr_1` because `n` is equal to the current length of `vstr_2`.

Related Functions

`memcpy`, `memset`, `strcat`, `strcpy`, `strncat`

strpbrk

Purpose

Searches for the first occurrence, within one string, of any of the characters from another string.

Syntax

```
#include <string.h>

char *strpbrk(...);       /*  Types of the arguments can vary  */

char *strpbrk_nstr_vstr(const char *s1, const char_varying *v2);
char *strpbrk_vstr_nstr(const char_varying *v1, const char *s2);
char *strpbrk_vstr_vstr(const char_varying *v1, const char_varying
*v2);
```

Explanation

A sample invocation of the `strpbrk` function is as follows:

```
return_value = strpbrk(string1, string2);
```

In VOS C, the `strpbrk` function is a generic, string-manipulation function that searches for the first occurrence, within the string pointed to by `string1`, of any character from the string pointed to by `string2`. The `string1` and `string2` arguments can be either pointers to `char` or pointers to `char_varying` strings, or a combination of the two pointer types.

With the generic, string-manipulation functions, the VOS C compiler takes a special action if either `string1` or `string2` contains the address of a `char_varying` string. Based on the data types of the function's arguments, the compiler translates the name of the function (`strpbrk`) into the name of the actual function that is invoked. The following table lists the actual function names that the compiler uses and the data types of each function's `string1` argument, `string2` argument, and return value.

Actual Function Name	string1	string2	Return Value
`strpbrk`	`char *`	`char *`	`char *`
`strpbrk_nstr_vstr`	`char *`	`char_varying(`*`l`*`) *`	`char *`
`strpbrk_vstr_nstr`	`char_varying(`*`l`*`) *`	`char *`	`char *`
`strpbrk_vstr_vstr`	`char_varying(`*`l`*`) *`	`char_varying(`*`m`*`) *`	`char *`

When you invoke the function, you can specify `strpbrk` or the actual function name in your program. For example, if `string1` is a pointer to a `char_varying` string and `string2` is a pointer to `char`, you can invoke the function using `strpbrk`, or the actual function name, `strpbrk_vstr_nstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype.

The lengths of the strings, `string1` and `string2`, are determined as follows:

- If the string is a pointer to `char`, the length of the string equals the number of characters preceding the first null character.
- If string is a pointer to a `char_varying` string, the length of the string equals its current length.

Return Value

If `strpbrk` finds any of the characters from `string2`, the function returns a pointer to the located character within `string1`. The return value has the type pointer to `char`.

If `strpbrk` does not find any of the characters from `string2`, it returns the `NULL` pointer constant.

Examples

The following program contains two examples of the `strpbrk` function. In each example, `strpbrk` searches one string for characters from another string. In the first example, each argument to `strpbrk` is a pointer to `char`. In the second example, one argument is a pointer to a `char_varying` string, and the other argument is a pointer to `char`.

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

char nstr_1[20] = "abcdefghijklmn";
char *nstr_2;
char_varying(20) vstr = "amount:  $9.99";

char_varying(20) substring;
unsigned int offset;

main()
{
   char *char_ptr;

   nstr_2 = "njl";
   char_ptr = strpbrk(nstr_1, nstr_2);             /*  Example 1  */
   if (char_ptr != NULL)
      printf("Example 1:  %s\n", char_ptr);
```

(Continued on next page)

```
          nstr_2 = "0123456789";
         char_ptr = strpbrk_vstr_nstr(&vstr, nstr_2);       /*  Example 2  */
          if (char_ptr != NULL)
             {
             offset = (char_ptr - &vstr) - 1;
             substring = $substr(vstr, offset);
             printf("Example 2:  %v\n", &substring);
             }
      }
```

The output from the preceding program is as follows:

```
Example 1:  jklmn
Example 2:  9.99
```

See the `strchr` function for information on how to use the `$substr` built-in function (as in example 2) to extract the `char_varying` substring whose initial character is pointed to by the return value of `strpbrk`.

Related Functions

`memchr`, `strchr`, `strcspn`, `strrchr`, `strspn`, `strstr`, `strtok`

strrchr

Purpose

Searches for the last occurrence of a specified character within a given string.

Syntax

```
#include <string.h>

char *strrchr(...);      /*  Types of the arguments can vary  */

char *strrchr_vstr(const char_varying *v, int c);
```

Explanation

A sample invocation of the `strrchr` function is as follows:

```
return_value = strrchr(string, c);
```

In VOS C, the `strrchr` function is a generic, string-manipulation function that searches for the last occurrence of the character specified by `c` (converted to an `unsigned char`) within the string pointed to by `string`. The `string` argument can be either a pointer to `char` or a pointer to a `char_varying` string. If `string` is a pointer to `char`, the string's null character is part of the sequence of characters searched.

With the generic, string-manipulation functions, the VOS C compiler takes a special action if `string` contains the address of a `char_varying` string. In this case, the compiler translates the name of the function (`strrchr`) into the name of the actual function that is invoked. The following table lists the actual function names that the compiler uses and the data types of each function's `string` argument, `c` argument, and return value.

Actual Function Name	**string**	**c**	**Return Value**
`strrchr`	`char *`	`int`	`char *`
`strrchr_vstr`	`char_varying(`*`l`*`) *`	`int`	`char *`

When you invoke the function, you can specify `strrchr` or the actual function name in your program. For example, if `string` is a pointer to a `char_varying` string, you can invoke the function using `strrchr`, or the actual function name, `strrchr_vstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype.

The length of the string that `strrchr` searches is determined as follows:

- If `string` is a pointer to `char`, the length of the string equals the number of characters up to and including the first null character.
- If `string` is a pointer to a `char_varying` string, the length of the string equals its current length.

Return Value

If `strrchr` finds the specified character, the function returns a pointer to the located character within `string`. The return value has the type pointer to `char`.

If `strrchr` does not find the specified character, it returns the `NULL` pointer constant.

Examples

The following program contains two examples of the `strrchr` function. In each example, the function searches for the last occurrence of an `X` within a specified string. In the first example, the `string` argument is a pointer to `char`. In the second example, the `string` argument is a pointer to a `char_varying` string.

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

char nstr[30] = "abcdXefghijXklmn";
char_varying(30) vstr = "abcdXefghiXjklmn";
char_varying(30) substring;

int c = 'X';
unsigned int offset;

main()
{
   char *char_ptr;

   char_ptr = strrchr(nstr, c);                    /*  Example 1  */
   if (char_ptr != NULL)
      printf("Example 1:  %s\n", char_ptr);

   char_ptr = strrchr_vstr(&vstr, c);              /*  Example 2  */
   if (char_ptr != NULL)
      {
      offset = (char_ptr - &vstr) - 1;
      substring = $substr(vstr, offset);
      printf("Example 2:  %v\n", &substring);
      }
}
```

The output from the preceding program is as follows:

```
Example 1:  Xklmn
Example 2:  Xjklmn
```

See the `strchr` function for information on how to use the `$substr` built-in function (as in example 2) to extract the `char_varying` substring whose initial character is pointed to by the return value of `strrchr`.

Related Functions

`memchr`, `strchr`, `strcspn`, `strpbrk`, `strspn`, `strstr`, `strtok`

strspn

Purpose

Computes the length of the initial segment of a specified string that consists entirely of characters from another string.

Syntax

```
#include <string.h>

size_t strspn(...);      /*  Types of the arguments can vary  */

size_t strspn_nstr_vstr(const char *s1, const char_varying *v2);
size_t strspn_vstr_nstr(const char_varying *v1, const char *s2);
size_t strspn_vstr_vstr(const char_varying *v1, const char_varying
*v2);
```

Explanation

A sample invocation of the `strspn` function is as follows:

```
return_value = strspn(string1, string2);
```

In VOS C, the `strspn` function is a generic, string-manipulation function that calculates the length of the initial segment of the string pointed to by `string1` that consists entirely of characters from the string pointed to by `string2`. The `string1` and `string2` arguments can be either pointers to `char` or pointers to `char_varying` strings, or a combination of the two pointer types.

With the generic, string-manipulation functions, the VOS C compiler takes a special action if either `string1` or `string2` contains the address of a `char_varying` string. Based on the data types of the function's arguments, the compiler translates the name of the function (`strspn`) into the name of the actual function that is invoked. The following table lists the actual function names that the compiler uses and the data types of each function's `string1` argument, `string2` argument, and return value.

Actual Function Name	**string1**	**string2**	**Return Value**
`strspn`	`char *`	`char *`	`size_t`
`strspn_nstr_vstr`	`char *`	`char_varying(`*`l`*`) *`	`size_t`
`strspn_vstr_nstr`	`char_varying(`*`l`*`) *`	`char *`	`size_t`
`strspn_vstr_vstr`	`char_varying(`*`l`*`) *`	`char_varying(`*`m`*`) *`	`size_t`

When you invoke the function, you can specify `strspn` or the actual function name in your program. For example, if `string1` is a pointer to a `char_varying` string and `string2` is a pointer to `char`, you can invoke the function using `strspn`, or the actual function name, `strspn_vstr_nstr`. However, it is **recommended** that you use the actual function name so that the compiler can check the number and types of the arguments in the function call against the information in the function prototype.

The lengths of the strings, `string1` and `string2`, are determined as follows:

- If the string is a pointer to `char`, the length of the string equals the number of characters preceding the first null character.
- If the string is a pointer to a `char_varying` string, the length of the string equals its current length.

Return Value

The `strspn` function returns the length of the initial segment of `string1` that consists entirely of characters from `string2`. The return value has the type `size_t`, which is a type definition declared in the `stddef.h` header file.

Examples

The following program contains two examples of the `strspn` function. In each example, `strspn` calculates the length of the initial segment of one string consisting entirely of characters from another string. In the first example, both arguments to `strspn` are pointers to `char`. In the second example, one argument is a pointer to `char`, and the other argument is a pointer to a `char_varying` string.

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

char nstr_1[30] = "\t there's magic in the web of it";
char *nstr_2;

char_varying(30) vstr;

size_t segment_length;

main()
{
   nstr_2 = "\t \n";
   segment_length = strspn(nstr_1, nstr_2);              /*  Example
1  */
   printf("Example 1:  segment_length = %u\n", segment_length);

   strcpy(vstr, "\t\n abcdefghijklmnopqrstuvwxyz");
  segment_length = strspn_nstr_vstr(nstr_1, &vstr);      /*  Example
2  */
   printf("Example 2:  segment_length = %u\n", segment_length);
}
```

The output from the preceding program is as follows:

```
Example 1:  segment_length = 2
Example 2:  segment_length = 7
```

Related Functions

`memchr`, `strchr`, `strcspn`, `strpbrk`, `strrchr`, `strstr`, `strtok`

strstr

Purpose

Searches for the first occurrence of a specified sequence of characters within a given string.

Syntax

```
#include <string.h>

char *strstr(const char *string1, const char *string2);
```

Explanation

The `strstr` function searches for the first occurrence of the sequence of characters pointed to by `string2` within the string pointed to by `string1`. The sequence of characters for which `strstr` searches does **not** include the null character that terminates the string in `string2`.

Return Value

If `strstr` finds the sequence of characters pointed to by `string2`, the function returns a pointer to the first character of the searched-for string within `string1`.

If `strstr` does not find the specified sequence of characters, it returns the `NULL` pointer constant. When `string2` points to a string with zero length, `strstr` returns the string pointed to by `string1`.

Examples

The following program uses `strstr` to find the first occurrence of the sequence of characters `defg` within a specified string.

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

char string1[15] = "abcdefghijklmn";
char string2[5] = "defg";

main()
{
   char *returned_ptr;

   returned_ptr = strstr(string1, string2);

   printf("The first occurrence starts here:  %s\n", returned_ptr);
}
```

The output from the preceding program is as follows:

```
The first occurrence starts here:  defghijklmn
```

Related Functions

`memchr`, `strchr`, `strcspn`, `strpbrk`, `strrchr`, `strspn`, `strtok`

strtod

Purpose

Converts a string of ASCII characters into a `double` value.

Syntax

```
#include <stdlib.h>

double strtod(const char *string, char **end_ptr);
```

Explanation

The `strtod` function converts a string pointed to by `string` into a value of the type `double`.

The `string` argument points to a `char`, the first character of the string to convert. The `end_ptr` argument is the address of a pointer to a `char`. When `strtod` returns, `end_ptr` points to the character in the string where the conversion process stopped.

The string contains the ASCII character representation of a floating-point number. The string can contain leading white-space characters, an optional sign, and a string of digits optionally containing a decimal point. It can end with an optional `e` or `E` followed by an optionally signed integer. The following examples are valid character representations of floating-point numbers that `strtod` could convert. In the third example, the symbol   indicates a space character.

```
6E-10

-6.0e+6

 123.987
```

The conversion process begins with the first character pointed to by `string`. If `strtod` encounters a character that it does not recognize, it ends the conversion. On return, `strtod` assigns to `end_ptr` the address of the character where the representation of the floating-point value stopped, or where the conversion ended. The character pointed to by `end_ptr` can be the null character (`\0`). Using the value returned in `end_ptr`, you can process the remainder of the string, if needed.

> **Note:** When you pass `NULL` or `OS_NULL_PTR` instead of an `end_ptr` argument, the `strtod` function assigns no address to `end_ptr`.

If the return value creates an overflow or underflow condition, `strtod` sets `errno` to `ERANGE`, which is a constant defined in the `math.h` header file.

Return Value

The `strtod` function returns the converted string as a value of the type `double`.

If the string begins with an unrecognized character, `strtod` returns 0. On underflow, `strtod` returns 0. On overflow, `strtod` returns positive or negative `HUGE_VAL` depending on the return value's sign.

Examples

The following program uses `strtod` to convert four strings into floating-point numbers.

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

double d;
char *string;
char *end_ptr;

main()
{
  string = "  1.23456e3ABC";                  /*  Example 1
*/
   d = strtod(string, &end_ptr);
  printf("d = %6.2lf\n", d);                  /*  d = 1234.56
*/
   printf("Rest of string = %s\n", end_ptr);      /*  Rest of string
= ABC  */

  string = "1.23456e789ABC";                  /*  Example 2
*/
   d = strtod(string, &end_ptr);
  printf("d = %6.2lf\n", d);                  /*  d = INFINITY
*/
   printf("Rest of string = %s\n", end_ptr);      /*  Rest of string
= ABC  */

  string = "1.23456e-789ABC";                 /*  Example 3
*/
   d = strtod(string, &end_ptr);
  printf("d = %6.2lf\n", d);                 /*  d =   0.00          */
   printf("Rest of string = %s\n", end_ptr);      /*  Rest of string
= ABC  */

  string = "s1.23e3ABC";               /*  Example 4               */
   d = strtod(string, &end_ptr);
  printf("d = %6.2lf\n", d);            /*  d =   0.00              */
   printf("Rest of string = %s\n", end_ptr); /*  Rest of string =
s1.23e3ABC */
}
```

In the preceding program, Examples 2 through 4 generate the output shown for the following reasons.

- Example 2 creates an overflow condition. Therefore, the return value equals `INFINITY`.
- Example 3 creates an underflow condition. Therefore, the return value equals `0.00`.
- Example 4 attempts to convert an invalid character (`s`) as part of the string. Therefore, the return value equals `0.00`.

Related Functions

`atof`, `strtol`, `strtoul`

`strtok`

Purpose

Breaks a string into a sequence of tokens, each of which is delimited by a character from another string.

Syntax

```
#include <string.h>

char *strtok(char *string1, const char *string2);
```

Explanation

When called repeatedly, the `strtok` function breaks the string pointed to by `string1` into a sequence of zero or more tokens, each of which is delimited by a character from the string pointed to by `string2`.

The first call to `strtok` searches the string pointed to by `string1` for the first character that is **not** contained in the current separator string pointed to by `string2`.

If no such character is found in `string1`, then `string1` contains no tokens. The `strtok` function returns the `NULL` pointer constant.

If such a character is found, the character is the start of the first token in `string1`. Then, `strtok` searches from that character for a character that **is** contained in the current separator string pointed to by `string2`.

- If no such character is found, the current token extends to the end of `string1`. In subsequent searches for a token, `strtok` returns the `NULL` pointer constant.
- If such a character is found, `strtok` overwrites the character in `string1` with a null character, which terminates the current token. In addition, `strtok` saves a pointer to the character following the null character, from which the next search for a token will start.

Whenever `strtok` finds a token, the function returns a pointer to the token's first character.

Each subsequent call to `strtok`, with `string1` equal to the `NULL` pointer constant, starts searching from the saved pointer and behaves as described in the preceding explanation.

In brief, the procedure for using `strtok` to divide a string into a series of tokens is as follows:

1. Call `strtok` with `string1` specifying the string to divide into tokens and `string2` specifying the characters that can delimit the tokens.

2. **2.** Call `strtok` as many times as needed until the string has been completely tokenized. In these calls to `strtok`, the `string1` argument equals the `NULL` pointer constant and `string2` specifies one or more characters that can delimit the given token. The characters specified for `string2` can be different from call to call.

Notice that `strtok` modifies `string1` when the function writes a null character in the position of the delimiter following a token.

Return Value

If `strtok` finds a token, the function returns a pointer to the token's first character.

If `strtok` does not find a token, the function returns the `NULL` pointer constant.

Examples

The following program uses `strtok` to divide a string into a series of tokens, each of which is delimited by characters from another string.

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

char string1[] = "    aaaaaa: bbbbb, cccc;";
char string2[] = "\t ;,:";

char *token[10];

main()
{
   unsigned int i = 0;

   token[i] = strtok(string1, string2);

   while (token[i] != NULL)
      {
      printf("token[%u] =  %s\n", i, token[i]);

      token[++i] = strtok(NULL, string2);
      }
}
```

The output from the preceding program is as follows:

```
token[0] =  aaaaaa
token[1] =  bbbbb
token[2] =  cccc
```

Related Functions

`memchr`, `strchr`, `strcspn`, `strpbrk`, `strrchr`, `strspn`, `strstr`

strtol

Purpose

Converts a string of ASCII characters into a `long int` value.

Syntax

```
#include <stdlib.h>

long strtol(const char *string, char **end_ptr, int base);
```

Explanation

The `strtol` function converts a string pointed to by `string` into an integer value of the type `long int`.

The `string` argument points to a `char`, the first character of the string to convert. The `end_ptr` argument is the address of a pointer to a `char`. When `strtol` returns, `end_ptr` points to the character in the string where the conversion process stopped. The `base` argument is an `int` value specifying the radix, or base, of the character representation of the integer value that will be converted.

The string contains the ASCII character representation of an integer. The string can contain leading white-space characters, an optional sign, and a string of digits. The syntax for the string is as follows:

$$\left[\texttt{whitespace}\right]\left[\begin{array}{c}\texttt{-}\\ \texttt{+}\end{array}\right]\left\{\begin{array}{l}\texttt{0x}\ \textit{hexadecimal\_digits}\\ \texttt{0X}\ \textit{hexadecimal\_digits}\\ \quad\texttt{0}\ \textit{octal\_digits}\\ \quad\textit{decimal\_digits}\end{array}\right\}$$

If `base` equals 0, `strtol` uses the first characters of the string following the sign, if any, to determine the base of the value to convert. The following table shows the first character or characters that specify a decimal, octal, or hexadecimal base. It also shows the subsequent characters that you can use as *`decimal_digits`*, *`octal_digits`*, and *`hexadecimal_digits`*.

First Character(s)	**Base**	**Subsequent Characters Allowed in the Value**
`1` to `9`	decimal	`0` to `9`
`0`	octal	`0` to `7`
`0x` or `0X`	hexadecimal	`0` to `9`, `a` to `f`, and `A` to `F`

If `base` equals a value in the range 2 through 36, `strtol` uses `base` to determine the base of the value to convert. In this case, after an optional leading sign, leading zeroes are ignored. If `base` is greater than 9, the uppercase letters `A` through `Z` and the lowercase letters `a` through `z` represent, in sequence, the digits after `9`. The `strtol` function recognizes as valid only those letters that can be used to represent values in the specified base.

The following examples are valid character representations of integers that `strtol` could convert. In the first example, the symbol   indicates a space character.

```
  -6789      /*  Decimal base */
+060         /*  Octal base */
0XFFFFFFFF   /*  Hexadecimal base  */
```

The conversion process begins with the first character pointed to by `string`. If `strtol` encounters a character that it does not recognize, it ends the conversion. On return, `strtol` assigns to `end_ptr` the address of the character where the representation of the integer value stopped, or where the conversion ended. The character pointed to by `end_ptr` can be the null character (`\0`). Using the value returned in `end_ptr`, you can process the remainder of the string, if needed.

> **Note:** When you pass `NULL` instead of an `end_ptr` argument, `strtol` assigns no address to `end_ptr`.

When the string is not in the correct format or the base is invalid, `strtol` assigns the address of `string` to `end_ptr`.

If the return value is outside the range of a `long int`, `strtol` sets `errno` to `ERANGE`, which is a constant defined in the `math.h` header file.

Return Value

The `strtol` function returns the converted string as a value of the type `long int`.

If the string is not in the correct format or the base is invalid, `strtol` returns the value 0. When the converted value is outside the range of a `long int`, `strtol` returns `LONG_MIN` if the value is negative or `LONG_MAX` if the value is positive. Both `LONG_MIN` and `LONG_MAX` are defined in the `limits.h` header file.

If `base` equals 0 or 16 and the first characters are `0x` or `0X` followed by a character that is not a valid hexadecimal digit, the return value equals 0. In this case, `end_ptr` points to the `x` or `X`.

Examples

The following program uses `strtol` to convert four strings into values of the type `long int`.

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

long int long_i;
char *string;
char *end_ptr;
int base = 0;

main()
{
   string = "  -12345ABC";
   long_i = strtol(string, &end_ptr, base);
  printf("long_i = %d\n", long_i);           /*  long_i = -12345
*/
   printf("Rest of string = %s\n", end_ptr);      /*  Rest of string
= ABC  */

   string = "-0XFFFFGHI";
   long_i = strtol(string, &end_ptr, base);
  printf("long_i = %d\n", long_i);           /*  long_i = -65535
*/
   printf("Rest of string = %s\n", end_ptr);      /*  Rest of string
= GHI  */

   string = "060ABC";
   long_i = strtol(string, &end_ptr, base);
  printf("long_i = %d\n", long_i);            /*  long_i = 48
*/
   printf("Rest of string = %s\n", end_ptr);      /*  Rest of string
= ABC  */

   string = " 123.456ABC";
   long_i = strtol(string, &end_ptr, base);
  printf("d = %d\n", long_i);              /*  long_i = 123
*/
   printf("Rest of string = %s\n", end_ptr);    /*  Rest of string =
.456ABC */
}
```

Related Functions

`atoi`, `atol`, `strtod`, `strtoul`

`strtoul`

Purpose

Converts a string of ASCII characters into an `unsigned long` value.

Syntax

```
#include <stdlib.h>

unsigned long strtoul(const char *string, char **end_ptr, int base);
```

Explanation

The `strtoul` function converts a string pointed to by `string` into an integer value of the type `unsigned long`.

The `string` argument points to a `char`, the first character of the string to convert. The `end_ptr` argument is the address of a pointer to a `char`. When `strtoul` returns, `end_ptr` points to the character in the string where the conversion process stopped. The `base` argument is an `int` value specifying the radix, or base, of the character representation of the integer value that will be converted.

The string contains the ASCII character representation of an integer. The string can contain leading white-space characters, an optional sign, and a string of digits. The syntax for the string is as follows:

$$\left[\texttt{whitespace}\right]\left[\begin{array}{c}\texttt{-}\\ \texttt{+}\end{array}\right]\left\{\begin{array}{l}\texttt{0x}\ \textit{hexadecimal\_digits}\\ \texttt{0X}\ \textit{hexadecimal\_digits}\\ \quad\texttt{0}\ \textit{octal\_digits}\\ \quad\textit{decimal\_digits}\end{array}\right\}$$

If `base` equals 0, `strtoul` uses the first characters of the string following the sign, if any, to determine the base of the value to convert. The following table shows the first character or characters that specify a decimal, octal, or hexadecimal base. It also shows the subsequent characters that you can use as *`decimal_digits`*, *`octal_digits`*, and *`hexadecimal_digits`*.

First Character(s)	**Base**	**Subsequent Characters Allowed in the Value**
`1` to `9`	decimal	`0` to `9`
`0`	octal	`0` to `7`
`0x` or `0X`	hexadecimal	`0` to `9`, `a` to `f`, and `A` to `F`

If `base` equals a value in the range 2 through 36, `strtoul` uses `base` to determine the base of the value to convert. In this case, after an optional leading sign, leading zeroes are ignored. If `base` is greater than 9, the uppercase letters `A` through `Z` and the lowercase letters `a` through `z` represent, in sequence, the digits after `9`. The `strtoul` function recognizes as valid only those letters that can be used to represent values in the specified base.

The following examples are valid character representations of integers that `strtoul` could convert. In the first example, the symbol   indicates a space character.

```
  -6789       /*  Decimal base */
+060          /*  Octal base */
0XFFFFFFFF    /*  Hexadecimal base */
```

The conversion process begins with the first character pointed to by `string`. If `strtoul` encounters a character that it does not recognize, it ends the conversion. On return, `strtoul` assigns to `end_ptr` the address of the character where the representation of the integer value stopped, or where the conversion ended. The character pointed to by `end_ptr` can be the null character (`\0`). Using the value returned in `end_ptr`, you can process the remainder of the string, if needed.

> **Note:** When you pass `NULL` instead of an `end_ptr` argument, `strtoul` assigns no address to `end_ptr`.

When the string is not in the correct format or the base is invalid, `strtoul` assigns the address of `string` to `end_ptr`.

If the return value is outside the range of an `unsigned long`, `strtoul` sets `errno` to `ERANGE`, which is a constant defined in the `math.h` header file.

Return Value

The `strtoul` function returns the converted string as a value of the type `unsigned long`. If the converted value is in the range of an `unsigned long` but the character representation was preceded by a minus sign, `strtoul` returns the two's complement of the value.

If the string is not in the correct format or the base is invalid, `strtoul` returns the value 0. When the converted value is outside the range of an `unsigned long`, `strtoul` returns `ULONG_MAX`. The `ULONG_MAX` constant is defined in the `limits.h` header file.

If `base` equals 0 or 16 and the first characters are `0x` or `0X` followed by a character that is not a valid hexadecimal digit, the return value equals 0. In this case, `end_ptr` points to the `x` or `X`.

Examples

The following program uses `strtoul` to convert four strings into values of the type `unsigned long`.

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

unsigned long long_i;
char *string;
char *end_ptr;
int base = 0;

main()
{
   string = "  -12345ABC";
   long_i = strtoul(string, &end_ptr, base);
  printf("long_i = %d\n", long_i);           /*  long_i = -12345
*/
   printf("Rest of string = %s\n", end_ptr);      /*  Rest of string
= ABC  */

   string = "-0XFFFFGHI";
   long_i = strtoul(string, &end_ptr, base);
  printf("long_i = %d\n", long_i);           /*  long_i = -65535
*/
   printf("Rest of string = %s\n", end_ptr);      /*  Rest of string
= GHI  */

   string = "060ABC";
   long_i = strtoul(string, &end_ptr, base);
  printf("long_i = %d\n", long_i);            /*  long_i = 48
*/
   printf("Rest of string = %s\n", end_ptr);      /*  Rest of string
= ABC  */

   string = " 123.456ABC";
   long_i = strtoul(string, &end_ptr, base);
  printf("d = %d\n", long_i);             /*  long_i = 123
*/
   printf("Rest of string = %s\n", end_ptr);   /*  Rest of string =
.456ABC */
}
```

Related Functions

`atoi`, `atol`, `strtod`, `strtol`

system

Purpose

Starts a subprocess that executes a specified program module.

Syntax

```
#include <stdlib.h>

int system(const char *string);
```

Explanation

The `system` function starts a subprocess that executes a specified program module. The `string` argument is a pointer to `char` that points to a string indicating the full or relative path name of the program module to execute.

No startup command macro is executed when the subprocess is created. The subprocess does not inherit the library paths of the program's process. In the subprocess, the string is passed to the command processor. No `ready` prompt is displayed.

> **Note:** The `system` function does **not** expand abbreviations. If desired, you can call the `s$abbrev` subroutine to expand the string that will be passed to `system` before calling the function. For information on the `s$abbrev` subroutine, see the *VOS C Subroutines Manual (R068).*

While the specified program module is executing, execution of the program that calls `system` is suspended.

If an error occurs, `system` sets the external variable `errno` to the error code number returned by the operating system.

Return Value

The `system` function returns the value of the `command_status` variable.

When the specified program module begins to execute, the operating system sets the value of the `command_status` variable to 0. If the program module executes normally without errors, the value of `command_status` remains 0. Otherwise, the value of `command_status` is set to one of the following:

- the value of the error code passed, by the specified program module, to the `s$error` or `s$stop_program` subroutine

- the value of the `status` argument when the specified program module calls the `exit` library function
- the value, if any, returned by the specified program module's `main` function.

If `string` is equal to `NULL`, the `system` function returns the value 0. In this case, no program module executes.

Examples

The following program uses `system` to start a subprocess and execute a program module whose name is entered at the terminal's keyboard.

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

char string[81];
int status;

main()
{
   puts("Enter the name of the program module.");
   scanf("%s", string);

   errno = 0;
   status = system(string);

   if (status == 0)
      puts("After execution of the program, status = 0.");
   else
      {
     puts("No program found, or the program returned a nonzero error
code.");
      printf("errno = %d\n", errno);
      }
}
```

Related Functions

`exit`

tan

Purpose

Computes the tangent of an angle, which is expressed in radians.

Syntax

```
#include <math.h>

double tan(double x);
```

Explanation

The `tan` function calculates the tangent of `x`. The `x` argument is an angle expressed in radians.

If the absolute value of `x` exceeds 32,767, `tan` sets `errno` to the error code number returned by the operating system.

Return Value

The `tan` function returns a tangent. When the absolute value of `x` exceeds 32,767, `tan` returns the value 0.

Examples

The following program uses `tan` to calculate the tangent of an angle, expressed in radians.

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

main()
{
   double x, tangent;

   printf("Enter an angle (in radians) for which ");
   printf("you want to find the tangent.\n");
   scanf("%le", &x);

   errno = 0;
   tangent = tan(x);

   if (errno != 0)
      printf("Error condition occurred:  errno = %d\n", errno);
   else
      printf("The tangent of %lE = %lE.\n", x, tangent);
}
```

If `1.57` were entered at the terminal's keyboard, the output would be as follows:

```
The tangent of 1.570000E+00 = 1.255766E+03.
```

Related Functions

`atan`, `atan2`, `cos`, `sin`

tanh

Purpose

Computes the hyperbolic tangent of a floating-point value.

Syntax

```
#include <math.h>

double tanh(double x);
```

Explanation

The `tanh` function calculates the hyperbolic tangent of `x`. The `x` argument is a value of the type `double`.

No domain or range errors are possible with the `tanh` function.

Return Value

The `tanh` function returns a hyperbolic tangent.

Examples

The following program uses `tanh` to calculate the hyperbolic tangent of a number entered at the terminal's keyboard.

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

main()
{
   double x, h_tangent;

   printf("Enter a number for which you want ");
   printf("to find the hyperbolic tangent.\n");
   scanf("%le", &x);

   h_tangent = tanh(x);

   printf("The hyperbolic tangent of %lE = %lE.\n", x, h_tangent);
}
```

If `1.0` were entered at the terminal's keyboard, the output would be as follows:

```
The hyperbolic tangent of 1.000000E+00 = 7.615942E-01.
```

Related Functions

`cosh`, `sinh`

time

Purpose

Gets the current calendar time: the number of seconds since January 1, 1980 GMT.

Syntax

```
#include <time.h>

time_t time(time_t *timer);
```

Explanation

The `time` function gets the current calendar time. In VOS C, the current calendar time is the number of seconds from 00:00:00 hour January 1, 1980 Greenwich Mean Time (GMT) to the current date and time. The `timer` argument is a pointer to a variable of the type `time_t`, which is a type defined in the `time.h` header file.

If the `timer` argument is not equal to `NULL`, the `time` function stores the current calendar time in the variable pointed to by `timer`.

To convert the value returned by `time` to UNIX time, add the value of the following expression to the function's return value:

```
(365 * 8 + 366 * 2) * 24 * 60 * 60
```

UNIX time is the number of seconds from 00:00:00 hour January 1, 1970 GMT to the current date and time.

Return Value

The `time` function returns the number of seconds from 00:00:00 hour January 1, 1980 GMT to the current date and time.

If the calendar time is not available, `time` returns the value -1, cast to the type `time_t`.

Examples

The following program uses `time` to obtain the current calendar time.

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

main()
{
   time_t timer, return_value, unix_time;

   return_value = time(&timer);

   printf("The current calendar time = %d seconds\n", return_value);

   unix_time = return_value + ( (365 * 8 + 366 * 2) * 24 * 60 * 60 );

   printf("UNIX time = %d seconds\n", unix_time);
}
```

Related Functions

`asctime`, `clock`, `ctime`, `difftime`, `gmtime`, `localtime`

tmpfile

Purpose

Creates and opens a temporary file.

Syntax

```
#include <stdio.h>

FILE *tmpfile(void);
```

Explanation

The `tmpfile` function creates a temporary file in the current directory of the program's process. The `tmpfile` function opens the file for update with `"wb+"` as the `mode` argument.

You access the temporary file by specifying the returned `file_ptr` with any of the standard I/O functions.

The system removes the temporary file when it is closed or when the program terminates.

Return Value

If `tmpfile` creates and opens the file successfully, the function returns a pointer to the `FILE` structure associated with the temporary file.

If `tmpfile` fails to create and open the file, the function returns the `NULL` pointer constant.

Examples

The following program uses `tmpfile` to create and open a temporary file.

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

FILE *file_ptr;
char data[50] = "abcdefghijkl";
char buffer[50];
int chars_written;
```

(Continued on next page)

```
main()
{
   if ( (file_ptr = tmpfile()) == NULL )
      {
      puts("Could not create/open temporary file.");
      exit(1);
      }

   chars_written = fwrite(data, sizeof(char), 50, file_ptr);

   printf("chars_written = %d\n", chars_written);

   rewind(file_ptr);

   fread(buffer, sizeof(char), chars_written, file_ptr);

   printf("Data written to the temporary file:  %s\n", buffer);
}
```

Related Functions

`tmpnam`

tmpnam

Purpose

Generates a unique string that is a valid file name.

Syntax

```
#include <stdio.h>

char *tmpnam(char *string);
```

Explanation

The `tmpnam` function generates a null-terminated string that is a valid VOS file name and that is not the same as the name of any existing file. The value of the `string` argument, a pointer to `char`, determines where `tmpnam` stores the name.

- If you want `tmpnam` to write the name to a buffer allocated by your program, the `string` argument locates an array of `char` containing at least `L_tmpnam` characters. The `L_tmpnam` constant is defined in the `stdio.h` header file.

- If you want `tmpnam` to write the name to a buffer that is allocated by `tmpnam` and has static (permanent) storage duration, the `string` argument equals the `NULL` pointer constant. In this case, subsequent calls to `tmpnam` modify the contents of this buffer.

The `tmpnam` function generates a unique string that has 17 characters, beginning with the underline character (_) and followed by 16 other characters. In VOS C, the `tmpnam` function generates a different name each time it is called even if it is called more than `TMP_MAX` times.

> **Note:** The name generated by `tmpnam` is guaranteed to be unique only when the current module's bootload time is never reset to a value that it has previously attained. Also, the name is not guaranteed to be unique relative to names generated by other processes, especially those running on other modules. However, even with these conditions, two identical names are unlikely to be generated.

Any file created using the name returned by `tmpnam` is temporary only in the sense that its name should not conflict with the names of existing files. You must use `remove` if you want to delete the file before the program ends.

Return Value

If the `string` argument is not equal to `NULL` (that is, `string` locates an array of `char`), `tmpnam` writes the name into the pointed-to array, and the function returns `string` as its return value.

If `string` equals `NULL`, `tmpnam` writes the name into a `static` array of `char` that is allocated by `tmpnam`. In this case, the function returns a pointer to the array it has allocated. Subsequent calls to `tmpnam` modify the contents of this array.

Examples

The following program uses `tmpnam` to generate a name.

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

char name[L_tmpnam];

main()
{
   char *ptr = NULL;

   if ( ( ptr = tmpnam(name)) == NULL )
      {
      puts("Error:  tmpnam did not generate a name.");
      exit(1);
      }

   printf("The name is %s\n", name);
}
```

The output from the preceding program might be as follows:

```
The name is _aaarkbrf3biaWhql
```

Related Functions

`tmpfile`

`toascii`

Purpose

Sets all but the rightmost seven bits of an integer variable to 0 so that the resulting value represents a valid character in the ASCII character set.

Syntax

```
#include <ctype.h>

int toascii(int c);
```

Explanation

The `toascii` function converts the integer value in `c` to an ASCII character by setting all but the seven rightmost (low-order) bits to the value 0. The seven rightmost bits are not affected by the conversion.

For example, consider the following integer variable, which stores the decimal value 4095 in binary format.

```
00000000 00000000 00001111 11111111
```

If you used this variable as input to the `toascii` function, the returned value would be 127 decimal. In binary format, all bits but the rightmost seven would be set to 0 as follows:

```
00000000 00000000 00000000 01111111
```

You might use the `toascii` function to convert the high-order bits to 0 if these bits contained formatting information for a character and this information was unwanted for a particular procedure, such as data communication.

Return Value

The `toascii` function returns a converted integer value representing a valid ASCII character. If the value in `c` already represents a valid ASCII character, the function returns the value unchanged.

Examples

The following program uses the `toascii` function to convert three integer values into valid characters in the ASCII character set. In each case, the function returns the value 65, the ASCII code for the character `A`.

```
#include <ctype.h>

int c, ascii_code;

main()
{
   c = 65;
   ascii_code = toascii(c);        /*  ascii_code = 65  */

   c = 193;
   ascii_code = toascii(c);        /*  ascii_code = 65  */

   c = 32833;
   ascii_code = toascii(c);        /*  ascii_code = 65  */
}
```

Related Functions

`isascii`, `tolower`, `_tolower`, `toupper`, `_toupper`

`tolower`, **`_tolower`**

Purpose

Converts an uppercase ASCII character to lowercase.

Syntax

```
#include <ctype.h>

int tolower(int c);       /*  The function  */

int _tolower(int c);      /*  The macro     */
```

Explanation

In VOS C, `tolower` is a function, and `_tolower` is a macro. Both the `tolower` function and the `_tolower` macro convert an uppercase ASCII character to lowercase. The ANSI C Standard specifies the behavior of the `tolower` function. The `_tolower` macro is a common extension to many C compilers.

The `tolower` function is slower than the macro. The `tolower` function tests whether the `c` argument is an ASCII uppercase character.

The `_tolower` macro is faster than the function. The `_tolower` macro does **not** test whether the `c` argument is an ASCII uppercase character. In addition, the `_tolower` macro can yield unexpected results if it is passed an expression with side effects.

Return Value

Both the `tolower` function and the `_tolower` macro return a lowercase ASCII character if the `c` argument is an uppercase ASCII character. If the `c` argument is not an uppercase ASCII character, the return value is as follows:

- The `tolower` function returns the character unchanged.
- The `_tolower` macro returns an unrelated character.

Examples

The following program contains four examples. The first two examples use the `tolower` function to convert two ASCII characters, `A` and `%`, to lowercase. The last two examples use the `_tolower` macro to convert the same ASCII characters to lowercase. The `%` character is not a valid ASCII uppercase character.

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

int ch;

main()
{
   /*  The first two examples use the tolower FUNCTION  */

   ch = 'A';
   printf("ch is %c\n", tolower(ch) );    /*  ch is a   */

   ch = '%';
   printf("ch is %c\n", tolower(ch) );    /*  ch is %   */

   /*  The next two examples use the _tolower MACRO     */

   ch = 'A';
   printf("ch is %c\n", _tolower(ch) );  /*  ch is a    */

   ch = '%';
   printf( "ch is %c\n", _tolower(ch) ); /*  ch is E (unrelated
character)  */
}
```

Related Functions

`isascii`, `toascii`, `toupper`, `_toupper`

`toupper`, `_toupper`

Purpose

Converts a lowercase ASCII character to uppercase.

Syntax

```
#include <ctype.h>

int toupper(int c);       /*  The function  */

int _toupper(int c);      /*  The macro     */
```

Explanation

In VOS C, `toupper` is a function, and `_toupper` is a macro. Both the `toupper` function and the `_toupper` macro convert a lowercase ASCII character to uppercase. The ANSI C Standard specifies the behavior of the `toupper` function. The `_toupper` macro is a common extension to many C compilers.

The `toupper` function is slower than the macro. The `toupper` function tests whether the `c` argument is a valid ASCII lowercase character.

The `_toupper` macro is faster than the function. The `_toupper` macro does **not** test whether the `c` argument is a valid ASCII lowercase character. In addition, the `_toupper` macro can yield unexpected results if it is passed an expression with side effects.

Return Value

Both the `toupper` function and the `_toupper` macro return an uppercase ASCII character if the `c` argument is a lowercase ASCII character. If the `c` argument is not a lowercase ASCII character, the return value is as follows:

- The `toupper` function returns the character unchanged.
- The `_toupper` macro returns an unrelated character.

Examples

The following program contains four examples. The first two examples use the `toupper` function to convert two ASCII characters, `a` and `{`, to uppercase. The last two examples use the `_toupper` macro to convert the same ASCII characters to uppercase. The `{` character is not a valid ASCII lowercase character.

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

int ch;

main()
{
   /*  The first two examples use the toupper FUNCTION  */

   ch = 'a';
   printf("ch is %c\n", toupper(ch) );   /*  ch is A    */

   ch = '{';
   printf("ch is %c\n", toupper(ch) );   /*  ch is {    */

   /*  The next two examples use the _toupper MACRO     */

   ch = 'a';
   printf("ch is %c\n", _toupper(ch) );  /*  ch is A    */

   ch = '{';
   printf( "ch is %c\n", _toupper(ch) ); /*  ch is [  (unrelated
character)  */
}
```

Related Functions

`isascii`, `toascii`, `tolower`, `_tolower`

`ungetc`

Purpose

Pushes a specified character back onto the input stream associated with a file.

Syntax

```
#include <stdio.h>

int ungetc(int c, FILE *file_ptr);
```

Explanation

The `ungetc` function pushes the character `c` back onto the input stream, or buffer, pointed to by `file_ptr`. Subsequent read operations on that stream return any pushed-back characters in the reverse order of their being pushed onto the stream. The `c` argument is converted to an `unsigned char` before it is pushed back on the stream. The `file_ptr` argument is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`.

The `ungetc` function affects only the I/O buffer specified by `file_ptr`. The file associated with `file_ptr` is unchanged.

The file-positioning functions, `fseek` and `rewind`, discard any pushed-back characters for the stream. After the `ungetc` call, a read operation does **not** read any pushed-back characters if an intervening call to a file-positioning function specifying that file has executed successfully.

Some restrictions apply to the use of `ungetc`.

- If `c` equals `EOF`, the push-back operation fails, and `ungetc` returns `EOF`. In this case, `ungetc` sets the external variables `errno` and `os_errno` to the error code `e$end_of_file` (`1025`).
- The results of multiple calls to `ungetc` are unpredictable. Only one character of pushback is guaranteed. If you call `ungetc` multiple times without an intervening read operation or an intervening call to a file-positioning function, the push-back operation can fail.

After the program reads or discards all pushed-back characters, the value of the file's file-position indicator is the same as it was before the characters were pushed back. In both text and binary mode, the file-position indicator is decremented by each successful call to `ungetc`. However, if the file-position indicator was at 0 before the call, its value after the call is unpredictable. A successful call to `ungetc` clears the file's end-of-file indicator if it was set.

If a call to `ungetc` fails, the function sets `errno` and `os_errno` to the appropriate error code: either `e$end_of_file` (`1025`) or `e$invalid_io_operation` (`1040`).

Return Value

The `ungetc` function returns the pushed-back character if it is successful. If the push-back operation fails, `ungetc` returns `EOF`.

Examples

The following program uses the `ungetc` function to push back, onto the input stream pointed to by `stdin`, the first nondigit that the user enters at the terminal's keyboard. It then displays the digits entered and the remainder of the line of input.

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

char digit_array[81];
char remainder_array[81];
int c, i;

main()
{
   puts("On one line, enter some digits and then some other
characters.");

  while ( (c = getchar()) != EOF && isdigit(c) )   /*  Store characters
*/
     {                                          /*  until a nondigit    */
     digit_array[i] = (char)c;                     /*  is encountered
*/
      i++;
      }

   digit_array[i] = '\0';
                                              /*  Push back c as it    */
  if ( ungetc(c, stdin) == EOF )                  /*  is not a digit
*/
      puts("The push-back operation failed.");
   else
      {
      gets(remainder_array);
      printf("The digits are %s.\n", digit_array);
      printf("The remainder of the line is %s.\n", remainder_array);
      }
}
```

If the line `12345abcdefghi` were entered at the terminal's keyboard, the output from the preceding program would be as follows:

```
The digits are 12345.
The remainder of the line is abcdefghi.
```

Related Functions

`fgetc`, `fputc`, `getc`, `getchar`, `putc`, `putchar`

va_arg

Purpose

Accesses the next argument from the varying part of a parameter list.

Syntax

```
#include <stdarg.h>

type va_arg(va_list arg_ptr, type);
```

Explanation

The `va_arg` macro accesses the next argument, in sequence, from the varying part of a parameter list. The `va_arg` macro expands to an expression that yields the type and value of the next argument in the function call.

The `arg_ptr` argument has the type `va_list`, which is a type defined in the `stdarg.h` header file. Before you call `va_arg`, you must call `va_start` so that `arg_ptr` is initialized to point to the first argument in the varying part of the list.

The *`type`* argument is a type name specified in such a way that the type of a pointer to an object of the specified type can be constructed by postfixing an `*` to *`type`*, as in *`type`* `*`. If the *`type`* specified, after application of the default argument promotions, is not compatible with the type of the actual argument that will be accessed, the result of the `va_arg` call is unpredictable.

The `va_start`, `va_arg`, and `va_end` macros are used together to access the arguments from the varying part of a parameter list. For more information on how to use these macros, see the “Variable Arguments” section earlier in this chapter.

Return Value

After `va_start` is invoked to initialize `arg_ptr`, the first invocation of the `va_arg` macro returns the value of the argument that appears after the argument specified by `parm_n`. The type of the return value is the same as was specified in the *`type`* argument. For information on `parm_n`, see the description of the `va_start` macro.

Successive invocations of `va_arg` return, in sequence, the values of the remaining arguments. When there is no next actual argument, the return value of the `va_arg` call is unpredictable.

Examples

The following program uses the `va_start`, `va_arg`, and `va_end` macros to access arguments in the varying part of a parameter list. In this program, `number_args` is the nonvarying argument, which is denoted as `parm_n` in the function prototype for `va_start`.

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

int get_sum(int number_args, ...);

int number_of_args, total;
int num_1 = 100, num_2 = 200, num_3 = 300, num_4 = 400;

main()
{
   number_of_args = 4;

   total = get_sum(number_of_args, num_1, num_2, num_3, num_4);

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

int get_sum(int number_args, ...)
{
   va_list arg_ptr;
   int sum = 0;

   va_start(arg_ptr, number_args);     /*  va_start initializes
arg_ptr      */

   while (number_args > 0)
      {
    sum += va_arg(arg_ptr, int);    /*  va_arg accesses each argument
in  */
      --number_args;                   /*  the varying part of the
list      */
      }

   va_end(arg_ptr);                    /*  va_end makes arg_ptr
unuseable    */

   return(sum);
}
```

Related Functions

`va_end`, `va_start`

va_end

Purpose

Modifies the argument pointer, `arg_ptr`, so that it is no longer useable.

Syntax

```
#include <stdarg.h>

void va_end(va_list arg_ptr);
```

Explanation

After `va_start` has been called to initialize `arg_ptr`, the `va_end` macro modifies the pointer so that it is no longer useable without an intervening call to `va_start`. The `arg_ptr` argument has the type `va_list`, which is a type defined in the `stdarg.h` header file.

The `va_start`, `va_arg`, and `va_end` macros are used together to access the arguments from the varying part of a parameter list. For more information on how to use these macros, see the "Variable Arguments" section at the beginning of this chapter.

Return Value

The `va_end` macro returns no value.

Examples

See the description of the `va_arg` macro for an example of how to use the `va_end` macro.

Related Functions

`va_arg`, `va_start`

va_start

Purpose

Initializes the argument pointer, `arg_ptr`, so that it can be used by `va_arg` and `va_end`.

Syntax

```
#include <stdarg.h>

void va_start(va_list arg_ptr, type parm_n);
```

Explanation

The `va_start` macro initializes `arg_ptr` to point to the first argument in the varying part of a parameter list. Subsequent calls to the `va_arg` and `va_end` macros allow you to access the parameters in the varying part of a parameter list. The `arg_ptr` argument has the type `va_list`, which is a type defined in the `stdarg.h` header file. The `parm_n` argument is the rightmost nonvarying parameter in the parameter list (the parameter immediately preceding the `, ...`).

The varying part of a parameter list is denoted by an ellipsis (`...`) and can contain a varying number of arguments or arguments of varying types. To access arguments in the varying part of the list, the parameter list must contain at least one nonvarying argument. This argument is designated as `parm_n` in the prototype. The `parm_n` argument can be any type except a function or array type, or a type that is not compatible with the type that results after the default argument promotions are applied to `parm_n`. If `parm_n` is declared with the `register` storage class, the behavior is unpredictable.

The `va_start`, `va_arg`, and `va_end` macros are used together to access the arguments from the varying part of a parameter list. For more information on how to use these macros, see the "Variable Arguments" section at the beginning of this chapter.

Return Value

The `va_start` macro returns no value.

Examples

See the description of the `va_arg` macro for an example of how to use `va_start`.

Related Functions

`va_arg`, `va_end`

vfprintf

Purpose

Writes a formatted string of characters to a file. Unlike `fprintf`, this function accepts a pointer to a list of arguments, not the arguments themselves.

Syntax

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

int vfprintf(FILE *file_ptr, const char *format, va_list arg_ptr);
```

Explanation

The `vfprintf` function writes output to the file pointed to by `file_ptr` much like `fprintf` does. The `vfprintf` function accepts as an argument a pointer, `arg_ptr`, to a list of arguments. In contrast, `fprintf` expects the arguments themselves. The `arg_ptr` argument has the type `va_list`, which is a type defined in the `stdarg.h` header file.

The `file_ptr` argument to `vfprintf` is a pointer returned, for example, by a previous call to `fdopen`, `fopen`, or `freopen`. The output written is under the control of the string pointed to by `format`. The string pointed to by `format` can consist of zero or more directives. The `vfprintf` function executes each directive in turn. It returns when it encounters the end of the format string. The directives consist of the following:

- characters, including escape sequences
- conversion specifications.

The `vfprintf` function writes ordinary characters (not `%`) unchanged to the file. The format string can include conversion specifications. Each *conversion specification* begins with a `%` and indicates how the value of a subsequent argument will be converted and written. The number and types of the arguments in a `vfprintf` call depend on the conversion specifications given in the format string. Each conversion specification results in fetching zero or more arguments.

Each argument in the varying part of the argument list (denoted by `...` in the function definition) must be an expression yielding an object type appropriate for the corresponding conversion specification. If there are insufficient arguments for the format, the conversions have unpredictable results. If the format is exhausted while arguments remain, the extra arguments are evaluated but otherwise ignored.

The format for a conversion specification is as follows:

```
%[flags][width] .[precision][modifier] conversion_specifier
```

For information on each element of a conversion specification, see the `printf` function.

The procedure used to call the `vfprintf` function is as follows:

1. Before you invoke `vfprintf`, call the `va_start` macro to initialize `arg_ptr` so that it points to the base of the varying part of the parameter list.
2. When you invoke `vfprintf`, specify `arg_ptr` as the third argument.
3. After you invoke `vfprintf` (thereby accessing the arguments in the list), call the `va_end` macro to modify `arg_ptr` so that it is no longer useable.

Return Value

If the formatted output specified is successfully written, `vfprintf` returns the number of characters written to the file.

If an output error occurs, `vfprintf` returns a negative value.

Examples

The following program fragment uses `vfprintf` to convert an `int` and a `double` value into ASCII character representations of the values and to write the formatted output to a specified file.

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

int i_num;
double d_num;
short error_code;
char *fmt;

void file_print_func(char *format, ...);

main()
{
   i_num = 111;
   d_num = 222.222;
      .
      .
      .
   if (error_code != 0)
      fmt = NULL;
   else
      fmt = "i_num = %d\nd_num = %f\n";

   file_print_func(fmt, i_num, d_num);
}
```

(Continued on next page)

```
void file_print_func(char *format, ...)
{
   va_list arg_ptr;
   FILE *file_ptr;

   if ( (file_ptr = fopen("test_file", "w")) == NULL )
      {
      printf("Could not open test_file\n");
      exit(1);
      }

   if (format != NULL)
      {
      va_start(arg_ptr, format);

      vfprintf(file_ptr, format, arg_ptr);

      va_end(arg_ptr);
      }
}
```

The output written to the file is as follows:

```
i_num = 111
d_num = 222.222000
```

Related Functions

`fprintf`, `printf`, `sprintf`, `vprintf`, `vsprintf`

vprintf

Purpose

Writes a formatted string of characters to the standard output file, `stdout`. Unlike `printf`, this function accepts a pointer to a list of arguments, not the arguments themselves.

Syntax

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

int vprintf(const char *format, va_list arg_ptr);
```

Explanation

The `vprintf` function writes output to the preopened file pointed to by `stdout` much like `printf` does. However, the `vprintf` function accepts as an argument a pointer, `arg_ptr`, to a list of arguments. In contrast, `printf` expects the arguments themselves. The `arg_ptr` argument has the type `va_list`, which is a type defined in the `stdarg.h` header file.

The output written is under the control of the string pointed to by `format`. The string pointed to by `format` can consist of zero or more directives. The `vprintf` function executes each directive in turn. It returns when it encounters the end of the format string. The directives consist of the following:

- characters, including escape sequences
- conversion specifications.

The `vprintf` function writes ordinary characters (not `%`) unchanged to the file. The format string can include conversion specifications. Each *conversion specification* begins with a `%` and indicates how the value of a subsequent argument will be converted and written. The number and types of the arguments in a `vprintf` call depend on the conversion specifications given in the format string. Each conversion specification results in fetching zero or more arguments.

Each argument in the variable part of the argument list (denoted by `...` in the function definition) must be an expression yielding an object type appropriate for the corresponding conversion specification. If there are insufficient arguments for the format, the conversions have unpredictable results. If the format is exhausted while arguments remain, the extra arguments are evaluated but otherwise ignored.

The format for a conversion specification is as follows:

```
%[flags][width] .[precision][modifier] conversion_specifier
```

For information on each element of a conversion specification, see the `printf` function.

The procedure used to call the `vprintf` function is as follows:

1. Before you invoke `vprintf`, call the `va_start` macro to initialize `arg_ptr` so that it points to the base of the varying part of the parameter list.
2. When you invoke `vprintf`, specify `arg_ptr` as the third argument.
3. After you invoke `vprintf` (thereby accessing the arguments in the list), call the `va_end` macro to modify `arg_ptr` so that it is no longer useable.

Return Value

If the formatted output specified is successfully written, `vprintf` returns the number of characters written to the file.

If an output error occurs, `vprintf` returns a negative value.

Examples

The following program uses `vprintf` to display an error message on the terminal's screen.

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

int error_code;
char function_name[32];

void display_error(char *format, ...);

main()
{
   double sine;

   errno = 0;
   sine = sin(99999.0);          /*  Creates an error  */

   if (errno != 0)
      {
      strcpy(function_name, "main");
      error_code = errno;
      display_error("Error %d occurred in %s\n", error_code,
function_name);
      exit(1);
      }
}
```

(Continued on next page)

```
void display_error(char *format, ...)
{
   va_list arg_ptr;

   va_start(arg_ptr, format);

   vprintf(format, arg_ptr);

   va_end(arg_ptr);
}
```

The output from the preceding program is as follows:

```
Error 1755 occurred in main
```

Related Functions

`fprintf`, `printf`, `sprintf`, `vfprintf`, `vsprintf`

vsprintf

Purpose

Writes a formatted string of characters into an array. Unlike `sprintf`, this function accepts a pointer to a list of arguments, not the arguments themselves.

Syntax

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

int vsprintf(char *s, const char *format, va_list arg_ptr);
```

Explanation

The `vsprintf` function writes output to the array of `char` whose address is specified by `s`. The `vsprintf` function accepts as an argument a pointer, `arg_ptr`, to a list of arguments. In contrast, `sprintf` expects the arguments themselves. The `arg_ptr` argument has the type `va_list`, which is a type defined in the `stdarg.h` header file.

After the characters are written, `vsprintf` appends a null character so that the specified array contains a C string. One possible use of `vsprintf` is to convert a numeric value stored in a variable to an ASCII character representation of the value. For example, the value 123 stored in an `int` variable can be written to an array as `"123"`, three characters plus the appended null character.

The output written is under the control of the string pointed to by `format`. The string pointed to by `format` can consist of zero or more directives. The `vsprintf` function executes each directive in turn. It returns when it encounters the end of the format string. The directives consist of the following:

- characters, including escape sequences
- conversion specifications.

The `vsprintf` function writes ordinary characters (not `%`) unchanged to the file. The format string can include conversion specifications. Each *conversion specification* begins with a `%` and indicates how the value of a subsequent argument will be converted and written. The number and types of the arguments in a `vsprintf` call depend on the conversion specifications given in the format string. Each conversion specification results in fetching zero or more arguments.

Each argument in the variable part of the argument list (denoted by `...` in the function definition) must be an expression yielding an object type appropriate for the corresponding conversion specification. If there are insufficient arguments for the format, the conversions have unpredictable results. If the format is exhausted while arguments remain, the extra arguments are evaluated but otherwise ignored.

The format for a conversion specification is as follows:

`%`[`flags`][`width`] `.`[`precision`][`modifier`] *`conversion_specifier`*

For information on each element of a conversion specification, see the `printf` function.

The procedure used to call the `vsprintf` function is as follows:

1. Before you invoke `vsprintf`, call the `va_start` macro to initialize `arg_ptr` so that it points to the base of the varying part of the parameter list.
2. When you invoke `vsprintf`, specify `arg_ptr` as the third argument.
3. After you invoke `vsprintf` (thereby accessing the arguments in the list), call the `va_end` macro to modify `arg_ptr` so that it is no longer useable.

Return Value

If the formatted output specified is successfully written, `vsprintf` returns the number of characters, except for the terminating null character, written to the array.

Examples

The following program uses `vsprintf` to convert three `int` values into ASCII character representations of the values, and to write the string to a specified array.

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

char *convert_func(char *format, ...);

int num_1, num_2, num_3;
char *s_ptr;

main()
{
   num_1 = 111;
   num_2 = 222;
   num_3 = 333;

   s_ptr = convert_func("totals are:  %d, %d, %d\n", num_1, num_2,
num_3);

   printf("%s\n", s_ptr);
}
```

(Continued on next page)

```
char *convert_func(char *format, ...)
{
   static char string[100];

   va_list arg_ptr;

   va_start(arg_ptr, format);

   vsprintf(string, format, arg_ptr);

   va_end(arg_ptr);

   return(string);
}
```

The output written to the array `string` is as follows:

```
totals are:  111, 222, 333
```

Related Functions

`fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`

write

Purpose

Writes a specified number of bytes of data to a file associated with a given file descriptor.

Syntax

```
#include <c_utilities.h>

int write(int file_des, char *buffer, unsigned int nbytes);
```

Explanation

The `write` function writes `nbytes` of data from the memory area pointed to by `buffer` into the file specified by `file_des`. The `file_des` argument is a file descriptor returned by a previous call to `creat`, `fileno`, or `open`.

The data is written starting at the file's current position. After `write` writes data to the file, it increments the file-position indicator, associated with the file, by the number of bytes actually written.

If an error occurs, `write` sets the external variables `errno` and `os_errno` to the error code number returned by the operating system. The following table lists some of the error code values that are possible.

Error Code Name	**Number**
`e$abort_output`	`1279`
`e$caller_must_wait`	`1277`
`e$invalid_file_pointer`	`3929`
`e$invalid_io_operation`	`1040`
`e$invalid_record_size`	`1053`
`e$port_not_attached`	`1021`
`e$record_format_error`	`4265`
`e$record_in_use`	`2408`
`e$record_too_long`	`1041`
`e$record_too_short`	`1118`

Return Value

The `write` function returns the number of bytes actually written to the file.

If an error occurs, `write` returns the value -1.

Examples

The following program uses `write` to write a specified number of bytes of data from an array, whose address is `buffer`, to the file associated with the file descriptor `file_des`. The number of bytes written corresponds to the number of bytes read in a previous call to `read`.

```
#include <c_utilities.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int file_des, mode;
int chars_read, chars_written;
char buffer[80];

main()
{
   if ( (file_des = open("out_file", O_WRONLY | O_APPEND, mode)) == -1 )
      {
      printf("Error opening out_file.\n");
      exit(1);
      }

   puts("Enter a line to write to out_file.");
   chars_read = read(0, buffer, 80);     /*  file descriptor 0 = stdin  */

   errno = 0;
   chars_written = write(file_des, buffer, chars_read);

   if (chars_written == -1)
      printf("Write error occurred:  errno = %d\n", errno);
   else
      printf("%d bytes have been written to out_file\n", chars_written);

   close(file_des);
}
```

Related Functions

`close`, `creat`, `fwrite`, `lseek`, `open`, `read`

Chapter 12: Built-in Functions

The VOS C built-in functions are used for two general categories of tasks:

- handling National Language Support (NLS) characters
- generating `char_varying` values with `$substr`.

The VOS C built-in functions are **not** specified as part of the ANSI C library. They are VOS C extensions and, therefore, are not portable.

Each of the built-in functions begins with the dollar sign character (`$`). The `$` character is normally reserved for VOS C extensions, such as the built-in functions. If the compiler encounters a name that is preceded by the `$` character and the name is that of a built-in function, the compiler performs a special action. If the name that is preceded by the `$` character is not the name of a built-in function, the compiler removes the `$` character and treats the name as it treats any other identifier. NamesIdentifiers

You do not have to include any header file into the source module to use the built-in functions. Each built-in function's identifier is visible throughout the source module. The compiler diagnoses attempts to use a built-in function's name, such as `$substr`, in the declaration of a programmer-defined item **only if** the name is preceded by the `$` character. If you use a built-in function's name without preceding the name with the `$` character (for example, `substr`), the function's name is hidden by the declaration of a programmer-defined item that has the same name as the built-in.

Table 12-1 lists the VOS C built-in functions.

Table 12-1. VOS C Built-in Functions *(Page 1 of 2)*

Function Name	**Description**
`$charcode`	Gets the character set number or an encoded value associated with a specified character.
`$charwidth`	Gets the number of bytes occupied by a character in a specified character set.
`$iclen`	Computes the length of an NLS string.
`$lockingcharcode`	Gets the character set number for a specified locking-shift character.
`$lockingshiftintroducer`	Gets the value of the locking-shift introducer character.

Table 12-1. VOS C Built-in Functions *(Page 2 of 2)*

Function Name	Description
`$lockingshiftselector`	Gets the value of the locking-shift character for a specified character set.
`$shift`	Converts an NLS string into an equivalent canonical string using a specified default character set.
`$singleshiftchar`	Gets the single-shift character for a specified character set.
`$substr`	Converts a specified number of bytes from a given source into a `char_varying` value.
`$unshift`	By removing all single-shift characters for a specified character set, converts an unambiguous canonical string or a common string into an ambiguous NLS string with a default character set.

See the *National Language Support User's Guide (R212)* for information on National Language Support and additional examples of the NLS-related built-in functions.

Format for Built-in Functions. This chapter uses the same format to describe the built-in functions as is used to describe the library functions in Chapter 11. See the "VOS C Library Function Reference" section in Chapter 11 for information on this format.

Although none of the built-in functions has a function prototype that is declared before the function is used, when the compiler recognizes a built-in function's name, it checks the correctness of the number and data types of the arguments that are used in the function call. The Syntax in the description of each built-in function lists information on the number and data types of the function's arguments.

Character Set ID Defined Constants. With many of the NLS built-in functions, you specify a character set ID as an argument in the function call. You can use a defined constant, such as `LATIN_1_CHAR_SET`, for the character set ID argument if you include the `char_sets.incl.c` header file into the source module. This header file is located in the `>system>include_library` directory.

Table 12-2 lists the defined constant used for each character set ID (character set number) and the `short int` value indicated by each constant.

Table 12-2. Defined Constants for Character Set Numbers *(Page 1 of 2)*

Defined Constant	Value
`ASCII_CHAR_SET`	0
`LATIN_1_CHAR_SET`	1
`KANJI_CHAR_SET`	2
`KATAKANA_CHAR_SET`	3

Table 12-2. Defined Constants for Character Set Numbers *(Page 2 of 2)*

Defined Constant	Value
`HANGUL_CHAR_SET`	4
`SIMPLIFIED_CHINESE_CHAR_SET`	5
`CHINESE1_CHAR_SET`	6
`CHINESE2_CHAR_SET`	7
`USER_DOUBLE_BYTE_CHAR_SET`	8

Character Codes. To use the terminal's keyboard to enter the hexadecimal character code for a right graphic character or a shift character, press the ESC key and then enter a grave accent (`) followed by the hexadecimal code that represents the character.

$charcode

Purpose

Gets the character set number or an encoded value associated with a specified character.

Syntax

```
short $charcode(char c);
```

Explanation

The `$charcode` function gets the character set number or an encoded value associated with the character `c`.

Return Value

If the `c` argument is a single-shift character, the `$charcode` function returns the character set number associated with the character.

If the `c` argument is not a single-shift character, the `$charcode` function returns a value signifying the NLS character category to which `c` belongs.

The following table lists the possible values for the `c` argument and the corresponding values that `$charcode` returns. The table also contains the name and meaning associated with each `c` value.

(Page 1 of 2)

Value of `c` (in hex.)	**Return Value**	**Name: Meaning**
00 to 0E	-1	`C0`: Left (ASCII) control character
0F	0	`SS0`: Single-shift character for character set 0
10 to 1F	-1	`C0`: Left (ASCII) control character
20 to 7E	-2	`G0`: Left (ASCII) graphic character
7F	-6	`DEL`: Delete character
80	1	`SS1`: Single-shift character for character set 1
81	4	`SS4`: Single-shift character for character set 4
82	5	`SS5`: Single-shift character for character set 5
83	6	`SS6`: Single-shift character for character set 6

(Page 2 of 2)

Value of `c` (in hex.)	Return Value	Name: Meaning
84	7	`SS7`: Single-shift character for character set 7
85	8	`SS8`: Single-shift character for character set 8
86	9	`SS9`: Single-shift character for character set 9
87	10	`SS10`: Single-shift character for character set 10
88	11	`SS11`: Single-shift character for character set 11
89	12	`SS12`: Single-shift character for character set 12
8A	13	`SS13`: Single-shift character for character set 13
8B	14	`SS14`: Single-shift character for character set 14
8C	15	`SS15`: Single-shift character for character set 15
8d	-3	`C1`: Right control character
8E	2	`SS2`: Single-shift character for character set 2
8F	3	`SS3`: Single-shift character for character set 3
90	-5	`LSI`: Locking-shift introducer character
91 to 9F	-3	`C1`: Right control character (except LSI, SS*n*)
A0 to FF	-4	`G1`: Right graphic character

Examples

The following program contains two examples of how `$charcode` is used to get the character set number or an encoded value associated with a character.

```
#pragma default_char_set (latin_1)

#include <stdio.h>
#include <char_sets.incl.c>

char c;
short ret_value;

main()
{
   c = $ss(LATIN_1_CHAR_SET);
   ret_value = $charcode(c);
   printf("The character set associated with %#x is = %hd\n", c,
ret_value);
```

(Continued on next page)

```
      c = '\accent133o';
      ret_value = $charcode(c);
      printf("The encoded value associated with %c is = %hd\n", c,
   ret_value);
   }
```

The output from the preceding program is as follows:

```
   The character set associated with 0x80 is = 1
   The encoded value associated with \accent133o is = -4
```

Related Functions

`$charwidth`, `$lockingcharcode`

$charwidth

Purpose

Gets the number of bytes occupied by a character in a specified character set.

Syntax

```
short $charwidth(short char_set_id)
```

Explanation

The `$charwidth` function gets the number of bytes occupied by a character in the character set specified by `char_set_id`. The `char_set_id` argument is a `short` value specifying the number of a character set.

If you include the `char_sets.incl.c` header file into the source module, you can use a defined constant, such as `LATIN_1_CHAR_SET`, for a character set number when you specify the `char_set_id` argument.

Return Value

The `$charwidth` function returns the number of bytes occupied by a character in the character set specified by `char_set_id`. The following table lists the value that `$charwidth` returns for each character set.

Character Set Number	**Character Set Defined Constant**	**Return Value**
0	`ASCII_CHAR_SET`	1
1	`LATIN_1_CHAR_SET`	1
2	`KANJI_CHAR_SET`	2
3	`KATAKANA_CHAR_SET`	1
4	`HANGUL_CHAR_SET`	2
5	`SIMPLIFIED_CHINESE_CHAR_SET`	2
6	`CHINESE1_CHAR_SET`	2
7	`CHINESE2_CHAR_SET`	2
8	`USER_DOUBLE_BYTE_CHAR_SET`	2

Examples

The following program uses `$charwidth` to get the number of bytes occupied by a character in the Latin alphabet No. 1 character set.

```
#include <char_sets.incl.c>
#include <stdio.h>

short num_bytes;
short char_set_id = LATIN_1_CHAR_SET;

main()
{
   num_bytes = $charwidth(char_set_id);

   printf("For Latin alphabet No. 1, each character occupies %d
byte(s)\n",
          num_bytes);
}
```

The output from the preceding program is as follows:

```
For Latin alphabet No. 1, each character occupies 1 byte(s)
```

Related Functions

`$charcode`, `$lockingcharcode`

$iclen

Purpose

Computes the length of an NLS string.

Syntax

```
/*  Used with a char_varying argument  */

short $iclen(char_varying(n) cv_string[, short number])

/*  Used with a char * argument        */

short $iclen(char *string, short length[, short number])
```

Explanation

The `$iclen` function computes the length, in bytes, of an NLS string. The subject string can be a canonical string or a common string. If the string contains NLS characters, the length includes any 1-byte characters, 2-byte characters, and shift characters that comprise the logical NLS characters.

If the subject string is a `char_varying` string, the `$iclen` function can take two arguments.

- `cv_string` is a `char_varying` expression specifying the string.
- `number`, an optional argument, specifies the number of characters in an initial substring for which `$iclen` will calculate the length.

If the subject string is an array of `char`, the `$iclen` function can take three arguments.

- `string` is a pointer to `char` locating the string to convert.
- `length` specifies the length, in bytes, of the array.
- `number`, an optional argument, specifies the number of characters in an initial substring for which `$iclen` will calculate the length.

Whether the subject string is a `char_varying` string or an array of `char`, if you do not specify the `number` argument, `$iclen` calculates the length of the entire string.

Return Value

If the `number` argument is not specified, the `$iclen` function returns the length, in bytes, of all characters in the specified string.

If the `number` argument is specified, `$iclen` returns the length, in bytes, of the characters in the initial substring indicated by `number`.

When the `number` argument is specified and the subject string does not contain at least `number` characters, `$iclen` returns the value 0.

Examples

The following program contains three examples of how `$iclen` is used to compute the length of an NLS string.

```
/*  The pragma causes the compiler to store all strings as canonical
*/
/*  strings:  with a single shift before each right-graphic character,
*/
/*  even those from Latin alphabet No. 1                              */

#pragma default_char_set (none)

#include <stdio.h>
#include <char_sets.incl.c>

char string[10] = "c\accent133ot\accent135e";
char_varying(20) cv_string_1 = "c\accent133ot\accent135e";
char_varying(20) cv_string_2;

short length;

main()
{
    length = $iclen(string, 10, 4);
    printf("The length of %s in string = %d\n", string, length);

    length = $iclen(cv_string_1);
    printf("The length of %v in cv_string_1 = %d\n", &cv_string_1,
length);

/*  $unshift removes single-shift characters from all right graphic
*/
/*  characters from Latin alphabet No. 1                              */

    cv_string_2 = $unshift(cv_string_1, LATIN_1_CHAR_SET);

    length = $iclen(cv_string_2);
   printf("\nAfter $unshift, the length of %v in cv_string_2 = %d\n",
            &cv_string_2, length);
}
```

The output from the preceding program is as follows:

```
The length of c\accent133ot\accent135e in string = 6

The length of c\accent133ot\accent135e in cv_string_1 = 6

After $unshift, the length of c\accent133ot\accent135e in
cv_string_2 = 4
```

Related Functions

`$shift`, `$unshift`

`$lockingcharcode`

Purpose

Gets the character set number for a specified locking-shift character.

Syntax

```
short $lockingcharcode(char c);
```

Explanation

The `$lockingcharcode` function gets the character set number associated with the locking-shift character specified by `c`.

Return Value

If the `c` argument is a valid locking-shift character, the `$lockingcharcode` function returns the character set number associated with the character.

If the `c` argument is not a valid locking-shift character, the `$lockingcharcode` function returns the value -1.

The following table lists the possible values for the `c` argument and the corresponding values that `$lockingcharcode` returns. The table also contains the name and meaning associated with each `c` value.

(Page 1 of 2)

Value of `c` (in hex.)	**Return Value**	**Name: Meaning**
00 to 9F	-1	Invalid value for the `c` argument
A0	1	`LS1`: Locking-shift character for character set 1
A1	4	`LS4`: Locking-shift character for character set 4
A2	5	`LS5`: Locking-shift character for character set 5
A3	6	`LS6`: Locking-shift character for character set 6
A4	7	`LS7`: Locking-shift character for character set 7
A5	8	`LS8`: Locking-shift character for character set 8
A6	9	`LS9`: Locking-shift character for character set 9
A7	10	`LS10`: Locking-shift character for character set 10

(Page 2 of 2)

Value of `c` (in hex.)	**Return Value**	**Name: Meaning**
A8	11	`LS11`: Locking-shift character for character set 11
A9	12	`LS12`: Locking-shift character for character set 12
AA	13	`LS13`: Locking-shift character for character set 13
AB	14	`LS14`: Locking-shift character for character set 14
AC	15	`LS15`: Locking-shift character for character set 15
AD	-1	Invalid value for the `c` argument
AE	2	`LS2`: Locking-shift character for character set 2
AF	3	`LS3`: Locking-shift character for character set 3
B0 to FF	-1	Invalid value for the `c` argument

Examples

The following program uses `$lockingcharcode` to get the character set number associated with a locking-shift character.

```
#include <stdio.h>

char c;
short char_set_num;

main()
{
   c = '\xA0';
   char_set_num = $lockingcharcode(c);

   printf("The character set number associated with %#x = %hd\n",
          c, char_set_num);
}
```

The output from the preceding program is as follows:

```
The character set number associated with 0xa0 = 1
```

Related Functions

`$charcode`, `$lockingshiftintroducer`, `$lockingshiftselector`, `$singleshiftchar`

$lockingshiftintroducer

Purpose

Gets the value of the locking-shift introducer character.

Syntax

```
short $lockingshiftintroducer(void)

   /*  OR  */

short $lsi(void);
```

Explanation

The `$lockingshiftintroducer` function gets the value of the locking-shift introducer character. The locking-shift introducer character is the same for all right graphic character sets.

Return Value

The `$lockingshiftintroducer` function returns the value of the locking-shift introducer character (90 hexadecimal).

Examples

The following program uses `$lockingshiftintroducer` to get the value of the locking-shift introducer character.

```
#include <stdio.h>

short locking_shift_intro_char;

main()
{
   locking_shift_intro_char = $lsi();

   printf("The locking-shift introducer character is %#x\n",
          locking_shift_intro_char);
}
```

The output from the preceding program is as follows:

```
The locking-shift introducer character is 0x90
```

Related Functions

`$lockingcharcode`, `$lockingshiftselector`

$lockingshiftselector

Purpose

Gets the value of the locking-shift character for a specified character set.

Syntax

```
short $lockingshiftselector(short char_set_id)

     /*  OR  */

short $lss(short char_set_id)
```

Explanation

The `$lockingshiftselector` function gets the value of the locking-shift character for the character set specified by `char_set_id`. Each right graphic character set has a different locking-shift character associated with it. The `char_set_id` argument is a `short` value specifying the number of a character set.

If you include the `char_sets.incl.c` header file into the source module, you can use a defined constant, such as `LATIN_1_CHAR_SET`, for a character set number when you specify the `char_set_id` argument.

Return Value

The `$lockingshiftselector` function returns the value of the locking-shift character for the character set specified by `char_set_id`. The following table lists the return value of `$lockingshiftselector` for each character set.

(Page 1 of 2)

Character Set Number	**Character Set Defined Constant**	**Return Value (in hex.)**
1	`LATIN_1_CHAR_SET`	A0
2	`KANJI_CHAR_SET`	AE
3	`KATAKANA_CHAR_SET`	AF
4	`HANGUL_CHAR_SET`	A1
5	`SIMPLIFIED_CHINESE_CHAR_SET`	A2
6	`CHINESE1_CHAR_SET`	A3
7	`CHINESE2_CHAR_SET`	A4

(Page 2 of 2)

Character Set Number	Character Set Defined Constant	Return Value (in hex.)
8	`USER_DOUBLE_BYTE_CHAR_SET`	A5

Examples

The following program uses `$lockingshiftselector` to get the value of the locking-shift character for the Latin alphabet No. 1 character set.

```
#include <char_sets.incl.c>
#include <stdio.h>

short locking_shift_char;
short char_set_id = LATIN_1_CHAR_SET;

main()
{
   locking_shift_char = $lockingshiftselector(char_set_id);

   printf("For Latin alphabet No. 1, the locking-shift character is
%#x\n",
          locking_shift_char);
}
```

The output from the preceding program is as follows:

```
For Latin alphabet No. 1, the locking-shift character is 0xa0
```

Related Functions

`$lockingcharcode`, `$lockingshiftintroducer`

$shift

Purpose

Converts an NLS string into an equivalent canonical string using a specified default character set.

Syntax

```
/*  Used with a char_varying argument  */

char_varying(m) $shift(char_varying(n) cv_string [,short def_char_set])

/*  Used with a char * argument        */

char_varying(m) $shift(char *string, short length, [,short def_char_set])
```

Explanation

The `$shift` function converts an NLS string into an equivalent canonical string using the default character set specified by `def_char_set`. In a *canonical string*, each non-ASCII character is preceded by a single-shift character. If you omit the `def_char_set` argument, the `$shift` function assumes that the default character set is Latin alphabet No. 1.

If the subject string is a `char_varying` string, the `$shift` function can take two arguments.

- `cv_string` is a `char_varying` expression specifying the string to convert. The current length of `cv_string` must not exceed 16,383 characters.
- `def_char_set`, an optional argument, is a `short` value specifying the number for the default character set of the string to convert.

If the subject string is an array of `char`, the `$shift` function can take three arguments.

- `string` is a pointer to `char` locating the string to convert.
- `length` specifies the length, in bytes, of the array. The `length` argument must not exceed 16,383 bytes.
- `def_char_set`, an optional argument, is a `short` value specifying the number for the default character set of the string to convert.

If `cv_string` is a `char_varying` expression that evaluates to a constant or if `string` is a character-string literal, the default character set of the constant is, by definition, the character set indicated in a `#pragma` specifying `default_char_set` or the corresponding compiler argument or, in the absence of both, Latin alphabet No. 1. With either type of constant, it is **invalid** to specify another default character set.

If you include the `char_sets.incl.c` header file into the source module, you can use a defined constant, such as `LATIN_1_CHAR_SET`, for a character set number when you specify the `def_char_set` argument.

The `$shift` function always produces a valid NLS string. The subject string is assumed to be valid. When the subject string contains an invalid NLS character, `$shift` signals the `warning` condition at run time. If your program has not established a function to handle the `warning` condition, the default operating system handler displays one of the following errors on the terminal's screen, and the program continues execution.

Error Code Name	**Number**
`e$invalid_right_graphic_char`	`4151`
`e$missing_lockshift_selector`	`4155`
`e$truncated_locking_shift`	`4154`
`e$truncated_multibyte_char`	`4153`
`e$truncated_single_shift`	`4152`
`e$unknown_character_set`	`4156`

To establish a handler for the `warning` condition, use the `s$enable_condition` subroutine. See the *VOS C Subroutines Manual (R068)* for information on `s$enable_condition`.

Return Value

The `$shift` function returns a `char_varying` string containing a canonical string that is the equivalent of the subject string. The maximum length possible for the returned string is twice the current length of `cv_string` if a `char_varying` argument is used, or twice the length of the array if a pointer to `char` argument is used.

When the subject string contains invalid NLS characters, `$shift` returns an ASCII `SUB` character in place of each invalid character.

Examples

The following program uses `$shift` to convert an NLS string into an equivalent canonical string using Latin alphabet No. 1 as the default character set.

```
#pragma default_char_set (latin_1)

#include <stdio.h>
#include <string.h>

char string[15] = "\accent135el\accent136eve";

char_varying(30) return_string;


int i;

main()
{
   printf("Before the $shift call, the character codes in string
are:\n");
   i = 0;
   while (string[i] != '\0')
      {
      printf("%#x ", string[i]);
      i++;
      }

  return_string = $shift(string, 5); /* Default character set is
*/
                              /*  specified in the pragma directive  */

   strncpy_nstr_vstr( string, &return_string, $iclen(return_string)
);
   printf("\n\nAfter the $shift call, the character codes in string
are:\n");
   i = 0;
   while (string[i] != '\0')
      {
      printf("%#x ", string[i]);
      i++;
      }
   printf("\n");
}
```

The output from the preceding program is as follows:

```
Before the $shift call, the character codes in string are:
0xe9 0x6c 0xe8 0x76 0x65

After the $shift call, the character codes in string are:
0x80 0xe9 0x6c 0x80 0xe8 0x76 0x65
```

Related Functions

```
$unshift
```

$singleshiftchar

Purpose

Gets the single-shift character for a specified character set.

Syntax

```
char $singleshiftchar(short char_set_id)

     /*  OR  */

char $ss(short char_set_id)
```

Explanation

The `$singleshiftchar` function gets the value of the single-shift character for the character set specified by `char_set_id`. The `char_set_id` argument is a `short` value specifying the number of a character set.

If you include the `char_sets.incl.c` header file into the source module, you can use a defined constant, such as `LATIN_1_CHAR_SET`, for a character set number when you specify the `char_set_id` argument.

Return Value

The `$singleshiftchar` function returns the value of the single-shift character for the specified character set. The following table lists the return value of `$singleshiftchar` for each character set.

(Page 1 of 2)

Character Set Number	Character Set Defined Constant	Return Value (in hex.)
0	`ASCII_CHAR_SET`	0F
1	`LATIN_1_CHAR_SET`	80
2	`KANJI_CHAR_SET`	8E
3	`KATAKANA_CHAR_SET`	8F
4	`HANGUL_CHAR_SET`	81
5	`SIMPLIFIED_CHINESE_CHAR_SET`	82
6	`CHINESE1_CHAR_SET`	83

(Page 2 of 2)

Character Set Number	Character Set Defined Constant	Return Value (in hex.)
7	`CHINESE2_CHAR_SET`	84
8	`USER_DOUBLE_BYTE_CHAR_SET`	85

Examples

The following program uses `$singleshiftchar` to get the single-shift character for the Latin alphabet No. 1 character set.

```
#include <char_sets.incl.c>
#include <stdio.h>

char ss_char;

main()
{
   ss_char = $singleshiftchar(LATIN_1_CHAR_SET);

   printf("For Latin alphabet No. 1, the single-shift character is
%#x\n",
       ss_char);
}
```

The output from the preceding program is as follows:

```
For Latin alphabet No. 1, the single-shift character is 0x80
```

Related Functions

`$lockingcharcode`, `$lockingshiftintroducer`, `$lockingshiftselector`

$substr

Purpose

Converts a specified number of bytes from a given source into a `char_varying` value.

Syntax

```
char_varying(n) $substr( source [, short start [, short length]]);
```

Explanation

This explanation of the `$substr` built-in function covers the following subjects:

- `$substr` expressions as `char_varying` values
- `$substr` operands with the relational operators
- `$substr` operands with the address-of operator
- `$substr` operands and pointer arithmetic
- `$substr` expressions as arguments
- `$substr` expressions as left operands in assignments.

There are two distinct categories of `$substr` expressions. Depending on how a `$substr` expression is used, the expression either yields a `char_varying` value that is not an lvalue, or yields an lvalue. The lvalue is the memory location of a character, or a contiguous set of characters. A `$substr` expression yields an lvalue when the `$substr` expression is used as the operand of the address-of operator (`&`) or as the left operand in an assignment. If a `$substr` expression is used in either of these contexts, its first argument must be an lvalue. See the sections that follow for a full description of these two distinct uses of a `$substr` expression.

See the "Varying-Length Character String Type" section in Chapter 4 for information on that data type.

See Chapter 7 for information on all operations that are allowed with the `char_varying` type.

Using `$substr` Expressions as `char_varying` Values. When used to yield a `char_varying` value, `$substr` has the following syntax:

```
$substr( source [, short start [, short length]])
```

When used to yield a `char_varying` value, `$substr` converts `length` bytes of *`source`* starting at the byte specified in `start` into a `char_varying` value. The `char_varying(n)` result is conceptual and is **not** intended to describe the actual conversions and operations that occur with the `$substr` built-in function.

The *`source`* argument can be one of the following:

- a modifiable or nonmodifiable lvalue
- a character-string literal
- another `$substr` expression
- a function-call expression.

A modifiable or nonmodifiable lvalue can be any VOS C data type except a function type, incomplete type, or `void`. For example, *`source`* can be an array type, a structure type, or a union type.

The optional `start` argument is a starting location, in bytes, within *`source`* where the `$substr` operation will begin. The value given in the `start` argument must be equal to or greater than 1. If you omit `start`, its value is assumed to be 1.

> **Note:** The `$substr` built-in uses the value 1 (not the value 0 as in C array usages) as the first byte within *`source`*, even when *`source`* is an array.

The optional `length` argument specifies the length, in bytes, of the substring that the `$substr` operation will yield. The value given in the `length` argument must be equal to or greater than 0. The `length` value must not be greater than the length of *`source`* minus the number of bytes given in `start`. If you omit `length`, its value is assumed to be equal to the number of bytes remaining in *`source`* calculated as follows:

```
sizeof(source) - start + 1
```

When a `$substr` expression is used to yield a `char_varying` value, the `char_varying` result differs depending on whether the second and third arguments to `$substr` are constant expressions.

- If the second and third arguments of a `$substr` expression **are** constant expressions or are omitted, the result has the type `char_varying(`*`n`*`)` where *n*, the string's maximum length, is the value of `length`.

- If the second and third arguments of a `$substr` expression **are not** constant expressions, the result has the type `char_varying` with unspecified maximum length.

When the second and third arguments of `$substr` are not constant expressions, the result can be used only in contexts where the maximum length of the `char_varying` value is not required. For example, the result could not be used as a function argument, function parameter, or function return type.

In a `$substr` expression, the length of the *`source`* operand is determined as described in the following paragraphs.

If *source* is an array, the length of the array is assumed to be all of its characters up to and including any null character. For example, the following `$substr` expression yields the `char_varying` string `defghi\0`.

```
char array[10] = "abcdefghi";

$substr(array, 4)
```

If *source* is a character-string literal, the length of the string literal is assumed to be all of its characters up to but **not** including the implicit null character. For example, the following `$substr` expression yields the `char_varying` string `defghi`.

```
$substr("abcdefghi", 4)
```

By specifying an explicit `length` argument, you can include the character-string literal's implicit null character in the `char_varying` result. For example, the following `$substr` expression yields the `char_varying` string `defghi\0`.

```
$substr("abcdefghi", 4, 7)
```

If *source* is a `char_varying` string, the length of the string is assumed to be the current length of the `char_varying` string, not the maximum length. For example, the following `$substr` expression yields the `char_varying` string `defghi`.

```
char_varying(10) cv_array = "abcdefghi";

$substr(cv_array, 4)
```

If *source* is a data type other than an array, character-string literal, or `char_varying` string, the length of *source* is assumed to be identical to the expression `sizeof(`*`data_type`*`)`.

Using `$substr` Operands with the Relational Operators. A `$substr` expression can be used as one or both operands of the relational operators: <, <=, >, and >=. For example, the relational expression in the following program fragment has two `$substr` operands.

```
struct s
   {
   char array[5];
   int i_num;
   };

struct s struct_1 = {"abc", 100};
struct s struct_2 = {"abaa", 100};
   .
   .
   .
if ( $substr(struct_1, 2, 4) > $substr(struct_2, 2, 4) )    /*
Relational  */
   puts("struct_1 compares greater than struct_2.");        /*
expression  */
else
   puts("struct_1 does not compare greater than struct_2.");
```

The output from the preceding example is as follows:

```
struct_1 compares greater than struct_2.
```

In the preceding example, both operands of the > operator are `$substr` expressions and have the `char_varying` type. When two `char_varying` types are the operands in a relational operation, the two strings are compared, character by character, until two characters are found that are different. The string whose differing character has the higher ASCII code compares greater. In the case of unequal length strings, the shorter string is padded with spaces to the length of the longer operand.

Using `$substr` Operands with the Address-of Operator. When a `$substr` expression is the operand of the address-of operator (`&`), the result has the type "pointer to `char`." **The result is not the address of a** `char_varying(`*`n`*`)` object. When a `$substr` expression is the operand of the `&` operator, the result is the address of the byte that is the starting location within the *`source`* argument where the `$substr` operation will begin.

The following example shows an expression that uses the `$substr` built-in as the operand for the `&` operator. In this example, `$substr` has a `char` array argument.

```
char array[10] = "abcdefghi";
char *char_ptr;
short s = 1;
short l = 3;

char_ptr = &($substr(array, s, l));
```

In the preceding example, the expression `&($substr(array, s, l))` yields the address of `array[0]` because arrays in C begin with element 0. Notice that, in the `$substr` expression, a starting location of `array[0]` is specified by setting the second argument, `s`, equal to 1. Index counting with `$substr` begins with 1. The third argument, `l`, in the `$substr` built-in has no relevance to the address-of operation.

When the operand of the `&` operator is a `$substr` expression and when the first argument to `$substr` has the `char_varying` type, the result is the address of the starting location within the string where the `$substr` operation will begin. In a `char_varying` object, the string is preceded by a 2-byte current length field. Therefore, the first character in the string is offset two bytes beyond the beginning address of the `char_varying` object. The following example shows an expression that uses a `$substr` built-in as the operand of the `&` operator. In this example, `$substr` has a `char_varying` argument.

```
char_varying(10) cv_string = "abcdefghij";
char *char_ptr;

char_ptr = &($substr(cv_string, 1));
```

In the preceding example, the expression `&($substr(cv_string, 1))` yields the address of the first character (`a`) within the string part of `cv_string`, which is located at the address of `cv_string` plus two bytes. The two added bytes contain the 2-byte current length field.

Using `$substr` Operands and Pointer Arithmetic. In pointer arithmetic operations with the `++`, `--`, `+`, or `-` operators, when the address of a `$substr` expression is used as an operand, the scale factor is one byte because the result of the address of a `$substr` expression always has the type "pointer to `char`." The following example shows a pointer arithmetic expression that uses the `$substr` built-in as the operand of the `+` operator.

```
char_varying(10) cv_string = "abcdefghij";
int i = 2;
    .
    .
    .
printf( "The pointed-to character is %c\n", *(& $substr(cv_string, 1)
+ i) );
```

The output from the preceding example is as follows:

```
The pointed-to character is c
```

In the preceding example, the expression `& $substr(cv_string, 1)` yields the address of `a`. In the pointer arithmetic operation, the scale factor is 1 because the size of the pointed-to object, a single `char`, is 1. The address offset is 2, calculated as `i` multiplied by the scale factor (`i * 1`). Thus, the expression `(& $substr(cv_string, 1) + i)` yields the address of `c`, an address 2 bytes beyond the address of `a`. When this expression is dereferenced, it yields the character `c`.

For information on pointer arithmetic, see the "Pointer Arithmetic" section in Chapter 4.

Using `$substr` Expressions as Arguments. You can pass the **address** of a `$substr` expression as an argument in a function call. As explained in "Using `$substr` Operands with the Address-of Operator" earlier in the Explanation, the address of a `$substr` expression yields a "pointer to `char`," **not** a "pointer to `char_varying`." The compiler diagnoses any attempt to pass the `$substr` expression itself as an argument unless the called function is declared with a function prototype. If the called function has a prototype, the pointer argument is implicitly converted to the type of the corresponding parameter.

When the `$substr` built-in generates a result of `char_varying(`*n*`)`, the maximum length, *n*, can vary depending on the value of the second and third arguments to `$substr`. When a `char_varying` value is passed as an argument, the argument's maximum length is assigned to the corresponding parameter. If the `char_varying` result of `$substr` does not have a constant maximum length, the compiler cannot locate the next parameter in the called function. Thus, in the absence of a prototype or a cast, passing a `$substr` expression itself as an argument is not allowed because the result of such an expression has a dynamically produced data type `char_varying(`*n*`)`, where *n* can vary from one function call to the next.

The following example illustrates how the address of a `$substr` expression can be passed as an argument.

```
char_varying(10) cv_string = "abcdefghij";

void func(char *char_ptr);

main()
{
   func( & $substr(cv_string, 6, 5) );
}

void func(char *char_ptr)
{
   printf("The characters are %.5s\n", char_ptr);
}
```

The output from the preceding program is as follows:

```
The characters are fghij
```

In the preceding example, notice that the argument to `func` specifies the address of a `$substr` expression. The corresponding parameter, `char_ptr`, is a pointer to a `char` because the address of a `$substr` expression yields a pointer to `char`.

Although it is not valid to pass the `$substr` expression itself as an argument, you can pass such an expression if it is explicitly cast to the required type. The following example illustrates how a `$substr` expression can be cast and then passed as an argument in a function call.

```
char_varying(10) cv_string = "abcdefghij";

void func();

short s, l;

main()
{
   s = 6;
   l = 3;

   func((char_varying(5))$substr(cv_string, s, l)); /*  Argument is
cast     */
}                                        /*  before it is passed  */


void func()
char_varying(5) cv_param;
{
   printf("The characters are %v\n", &cv_param);
}
```

The cast expression `(char_varying(5))$substr(cv_string, s, 1)` yields a value of the type `char_varying(5)`. When a `char_varying` value is passed as an argument, the maximum length of the argument is assigned to the corresponding parameter. When a `$substr` expression is cast, the current length of the result is either the maximum length specified in the cast or the value given in the third argument to `$substr`, whichever is less. In this example, the current length of the cast expression is 3.

Using `$substr` Expressions as Left Operands in Assignments. When used as the left operand of the assignment operator, `$substr` has the following syntax:

```
$substr( target[, short start[, short length]] ) = expression
```

In an assignment expression that has a `$substr` expression as the left operand, the value of *`expression`* is converted to the type `char_varying` if *`expression`* does not have the `char_varying` type.

The `$substr` built-in writes, into *`target`*, `length` characters of the `char_varying` value that *`expression`* yields. The *`target`* argument is modified starting at the character within *`target`* that is specified in `start`. When used in this context, `$substr` provides a template of contiguous characters (bytes) that is used to modify the storage locations, or a portion of the storage locations occupied by the object specified by *`target`*.

> **Note:** The `$substr` built-in modifies the indicated number of bytes in *`target`* without regard for the range of values that are valid in an object of the type specified by *`target`*, and without regard for the number of bytes that are allocated for an object of the type specified by *`target`*.

The *`expression`* argument must be a `char_varying` type or must be convertible to `char_varying`.

The *`target`* argument can be a modifiable lvalue or another `$substr` expression that has a modifiable lvalue as its first operand. The *`target`* argument can be a character-string literal only if you specified `no_common_constants` in a `#pragma` preprocessor control line.

The optional `start` argument is a starting location, in bytes, within *`target`* where the characters will be written. The value given in the `start` argument must be equal to or greater than 1. If you omit `start`, its value is assumed to be 1.

> **Note:** The `$substr` built-in uses the value 1 (not the value 0 as in C array usage) as the first byte within *`target`*, even when *`target`* is an array.

The optional `length` argument specifies the number of bytes of the converted `char_varying` value from *`expression`* that will be written into *`target`*. The value given in the `length` argument must be equal to or greater than 0. If you omit `length`, its value is assumed to be equal to the number of bytes remaining in *`target`*.

- If the number of characters in *`expression`* is less than `length`, the converted `char_varying` value is padded with space characters on the right.
- If the number of characters in *`expression`* is greater than `length`, the converted `char_varying` value is truncated from the right.

The following example shows how `$substr` can be used as the left operand in an assignment.

```
char array[10] = "abcdefghi";
   .
   .
   .
$substr(array, 4, 3) = "UVWXYZ";
printf("array = %s\n", array);
```

The output from the preceding example is as follows:

```
array = abcUVWghi
```

The result of the assignment expression `$substr(array, 4, 3) = "UVWXYZ"` has the type `char_varying(3)` because the third argument to `$substr` equals the value 3. The characters in the `char_varying(3)` result are `UVW`.

When `$substr` is the left operand in an assignment and the *`target`* argument is a `char_varying` string, the assignment does not affect the current length of the target string. For example:

```
char_varying(10) cv_string = "abc";

$substr(cv_string, 1, 1) = "X";
```

After the preceding assignment, the value of `cv_string` is `Xbc`. This assignment does not change the length of `cv_string`. The current length remains three characters.

> **Note:** Be careful if you use `$substr` as the left operand in an assignment to write to a `char_varying` string because `$substr` bypasses the mechanism that keeps the current-length field of the `char_varying` string consistent with the contents of the string.

The following assignment sets the current length of `cv_string` to one character.

```
* (short*)&cv_string = 1;
```

When `$substr` is the left operand in an assignment, the `$substr` expression is sometimes called a *pseudovariable* following the VOS PL/I terminology.

Return Value

When a `$substr` expression is used to yield a `char_varying` value, the `char_varying` return value differs depending on whether the second and third arguments to `$substr` are constant expressions.

- If the second and third arguments of a `$substr` expression **are** constant expressions or are omitted, the return value has the type `char_varying(`*`n`*`)` where *n*, the string's maximum length, is the value of `length`.
- If the second and third arguments of a `$substr` expression **are not** constant expressions, the return value has the type `char_varying` with unspecified maximum length.

See "Using `$substr` Expressions as `char_varying` Values" in the Explanation for detailed information on the return value of the `$substr` built-in function.

Examples

The following program uses `$substr` as the right operand in an assignment to generate a `char_varying` value, and as the left operand in an assignment to overwrite five characters within `cv_string`.

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

char_varying(10) cv_string = "abcdefghij";

main()
{
   $substr(cv_string, 1, 5) = $substr(cv_string, 6, 5);

   printf("cv_string = %v\n", &cv_string);
}
```

The output from the preceding program is as follows:

```
cv_string = fghijfghij
```

Related Functions

`strlen_vstr`

$unshift

Purpose

By removing all single-shift characters for a specified character set, converts an unambiguous canonical string or a common string into an ambiguous NLS string with a default character set.

Syntax

```
/*  Used with a char_varying argument  */

char_varying(n) $unshift(char_varying(m) cv_string [,short def_char_set])

/*  Used with a char * argument        */

char_varying(n) $unshift(char *string, short length, [,short
def_char_set])
```

Explanation

By removing all single shifts from a specified string, the `$unshift` function converts an unambiguous canonical string or a common string into an ambiguous NLS string with the default character set specified by `def_char_set`. If you omit the `def_char_set` argument, the `$unshift` function assumes that the default character set is Latin alphabet No. 1.

When you omit the `def_char_set` argument or when Latin alphabet No. 1 is specified as the default character set, the `$unshift` function generates a common string. A *common string* can contain characters from any right-graphic character set and has a default character set of Latin alphabet No. 1. In a common string, a single-shift character precedes each right graphic character except for Latin alphabet No. 1 characters. A common string contains no locking-shift characters. Common strings can be written to a terminal or to a text file.

If the subject string is a `char_varying` string, the `$unshift` function can take two arguments.

- `cv_string` is a `char_varying` expression specifying the string to convert. The current length of `cv_string` must not exceed 16,383 characters.

- `def_char_set`, an optional argument, is a `short` value specifying the number for the default character set of the string to convert.

If the subject string is an array of `char`, the `$unshift` function can take three arguments.

- `string` is a pointer to `char` locating the string to convert.
- `length` specifies the length, in bytes, of the array. The `length` argument must not exceed 16,383 bytes.
- `def_char_set`, an optional argument, is a `short` value specifying the number for the default character set of the string to convert.

If you include the `char_sets.incl.c` header file into the source module, you can use a defined constant, such as `LATIN_1_CHAR_SET`, for a character set number when you specify the `def_char_set` argument.

When the subject string contains an invalid NLS character, `$unshift` signals the `warning` condition at run time. If your program has not established a function to handle the `warning` condition, the default operating system handler displays one of the following errors on the terminal's screen, and the program continues execution.

Error Code Name	**Number**
`e$invalid_right_graphic_char`	`4151`
`e$missing_lockshift_selector`	`4155`
`e$unknown_character_set`	`4156`
`e$truncated_locking_shift`	`4154`
`e$truncated_multibyte_char`	`4153`
`e$truncated_single_shift`	`4152`

To establish a handler for the `warning` condition, use the `s$enable_condition` subroutine. See the *VOS C Subroutines Manual (R068)* for information on `s$enable_condition`.

Return Value

The `$unshift` function returns a `char_varying` string containing a string that is the equivalent of the subject string but with all single-shift characters for a particular character set removed.

The maximum length possible for the returned string is twice the current length of `cv_string` if a `char_varying` argument is used, or twice the length of the array if a pointer to `char` argument is used. When you omit the `def_char_set` argument or when Latin alphabet No. 1 is specified as the default character set, the `$unshift` function generates a common string. To create a common string, `$unshift` may have to add single-shift characters to right graphic characters from character sets other than Latin alphabet No. 1.

When the subject string contains invalid NLS characters, `$unshift` returns an ASCII `SUB` character in place of each invalid character.

Examples

The following program uses `$unshift` to remove, from a specified string, all single shifts preceding characters from Latin alphabet No. 1.

```
#pragma default_char_set (none)

#include <stdio.h>
#include <string.h>

char string[15] = "\accent135el\accent136eve";


char_varying(30) return_string;


short i;

main()
{
   printf("Before the $unshift call, the character codes in string
are:\n");
   i = 0;
   while (string[i] != '\0')
      {
      printf("%#x ", string[i]);
      i++;
      }

   return_string = $unshift(string, 7);

  strncpy_nstr_vstr( string, &return_string, ($iclen(return_string)
+ 1) );
  printf("\n\nAfter the $unshift call, the character codes in string
are:\n");
   i = 0;
   while (string[i] != '\0')
      {
      printf("%#x ", string[i]);
      i++;
      }
   printf("\n");
}
```

The output from the preceding program is as follows:

```
Before the $unshift call, the character codes in string are:
0x80 0xe9 0x6c 0x80 0xe8 0x76 0x65

After the $unshift call, the character codes in string are:
0xe9 0x6c 0xe8 0x76 0x65
```

Related Functions

```
$shift
```

Appendix A: Internal Storage

Internally, all data is stored as a series of *bits* or binary digits. Storage is divided into groups of eight bits, called *bytes*. Sixteen bits or two bytes constitutes a *word* of storage.

The examples in this appendix use both binary and hexadecimal format to show internal storage. Figure A-1 shows the contents of one word of memory in both binary and hexadecimal format. In binary format, each bit is equal to either 1 or 0.

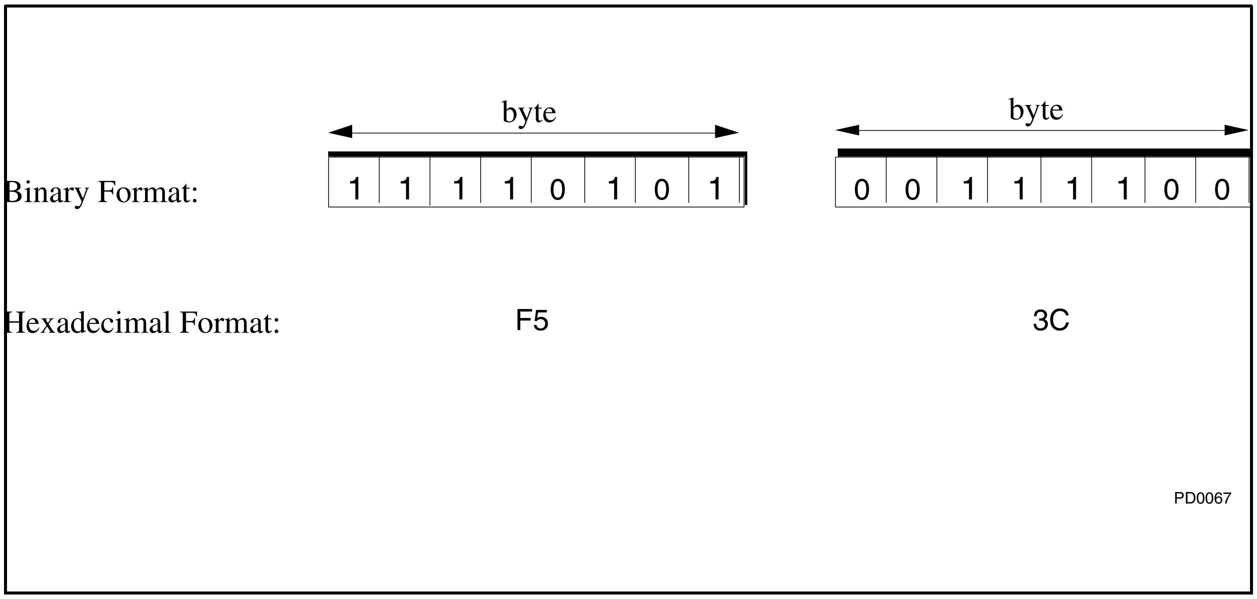


Figure A-1. Binary and Hexadecimal Format

In hexadecimal format, the first byte contains F5 hexadecimal, and the second byte contains 3C hexadecimal. In the byte or series of bytes that constitute an object, the leftmost bit is the *most significant* or *high-order* bit. The rightmost bit is the *least significant* or *low-order* bit.

A data type's *internal representation* is the format used to represent specific values in an object of that type. Each format uses a particular bit pattern to represent a specific value. Each object occupies a region of data storage or memory.

This appendix explains the internal representation of the integral and floating-point types.

Chapter 4 contains information on the allocation of each data type, including bit-field allocation and alignment. Chapter 4 also provides information on the internal representation of some data types, such as `char_varying` strings.

Internal Representation for Unsigned Integral Types

The unsigned integral types, including `unsigned char`, are stored using straight unsigned binary notation. Unsigned types cannot represent negative values. In the unsigned types, the most significant bit is not a sign bit. All bits represent the magnitude of the value. For example, an `unsigned short int` occupies 16 bits. When an `unsigned short int` object stores the value 32,783, it has the following bit pattern:

```
10000000 00001111
```

Operations on the unsigned integral types behave according to the rules of arithmetic modulo $2^{@n@}$ where *n* is the number of bits in the unsigned integral type. If you add 1 to the largest value an unsigned integral variable can hold, the result is always 0, or fixed-point overflow occurs if you select the `-fixedoverflow` compiler argument. As an example, with an `unsigned short int`, if you add 1 to 65,535 (its maximum value), the result is 0.

Internal Representation for Signed Integral Types

The signed integral types, including `signed char`, use the most significant bit as a sign bit. The remaining bits contain the magnitude of the value. Negative integer values are stored in two's complement binary notation. In two's complement notation, the bit pattern representing `-x` is the one's complement of the bit pattern representing `x-1`.

As an example of the storage of a positive number, when a `signed short int` object stores the value 12, it has the following bit pattern:

```
00000000 00001100
```

In the preceding example, the high-order bit equals 0, indicating a positive value. The remaining 15 bits equal the value 12.

As an example of the storage of a negative number, when a `signed short int` stores the value --12, it has the following bit pattern:

```
11111111 11110100
```

In this example, the high-order bit equals 1, indicating a negative value. In two's complement notation, the bit pattern for the value --12 is the one's complement of the bit pattern for 12 minus 1.

Internal Representation for Floating-Point Types

The floating-point internal representations conform to the IEEE standard for single-precision and double-precision normalized values. That standard is described in the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985).

The internal representation of a floating-point value consists of three parts:

- `s`: a sign bit
- `e`: a biased exponent representing the power of 2 by which the mantissa is multiplied
- `f`: a mantissa (stored normalized without the leading 1).

Figure A-2 shows the bit mapping for the internal representation of both single-precision (`float`) and double-precision (`double`) values.

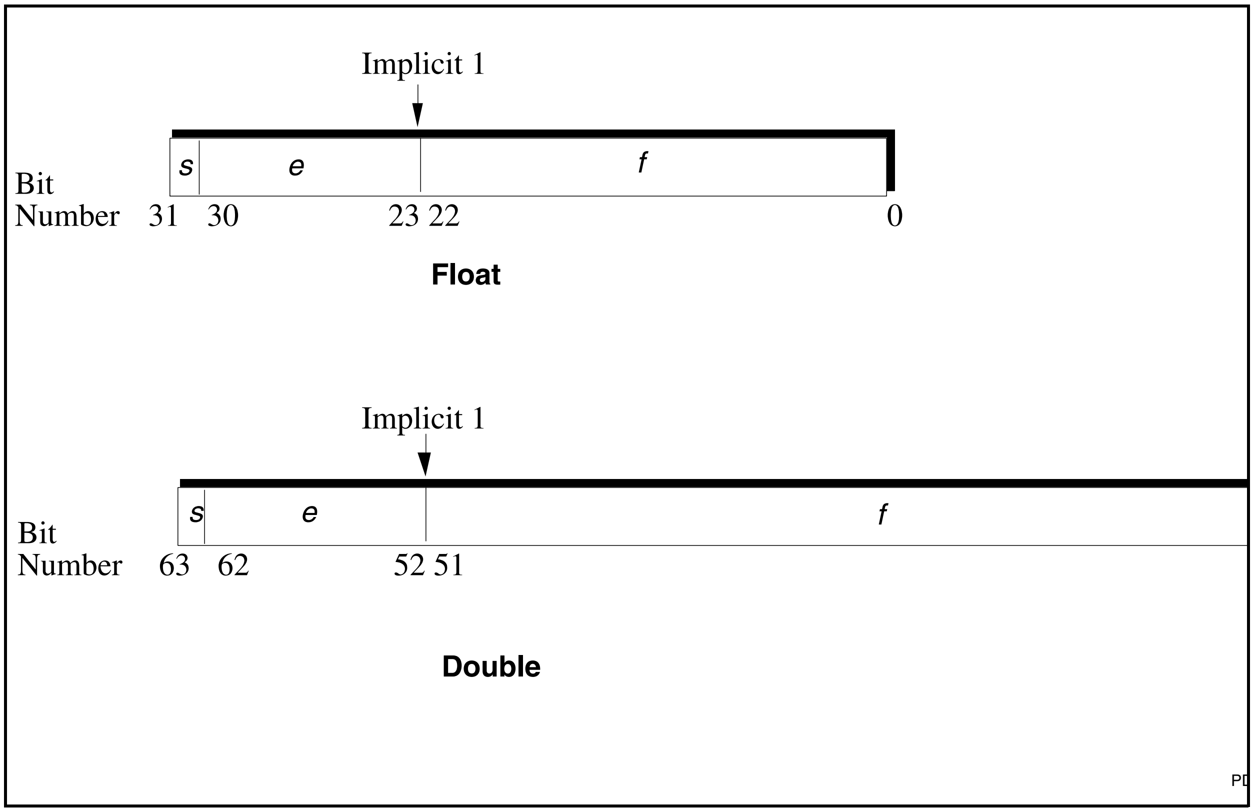


Figure A-2. IEEE Format for Floating-Point Values

The sign bit, `s`, can be either 1 for a positive value or 0 for a negative value. The exponent, `e`, is an integer stored in unsigned binary format. To ensure that the exponent is positive, a *bias* is added to the exponent before it is stored. Table A-1 shows information on the bias and range of possible values for `e`. In the table, the maximum and minimum `e` values are biased exponents (that is, the values shown in the table have a bias already added to them).

Table A-1. Exponent Bias and Range

Component	**`float`**	`double`
Bias	127	1023
Maximum `e`	255	2047
Minimum `e`	0	0

The fractional part of the mantissa, `f`, is a binary value with an implicit 1 immediately preceding its most significant bit. Table A-2 shows size information on the three components that make up the internal representation of both `float` and `double` values. Because the fractional part of the mantissa is preceded by an implicit 1, a `float` has 24-bit binary precision, and a `double` has 53-bit binary precision.

Table A-2. Floating-Point Internal Representation Sizes

	Length in Bits	
Component	**`float`**	**`double`**
Sign	1	1
Exponent	8	11
Fraction	23	52
Total	32	64

The general formula for determining the value of a single-precision (`float`), floating-point value is:

$(-1^{@s@})(1.f) * (2^{|(@e@ - 127)|})$

The three exceptions to the preceding formula are as follows:

- If `e` equals 255 and `f` equals 0, the value is positive or negative infinity, depending on the value of `s`.
- If `e` equals 255 and `f` is not equal to 0, the value is not a number.
- If `e` equals 0 and `f` equals 0, the value is 0.

The general formula for determining the value of a double-precision (`double`), floating-point value is:

$(-1^{@s@})(1.f) * (2^{|(@e@ - 1023)|})$

The three exceptions to the preceding formula are as follows:

- If `e` equals 2047 and `f` equals 0, the value is positive or negative infinity, depending on the value of `s`.
- If `e` equals 2047 and `f` is not equal to 0, the value is not a number.
- If `e` equals 0 and `f` equals 0, the value is 0.

The "Internal Representation Examples" section that follows contains examples of `float` and `double` storage.

Internal Representation Examples

Table A-3 shows how sample values are stored in the integral and floating-point data types. In the table, the internal representation of each value is shown in hexadecimal format.

Table A-3. Internal Representation Examples

Data Type	Value	Internal Representation	Explanation
`signed char`	-11	`F5`	High-order sign bit with 7-bit value in two's complement notation
`unsigned char`	11	`0B`	8-bit value in unsigned binary notation
`signed short`	-1	`FFFF`	High-order sign bit with 15-bit value in two's complement notation
`unsigned short`	65,535	`FFFF`	16-bit value in unsigned binary notation

`signed int`
`signed long`
`enum object`
	45	`0000002D`	High-order sign bit with31-bit value in binary notation

`unsigned int`
`unsigned long`
	129	`00000081`	32-bit value in unsigned binary notation

`float`	1.0E+10	`501502F9`	IEEE format for a single-precision, floating-point value

`double`	1.0E+10	`4202A05F`
		`20000000`
			IEEE format for a double-precision, floating-point value

Appendix B: Summary of VOS C Extensions

This appendix lists the VOS C extensions to the definition of the C language and library functions described in the ANSI C Standard.

Predefined Macros

The VOS C compiler predefines the following macros:

- `__VOS__` indicates the VOS operating system environment.
- `__PROTOTYPES__` indicates that the compiler supports function prototypes.
- Two or more macros indicate the processor family and processor type based on the value specified in the `processor` option in a `#pragma` preprocessor control line, or in the `-processor` command-line argument.

Preprocessor Directives

Three VOS C preprocessor directives are extensions: `#list`, `#nolist`, and `#page`.

Characters

The source character set conforms to the ANSI C Standard except that VOS C also allows the use of the dollar sign character (`$`).

Character Constants

The VOS C compiler allows double-byte character constants, which have the type `short int`.

Keywords

The following keywords are extensions: `accept`, `char_varying`, and `ext_shared`.

Storage-Class Specifiers

The VOS C compiler recognizes the `ext_shared` storage-class specifier for declaring objects that will be shared by multiple tasks in a tasking application.

Data Types

The varying-length character string (`char_varying`) data type is a VOS C extension.

Type Qualifiers

In a function declaration or definition, the VOS C compiler allows you to associate the `volatile` type qualifier with the function itself.

Alignment Specifiers

The VOS C compiler recognizes the `$shortmap` and `$longmap` alignment specifiers.

Multiple Object Definitions

In VOS C, an object (but not a function) can have multiple definitions **if** the definitions are identical in all respects, including initializers.

Structures and Unions

Two VOS C extensions relate to structures and unions.

- When you declare a structure or union, the object can contain one or more anonymous structures and unions as members. An anonymous structure or union is one that is nested within another structure or union and that does not have a name associated with it.

- When you reference a structure or union member, you can use the member name alone without preceding the member name with a qualifying expression. VOS C allows this partial qualification when the member name unambiguously identifies the structure or union variable and the member.

Address-of Operator

In VOS C, certain constant values can be used as the operand of the address-of operator **if** the expression is part of an argument list. This VOS C extension makes it possible to pass constant values to procedures, such as VOS operating system subroutines, that expect arguments to be passed by reference.

Miscellaneous Usages

The following miscellaneous usages are allowed by the VOS C compiler:

- specifying an external array definition for which the number of elements is not given (thereby implying an `extern` declaration)

- using an expression yielding a non-address in a context where an address is required

- referencing an identifier (implicitly defined as an `int`) before it is explicitly declared

Generic, String-Manipulation Functions

In VOS C, many of the functions in the `string.h` header file, such as `strcpy`, are generic, string-manipulation functions. Each generic, string-manipulation function performs the specified operation whether its arguments are pointers to `char`, pointers to `char_varying` strings, or a combination of the two pointer types. With a generic, string-manipulation function, the VOS C compiler takes a special action if an argument contains the address of a `char_varying` string.

Many of the functions in the `string.h` header file, such as `strcpy_vstr_vstr`, are defined to accept a pointer to a `char_varying` string in one or more argument positions.

Library Functions

In addition to the generic, string-manipulation functions, the following VOS C library functions, described in Chapter 11, are extensions.

```
access              lockf
alloca              lseek
chdir               memccpy
close               onexit
creat               open
fdopen              putw
getw                s$c_get_portid
isascii             s$c_get_portid_from_fildes
isatty              sleep
isoctal             toascii
kill                write
```

Also, the `_tolower` and `_toupper` macros are VOS C extensions.

The `fopen` and `freopen` Functions

As optional extensions, the `fopen` function (and in a more limited manner, the `freopen` function) allows argument values that specify the following:

- file organization
- locking mode
- raw input and output modes (for a terminal device only)
- bulk raw input mode (for a terminal device only)
- associated port.

In VOS C, if a file is opened in append mode, you can use the `_TRADITIONAL_` external variable to change the behavior of append mode when seeking with the `fseek` function.

Conversion Specifiers

The formatted input and formatted output functions from the `stdio.h` header file accept the `%v` conversion specifier, which is used for `char_varying` data.

Built-in Functions

Each of the built-in functions, such as `$substr`, described in Chapter 12 is a VOS C extension.

Appendix C:
The VOS Internal Character Code Set

Table C-1 shows the VOS internal character code set.

Table C-1. VOS Internal Character Code Set *(Page 1 of 10)*

Decimal Code	Hex Code	Symbol	Name
0	00	NUL	Null
1	01	SOH	Start of Heading
2	02	STX	Start of Text
3	03	ETX	End of Text
4	04	EOT	End of Transmission
5	05	ENQ	Enquiry
6	06	ACK	Acknowledge
7	07	BEL	Bell
8	08	BS	Backspace
9	09	HT	Horizontal Tabulation
10	0A	LF	Linefeed
11	0B	VT	Vertical Tabulation
12	0C	FF	Form Feed
13	0D	CR	Carriage Return
14	0E	SO	Shift Out
15	0F	SI	Shift In
16	10	DLE	Data Link Escape
17	11	DC1	Device Control 1
18	12	DC2	Device Control 2
19	13	DC3	Device Control 3

Table C-1. VOS Internal Character Code Set *(Page 2 of 10)*

Decimal Code	**Hex Code**	**Symbol**	**Name**
20	14	DC4	Device Control 4
21	15	NAK	Negative Acknowledge
22	16	SYN	Synchronous Idle
23	17	ETB	EOT Block
24	18	CAN	Cancel
25	19	EM	End of Medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File Separator
29	1D	GS	Group Separator
30	1E	RS	Record Separator
31	1F	US	Unit Separator
32	20	SP	Space
33	21	!	Exclamation Mark
34	22	“	Quotation Marks
35	23	#	Number Sign
36	24	$	Dollar Sign
37	25	%	Percent Sign
38	26	&	Ampersand
39	27	’	Apostrophe
40	28	(	Opening Parenthesis
41	29	)	Closing Parenthesis
42	2A	*	Asterisk
43	2B	+	Plus Sign
44	2C	,	Comma
45	2D	-	Hyphen, Minus Sign
46	2E	.	Period
47	2F	/	Slant

Table C-1. VOS Internal Character Code Set *(Page 3 of 10)*

Decimal Code	Hex Code	Symbol	Name
48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less-Than Sign
61	3D	=	Equals Sign
62	3E	>	Greater-Than Sign
63	3F	?	Question Mark
64	40	@	Commercial ‘‘at’’ Sign
65	41	A	Uppercase A
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K

Table C-1. VOS Internal Character Code Set *(Page 4 of 10)*

Decimal Code	**Hex Code**	**Symbol**	**Name**
76	4C	L	Uppercase L
77	4D	M	Uppercase M
78	4E	N	Uppercase N
79	4F	O	Uppercase O
80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[	Opening Bracket
92	5C	\	Reverse Slant
93	5D	]	Closing Bracket
94	5E	^	Circumflex
95	5F	_	Underline
96	60	`	Grave Accent
97	61	a	Lowercase a
98	62	b	Lowercase b
99	63	c	Lowercase c
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g

Table C-1. VOS Internal Character Code Set *(Page 5 of 10)*

Decimal Code	Hex Code	Symbol	Name
104	68	h	Lowercase h
105	69	i	Lowercase i
106	6A	j	Lowercase j
107	6B	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s
116	74	t	Lowercase t
117	75	u	Lowercase u
118	76	v	Lowercase v
119	77	w	Lowercase w
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Opening Brace
124	7C	|	Vertical Line
125	7D	}	Closing Brace
126	7E	~	Tilde
127	7F	DEL	Delete
128	80	SS1	Single-Shift 1
129	81	SS4	Single-Shift 4
130	82	SS5	Single-Shift 5
131	83	SS6	Single-Shift 6

Table C-1. VOS Internal Character Code Set *(Page 6 of 10)*

Decimal Code	Hex Code	Symbol	Name
132	84	SS7	Single-Shift 7
133	85	SS8	Single-Shift 8
134	86	SS9	Single-Shift 9
135	87	SS10	Single-Shift 10
136	88	SS11	Single-Shift 11
137	89	SS12	Single-Shift 12
138	8A	SS13	Single-Shift 13
139	8B	SS14	Single-Shift 14
140	8C	SS15	Single-Shift 15
141	8D		(Not Assigned)
142	8E	SS2	Single-Shift 2
143	8F	SS3	Single-Shift 3
144	90	LSI	Locking-Shift Introducer
145	91	WPI	Word Processing Introducer
146	92	XCI	Extended-Control Introducer
147	93	BDI	Binary-Data Introducer
148	94		(Not Assigned)
149	95		(Not Assigned)
150	96		(Not Assigned)
151	97		(Not Assigned)
152	98		(Not Assigned)
153	99		(Not Assigned)
154	9A		(Not Assigned)
155	9B		(Not Assigned)
156	9C		(Not Assigned)
157	9D		(Not Assigned)
158	9E		(Not Assigned)
159	9F		(Not Assigned)

Table C-1. VOS Internal Character Code Set *(Page 7 of 10)*

Decimal Code	Hex Code	Symbol	Name
160	A0	NBSP	No Break Space
161	A1	¡	Inverted Exclamation Mark
162	A2	¢	Cent Sign
163	A3	£	British Pound Sign
164	A4	¤	Currency Sign
165	A5	¥	Yen Sign
166	A6	¦	Broken Bar
167	A7	§	Paragraph Sign
168	A8	¨	Dieresis
169	A9	©	Copyright Sign
170	AA	ª	Feminine Ordinal Indicator
171	AB	Ç	Left-Angle Quote Mark
172	AC	¨	‘‘Not’’ Sign
173	AD	SHY	Soft Hyphen
174	AE	Æ	Registered Trademark Sign
175	AF	Ø	Macron
176	B0	°	Degree Sign, Ring Above
177	B1	±	Plus-Minus Sign
178	B2	^{2}	Superscript 2
179	B3	^{3}	Superscript 3
180	B4	´	Acute Accent
181	B5	µ	Micro Sign
182	B6	¶	Pilcrow Sign
183	B7	·	Middle Dot
184	B8	¸	Cedilla
185	B9	^{1}	Superscript 1
186	BA	º	Masculine Ordinal Indicator
187	BB	È	Right-Angle Quote Mark

Table C-1. VOS Internal Character Code Set *(Page 8 of 10)*

Decimal Code	Hex Code	Symbol	Name
188	BC	¼	One-Quarter
189	BD	½	One-Half
190	BE	¾	Three-Quarters
191	BF	¿	Inverted Question Mark
192	C0	À	A with Grave Accent
193	C1	Á	A with Acute Accent
194	C2	Â	A with Circumflex
195	C3	Ã	A with Tilde
196	C4	Ä	A with Dieresis
197	C5	Å	A with Ring Above
198	C6	Æ	Diphthong A with E
199	C7	Ç	C with Cedilla
200	C8	È	E with Grave Accent
201	C9	É	E with Acute Accent
202	CA	Ê	E with Circumflex
203	CB	Ë	E with Dieresis
204	CC	Ì	I with Grave Accent
205	CD	Í	I with Acute Accent
206	CE	Î	I with Circumflex
207	CF	Ï	I with Dieresis
208	D0	Đ	D with Stroke
209	D1	Ñ	N with Tilde
210	D2	Ò	O with Grave Accent
211	D3	Ó	O with Acute Accent
212	D4	Ô	O with Circumflex
213	D5	Õ	O with Tilde
214	D6	Ö	O with Dieresis
215	D7	×	Multiplication Sign

Table C-1. VOS Internal Character Code Set *(Page 9 of 10)*

Decimal Code	**Hex Code**	**Symbol**	**Name**
216	D8	Ø	O with Oblique Stroke
217	D9	Ù	U with Grave Accent
218	DA	Ú	U with Acute Accent
219	DB	Û	U with Circumflex
220	DC	Ü	U with Dieresis
221	DD	Ý	Y with Acute Accent
222	DE	Þ	Uppercase Thorn
223	DF	β	Sharp s
224	E0	à	a with Grave Accent
225	E1	á	a with Acute Accent
226	E2	â	a with Circumflex
227	E3	ã	a with Tilde
228	E4	ä	a with Dieresis
229	E5	å	a with Ring Above
230	E6	æ	Diphthong a with e
231	E7	ç	c with Cedilla
232	E8	è	e with Grave Accent
233	E9	é	e with Acute Accent
234	EA	ê	e with Circumflex
235	EB	ë	e with Dieresis
236	EC	ì	i with Grave Accent
237	ED	í	i with Acute Accent
238	EE	î	i with Circumflex
239	EF	ï	i with Dieresis
240	F0	ð	Lowercase Eth
241	F1	ñ	n with Tilde
242	F2	ò	o with Grave Accent
243	F3	ó	o with Acute Accent

Table C-1. VOS Internal Character Code Set *(Page 10 of 10)*

Decimal Code	**Hex Code**	**Symbol**	**Name**
244	F4	ô	o with Circumflex
245	F5	õ	o with Tilde
246	F6	ö	o with Dieresis
247	F7	÷	Division Sign
248	F8	ø	o with Oblique Stroke
249	F9	ù	u with Grave Accent
250	FA	ú	u with Acute Accent
251	FB	û	u with Circumflex
252	FC	ü	u with Dieresis
253	FD	ý	y with Acute Accent
254	FE	þ	Lowercase Thorn
255	FF	ÿ	y with Dieresis

Glossary

access

To read from or write to a file or device.

address

The location of an area of storage. An address is a 4-byte value.

aggregate types

Types that consist of an ordered collection of data objects. The aggregate types are the array and structure types.

aligned

Allocated on a particular storage boundary.

allocate

To set aside an area of storage for a particular purpose.

American National Standards Institute (ANSI)

A group that promotes standards for computer languages and devices.

American Standard Code for Information Interchange (ASCII)

A standard 7-bit character representation code that the operating system stores in an 8-bit byte.

ANSI

See **American National Standards Institute**.

arbitrary expression

An expression that can be evaluated at run time. An arbitrary expression can contain operands whose values are determined at run time.

argument

1. An expression that appears between parentheses (`()`) in a function-call expression. When a function is invoked, the value of each argument is assigned to the corresponding parameter.

2. A character string that specifies how a command is to be executed.

arithmetic types

The integral and floating-point types.

array

A set of objects of the same data type. In an array, the individual objects, or elements, are stored contiguously at increasing addresses in memory.

ASCII

1. In the VOS internal character coding system, the half of the 8-bit code page with code values in the range 00-7F (hexadecimal), representing the American Standard Code for Information Interchange.
2. See **American Standard Code for Information Interchange**.

assign

To associate a value with a variable.

automatic storage duration

In existence only for the duration of a block of code. An object with automatic storage duration is temporary. An object has automatic storage duration if the object is declared with no linkage and no explicit `static` storage-class specifier.

binary

Base 2; a base designation for arithmetic data.

binary file

A file in which a data item is stored in its internal form, as opposed to its character form. See also **text file**.

bind

To combine a set of one or more independently compiled object modules into a program module. Binding resolves symbolic references to external programs and variables that are shared by object modules in the set and in the object library. See also **library**.

bind map

A file, produced by the binder, containing information about a program module. A bind map is only produced if you specify the `-map` option of the `bind` command.

bind time

The time at which the binder is invoked to combine one or more user object modules and VOS support routines into a program module.

binder

The program that combines a set of independently compiled object modules into a program module. The binder is invoked with the `bind` command.

binder control file

A text file containing directives for the binder.

bit

The smallest unit of internal computer storage. A bit can have one of two values: 1 or 0.

bit field

A structure or union member that consists of a cluster of contiguous bits.

block

A compound statement, a set of statements grouped into one syntactic unit and delimited by braces (`{}`).

buffer

A temporary storage area for input or output data.

built-in functions

Functions that are predefined within the VOS C language.

byte

A unit of storage consisting of eight contiguous bits.

call

To activate a function, usually by means of a function-call expression.

canonical string

A character string with no default character set and no locking- shift characters. All non-ASCII characters are therefore preceded by a single-shift character. A canonical string may contain generic input or output sequences.

character

A symbol, such as a letter of the alphabet or a numeral, or a control signal, such as a carriage return or a backspace. Characters are represented in electronic media by character codes.

character constant

One or two characters or escape sequences enclosed in apostrophes. For example, `'x'` and `'\0'` are character constants. The value of a character constant is its numeric rank in the VOS internal character coding system. For ASCII characters, this value is the same as the character's ASCII code.

character-string literal

A sequence of zero or more characters or escape sequences enclosed in quotation marks. For example, `"bcd"` and `"AMOUNT = \%3.2f\n"` are character-string literals. A character-string literal is also called a string literal.

close

To disconnect from an operating system file or device. For example, the VOS C library function `fclose` closes a file.

code

1. Machine instructions generated by the compiler.

2. The contents of a source module.

code region

The portion of a standard program module that contains the actual instruction sequences that represent the program.

command

A program invoked from command level, either interactively or as a statement in a command macro.

comment

Documentary information included in source code that is ignored by the compiler; it has no effect on the execution of the program.

compile time

The time at which a compiler is invoked to translate a source module into an object module (program).

compiler

A program that translates a source module (source code) into machine code. The generated machine code is stored in an object module.

compound statement

A set of statements grouped into one syntactic unit and delimited by braces (`{}`).

concatenate

To join end-to-end.

condition

An exceptional occurrence during the execution of a program.

constant

A sequence of characters that represents a fixed numerical value. Every constant has an associated data type. Constants are grouped into the following categories: integer constants, floating-point constants, character constants, and enumeration constants.

constant expression

An expression that can be evaluated at compile time to a constant value.

control character

A character in the VOS internal character coding system that can perform a specified control function. There are two sets of control characters: ASCII and Stratus-specific. The ASCII control characters are represented by the hexadecimal character codes 00 to 1F. The Stratus-specific control characters are represented by the hexadecimal character codes 80 to 9F. Unlike graphic characters, control characters are nonprinting characters.

conversion

The process of transforming a value from one data type to another.

current directory

The directory currently associated with your process. The operating system uses your current directory as the default directory when you do not specifically name the directory containing an object that you want the operating system to find. For example, if you supply a relative path name in a command, the operating system uses the current directory as the reference point from which to locate the object in the directory hierarchy.

current position

The location in a file at which the next input or output operation will be performed.

data type

The collective attributes of a value, object, or function that determine the scheme by which the data item is stored and the operations that can be performed on the data item.

debug

To correct errors (bugs) in a program.

debugger

A VOS tool used as an aid in finding program errors.

declaration

A declaration specifies the attributes of one or more identifiers: storage class, data type, type qualifiers, alignment, scope, linkage, and storage duration. A declaration announces the properties of a data item, without fully specifying it, and does not allocate storage. Contrast with **definition**.

declarator

In a declaration, a declarator consists of an identifier and, optionally, one or more array (`[]`), function (`()`), or pointer (`*`) modifiers. In addition, with some derived types, a declarator can contain type qualifiers. Each declarator declares one identifier to have the data type and other attributes specified in the declaration.

default

The value or attribute used when a necessary value or attribute is omitted.

definition

A declaration that provides a complete description of the data item, and that also causes storage to be reserved for an object or function. Contrast with **declaration**.

delimiter

One or more white-space characters that separate two adjacent tokens. A delimiter is not part of either of the tokens it delimits.

detach

To disassociate a port from a file or device.

device

Any hardware component that can be referenced and used by the system or users of the system and that is defined in the device configuration table. Terminals, printers, tape drives, and communications lines are devices.

diagnostic

A message from the compiler warning of a possible error.

dimension

A section of an array having a certain size, an integral upper bound, and an integral lower bound.

directory

A segment of disk storage containing files, links, and subdirectories and having its own access limitations.

element

A single object in an array of objects.

entry point

A location in a program where execution can begin when the program is activated. An entry point can be specified in a binder control file.

entry value

The value of an entry point constant or the value assigned to an entry variable. An entry value consists of a display pointer, a code pointer, and a static pointer.

enumeration

An object of an enumerated type.

enumeration constant

One of a set of identifiers declared as members of a user-defined enumerated type. Variables of that `enum` type can be assigned any one of the specified enumeration constants.

error code

An arithmetic value indicating what, if any, error has occurred. An error code is a 2-byte integer, often representing a VOS status code.

escape sequence

A character sequence consisting of a reverse slant (`\`) followed by one or more characters. An escape sequence can represent a graphic or nongraphic character within a character constant or character-string literal.

execute

1. To process an executable statement at run time.
2. To run a program.

exponent

An arithmetic value representing the number of times some base number is to be multiplied by itself.

expression

A sequence of operators and operands that does one or a combination of the following: specifies the computation of a value, designates an object or function, or generates side effects.

external linkage

An identifier with external linkage is visible across multiple source modules. Each instance of a particular identifier with external linkage denotes the same object or function.

fence

A portion of a user's virtual address space adjacent to the process or a task's user stack. It allows the operating system to detect most references beyond the end of the stack without any corruption of data in adjacent regions.

file

A sequence of bytes or a set of records stored as a unit on a disk or tape. In C, a physical device such as a terminal is also considered to be a file.

file descriptor

A non-negative integer value returned from a call to the `creat` or `open` function, or to the `fileno` macro. A file descriptor specifies which file to access with the UNIX I/O functions. Contrast with **file pointer**.

file organization

The manner in which data in a VOS file is arranged. The operating system supports four file organizations: stream, sequential, fixed, and relative.

file pointer

A pointer returned by the `fdopen`, `fopen`, `freopen`, or `tmpfile` function. This file pointer specifies which file to access with many of the standard I/O functions from the `stdio.h` header file. Contrast with **file descriptor**.

fixed organization

A VOS file organization in which data are stored in records of equal length. See also **relative organization**, **sequential organization**, and **stream organization**.

floating-point constant

An approximation of a real number. A floating-point constant is specified using either a decimal fraction or exponential notation.

floating-point types

The `float` and `double` types.

full path name

For a file, directory, or link, a name that is composed of the name of the system, the name of the disk, the names of the directories that contain the object, and finally the name of the file, directory, or link.

For a device, a name that is composed of the name of the system and the name of the device.

function

A subprogram invoked during the evaluation of a function-call expression that designates the function. A function takes zero or more arguments as input values, and returns a single value to the point of invocation. In addition to returning a value, functions often produce side effects, such as modifying data defined outside the function.

function prototype

A function declaration or definition that serves as a model specifying the number and types of a function's parameters. In the source module where the prototype is specified, the compiler uses the prototype's parameter information to check all subsequent references to the function.

header file

A file that the compiler incorporates into the source module. The name of the header file is specified after the `#include` directive in a preprocessor control line. A header file is sometimes called an include file.

hexadecimal

Base 16; a base designation for arithmetic data.

I/O

Input and output.

identifier

A sequence of 1 to 2048 alphabetic characters, digits, underline characters (_) and dollar sign characters (`$`). The first character of an identifier must be a letter, underline character, or dollar sign character. An identifier denotes one of the following elements of a C program: an object; a function; a tag or a member of a structure, union, or enumeration; a `typedef` name; a label name; or a macro name or parameter.

implicit locking

A VOS locking mode in which the operating system does not lock the file or record for either reading or writing when it opens the file, but rather locks it for the appropriate access type each time a process performs an I/O operation on the file or record.

include file

See **header file**.

include library

One or more directories that the operating system searches for include files.

indirection

The unary `*` operator results in indirection. The operand of the `*` operator must yield the address of an object, which the indirection operator references.

initialize

To assign a value to a data object at the start of its lifetime.

integer

A whole number; an arithmetic value with no fractional part.

integral types

The integral types are as follows: `char`, the signed and unsigned integer types, and the enumerated types.

internal linkage

An identifier with internal linkage is visible within only one source module. Within one source module, each instance of an identifier with internal linkage denotes the same object or function.

invocation

The activation of a function, usually by means of a function-call expression.

keyword

1. A word that has special meaning to the VOS C compiler. For example, keywords identify data types, storage classes, statements, and alignment specifiers. You cannot use keywords as identifiers.

2. For VOS commands or requests, an argument label that begins with a hyphen.

label

See **statement label**.

left control character

An ASCII control character. See also **control character**.

left graphic character

A character located in the range 21x to 7Ex in the VOS internal character coding system. Left graphic characters compose the ASCII character set.

library

1. One or more directories in which the operating system looks for objects of a particular type. There are four types of libraries defined by the operating system:

   - include libraries, in which the compilers search for include files
   - object libraries, in which the binder searches for object modules
   - command libraries, in which the command processor searches for commands
   - message libraries, in which the operating system searches for message files associated with individual `.pm` files.

   One of each of these libraries is available in the `>system` directory of each module for all processes running on the module. In addition, you can define your own libraries.

link

1. An object in a directory that directs all references to itself to a file, directory, or another link. Like many other objects, a link has a path name that identifies it as a unique entity in the system directory hierarchy.

2. See also **bind**.

linkage

A characteristic that determines how references to objects or functions can be matched across blocks of code and among source modules, libraries, and object modules.

lock

1. A system data structure associated with a file, file record, file region, or device that can be set to restrict the use of the object; the restriction remains in effect until the lock is reset.

2. To set a lock on a file, record, file region, or device.

locking-shift character

A user-transparent character in an array or string, indicating that the remaining characters in the key or record are from a character set other than the default character set.

lvalue

An expression that designates a region of storage (object). For example, the name of a variable is an lvalue designating a particular region of storage.

macro

The identifier immediately following the `#define` directive in a preprocessor control line. The two types of macro definitions are: object-like macros and function-like macros.

member

An object that is part of a structure or union. A member can have its own name and a distinct type.

module

A single Stratus computer. A module is the smallest hardware unit of a system capable of executing a user's process.

National Language Support (NLS)

The ability of the operating system to represent text in languages other than English. A system of supporting different languages, character sets, date and time formats, and currency formats.

newline character

The linefeed character (ASCII code 10 decimal). The escape sequence for a newline character is `\n`. When a newline character is written to the terminal's screen, it moves the active position of the cursor to the beginning of the next line.

NLS

See **National Language Support**.

NLS string

A general term for any of the string formats, combined or used alone; therefore, a string containing characters from one or more NLS character sets with, optionally, locking- or single-shift characters.

null pointer

A pointer that is equal to `NULL`. A null pointer does not contain the address of a valid storage location.

null pointer constant

An integer constant expression equal to the value 0, or such an expression cast to the type `void *`. The `NULL` constant defines the null pointer constant. Setting a pointer to `NULL` indicates that the pointer does not currently locate any object or function.

null string

A varying-length character string with a current length of 0.

null-terminated string

See **string**.

object

1. A region of storage, the contents of which can represent values.
2. Any data structure or device in the system that you can refer to by a name or some other identifier. For example, all of the following are objects: directories, files, links, systems, modules, devices, groups, persons, ports, queues, locks, file indexes, and file records.

object library

One or more directories that the operating system searches for object modules.

object module

A file produced by a compiler that contains the machine-code version of one or more procedures; it usually contains symbolic references to external variables and programs. To execute the program, an object module must be processed by the binder to produce a program module, and then loaded by the loader.

open

To prepare a file or device for a particular type of access. For example, the VOS C library function `fopen` opens a file.

operand

A subexpression on which an operator acts.

operator

A symbol in an expression that performs an operation (evaluation) on one or more operands. The result of the operation specifies the computation of a value, designates an object or function, generates side effects, or produces a combination of these actions.

optimization

A code-improving modification that makes a program run faster, take less space, or both.

optimizer

Optionally invoked as part of the compilation process, an optimizer modifies executable code so that it runs faster, takes less space, or both.

organization

See **file organization**.

parameter

An object declared as part of a function definition that acquires a value when the function is called. Parameter types are specified in a function declaration that includes a prototype.

parameter type list

When a function type is specified, a list of one or more parameter declarations or parameter types. In a function definition, the parameter type list consists of parameter declarations. In a function declaration, the parameter type list consists of the types of the parameters with or without identifiers.

path name

A unique name that identifies a device or locates an object in the directory hierarchy. See also **full path name** and **relative path name**.

pointer

The address of a storage location that contains an object of a particular type. In VOS C, pointers of all types are four bytes long, have the same format, and store an unsigned address.

pointer constant

A pointer that is a nonmodifiable lvalue. For example, an array name is a pointer constant.

pointer variable

A pointer that is a modifiable lvalue. For example, an object declared as `int *` is a pointer variable.

port ID

A 2-byte integer used to identify a port.

port name

The character-string name of a port.

portable

1. Capable of being moved, without modification, from one machine type and/or operating system to another.
2. Machine or operating-system independent.

precision

The total number of significant digits in an arithmetic value.

preprocessor

A program that prepares a file for another program. For example, the C preprocessor processes a source module before the compiler translates the source module into object code.

preprocessor directive

The first token that appears after `#` on a preprocessor control line. Preprocessor directives specify an action that the preprocessor is to perform.

process

The sequence of states of the hardware and software during the execution of a user's programs. When you log in, the operating system creates a process for you to control the execution of your programs. Your process can create other processes at your request. A process is always in one of three states: running, waiting, or ready.

program

One or more functions, from one or more source modules, that together perform a task.

program entry

See **entry point**.

program module

A file containing an executable form of a program. The program module consists of one or more object modules (compiled source programs) bound together. The program module always has the suffix `.pm`.

program name

The full or relative path name that identifies a program module, optionally omitting the `.pm` suffix.

prototype

See **function prototype**.

punctuator

A token that has special syntactic and semantic significance to the compiler but does not specify an operation that yields a value. Typically, a punctuator indicates how an entity will be used, or it delimits an identifier or block of code.

raw I/O

In the operating system, the transfer of input and output characters between the user's buffer space and a device without any translation. The system subroutines `s$read_raw` and `s$write_raw` perform raw I/O.

read lock

A lock that allows other tasks or processes to set a read lock on a given object but prevents them from setting a write lock. It allows a reader to ensure that an object will not be modified while it is being read.

record

The data structure that the operating system uses to manage data in a file. For files other than stream files, a record is the smallest unit of data that the operating system I/O routines can access when performing I/O operations on files or devices. For stream files, a record consists of unstructured data.

relative organization

A VOS file organization in which data are stored as varying-length records in fixed-length fields. See also **fixed organization**, **sequential organization**, and **stream organization**.

relative path name

A name that identifies a device or an object in the directory hierarchy without specifying its full path name.

return

To terminate a function and transfer program control back to the calling function.

return value

The value returned from an invoked function.

right control character

A control character with a hexadecimal character code between 80 and 9F. See also **control character**.

right graphic character

A character located in the range A0x to FFx in the VOS internal character coding system. Right graphic characters compose the Latin alphabet No. 1, katakana, kanji, and hangul character sets.

run

To execute a program.

run time

The time at which a program module is invoked and executed.

scalar

A single, one-dimensional value or object; not an array or structure, though possibly an array element or structure member.

scalar types

Types that hold a single data object and yield a single value. The scalar types consist of the arithmetic types, pointer types, and the varying-length character string type.

scope

The region of a program's source text in which an identifier is visible (that is, can be used). An identifier can have file, block, function, or function prototype scope.

search list

A series of operating system directories to be searched for include files, object modules, or command macros and program modules.

sequential organization

A VOS file organization in which data are stored in varying-length records; each record is preceded and followed by two bytes representing its length. See also **fixed organization**, **relative organization**, and **stream organization**.

shared variable

An external variable that can be shared simultaneously by several object modules. Shared variables are allocated in the virtual address space of one or more processes.

sign

An indication of whether an arithmetic value is less than or greater than zero. A sign can be positive or negative.

signal

An asynchronous interrupt for an error condition, or exception, that may be reported during program execution.

source module

A text file (single source program) containing language statements, compile-time statements, and comments that can be compiled to produce an object module.

stack

An area of storage consisting of an ordered series of stack frames associated with the execution of a program.

stack frame

An area of storage on the stack associated with an activation of a function.

star name

A name that contains one or more asterisks or consists solely of an asterisk. A star name can be used to specify a set of objects.

statement

One of several programming constructs that specifies an action or actions to be executed by a program.

statement label

An identifier that is followed by a colon (`:`) and a statement. A labeled statement can be referenced in a `goto` statement.

static region

The region of an object module that contains external references and internal static data.

static storage duration

In existence throughout the execution of the entire program. An object with static storage duration is permanent, retaining its value even when the program is executing outside the object's scope.

storage class

A characteristic of an identifier indicating the linkage of an object or function and the storage duration of an object. VOS C supports six storage classes: `auto`, `extern`, `ext_shared`, parameter, `register`, and `static`.

storage duration

The amount of time an object exists. There are two storage durations: static and automatic.

stream organization

A VOS file organization in which stream files with varying length records are stored in a disk or tape region holding approximately the same number of bytes as the record. The record storage regions vary from record to record, and may be accessed on a record or byte basis.

When stream files are used to store text, each record contains one line of text. See also **fixed organization**, **relative organization**, and **sequential organization**.

string

An array of type `char` that contains a contiguous sequence of characters terminated by and including the first null character (`\0`).

structure

A sequentially allocated set of objects, grouped under a single name. Within the structure, each object or member can have its own name and a distinct type.

subroutine

A sequence of statements that can be invoked as a set at one or more points in a program to execute a specific operation.

subscript

An arithmetic value used to specify a particular element or elements of an array.

suffix

A character string that begins with a period and is appended to an object name to indicate the type of the object.

symbol table

A construct that the compiler creates in the symtab region of an object module to facilitate symbolic debugging. The symbol table allows the debugger to convert user-defined variable names to locations of data or instructions. The compiler creates a symbol table only if the `-table` or `-production_table` argument of a compile command is specified.

tag

An optional name that follows the `enum`, `struct`, or `union` specifier, and that can be used with any one of these keywords to specify an object.

text file

In general usage, a file of characters. A text file can contain graphic characters and control characters from the Stratus internal character set. See also **binary file**.

token

The minimal lexical unit of the C language. Tokens are categorized into six classes: keywords, identifiers, operators, punctuators, constants, and character-string literals.

union

An overlapping set of member objects, grouped under a single name. Each object in a union can have its own name and a distinct type.

value

A measurable, describable, storable quantity that is associated with a constant, variable, or expression.

variable

A declared object that can be assigned a value.

varying-length character string

A character string that can have any length from 0 to 32,766 characters.

VOS internal character coding system

The system used internally for encoding character data on Stratus systems. This system, based on the international standard ISO-2022-1986, allows encoding of the multiple character sets needed for National Language Support.

VOS operating system

The virtual operating system of a Stratus computer.

white space

One or more of the following characters: space, horizontal tab, vertical tab, form feed, carriage return, and newline. Within a source module, white-space characters are used as separators between adjacent tokens, or within character constants or character-string literals. Otherwise, the compiler ignores these characters.

write lock

A lock that prevents all other tasks and processes from setting either a read or a write lock on a given object. It allows a writer to ensure that an object will not be read while being modified and that multiple tasks or processes are not simultaneously modifying the same object.

XA2000-series module

A Stratus module that uses processors from the MC68000 family of processors.

XA/R-series module

A Stratus module that uses processors from the i860 family of processors.

Index