Introduction to C Programming

Seneca

Table of contents:

- Structures
  - Learning Outcomes
  - Introduction
  - Types
    - Primitive Types
    - Derived Types
    - Declaration
    - Allocating Memory
    - Initialization
  - Member Access
  - Walkthrough
- Functions
  - Learning Outcomes
  - Introduction
  - Modular Design
    - Design Principles
    - Cohesion
    - Coupling
  - Functions
    - Definition
    - Special Cases
    - Function Calls
    - Pass By Value
    - Mixing Types
  - Walkthroughs
  - Validation (optional)
  - Walkthrough
- Pointers
  - Learning Outcomes
  - Introduction
  - Addresses
    - Pointer
    - Pointer Types
    - NULL Address
  - Parameters
    - Pass-by-Value
    - Pass-by-Address
  - Multiple Return Values
- Functions, Arrays and Structs
  - Learning Outcomes
  - Introduction
  - Prototypes
    - include
    - Current Directory(Optional)
  - Scope
    - Global Scope
    - Function Scope
    - Local Scope
    - Block Scope
    - Overlapping Scope (Optional)
  - Passing Arrays
    - Array Arguments
    - Parameters
    - Barring Changes
  - Passing Structures
    - Copying
    - Efficiency

On this page

Table of contents

Appendices

Computers

Learning Outcomes

After reading this section, you will be able to:

- Describe the major components of a modern computer
- Describe the software that controls a modern computer
- Outline the contents of these notes

Introduction

Computers are available in many flavours: mobile devices, smart phones, laptops, tablets, desktops, workstations and servers to name a few. All of these devices control their operations through software. Programmers create this software. Users rely on this software to operate their devices.

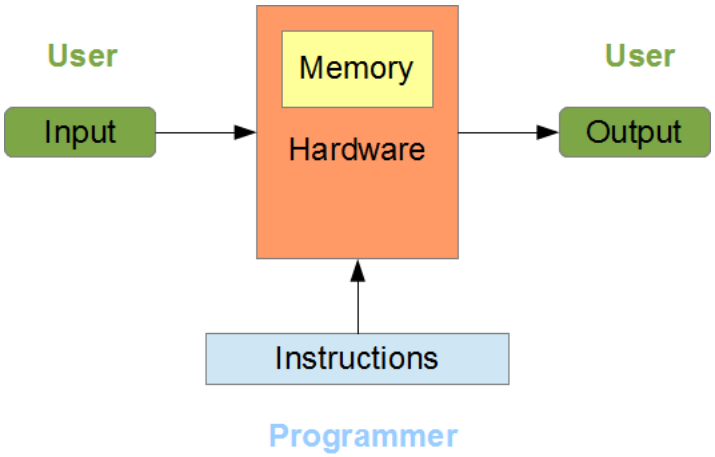
We refer to the software with which a user operates their device as application software. Application software consists of one or more programs. Each program is a full set of instructions that performs a well-defined task on the host device. Programmers code these instructions in a programming language.

The instructions that programmers code represent algorithms. An algorithm is a step-by-step procedure that describes how to achieve a specified task. Examples include searching, sorting and mixing. Application programmers study different algorithms and create their own if required. They find the code for some algorithms in libraries and write the code for their own algorithms.

This introductory chapter describes the major components of a modern computer, components to which programmers often refer. Subsequent chapters show how to write programs to use the principal features of these components efficiently.

Hardware

The figure below illustrates the relation of the programmer to a user of a software application. The boxes identify the principal elements for any programmer. The green boxes identify the elements of concern to the user.

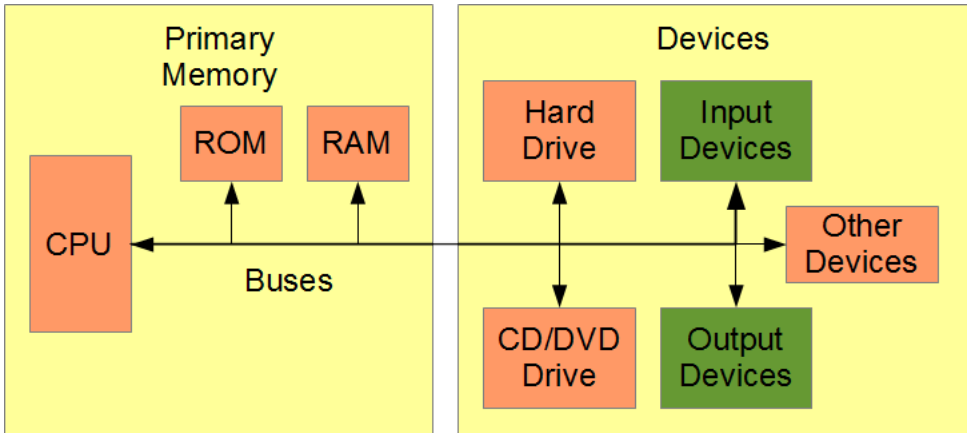


Computer hardware stores the program instructions in its own memory, accepts input from the user, processes that input according to the stored instructions, and generates the output for the user. The user can rerun the program to process different inputs and produce corresponding outputs.

Modern Computers

In 1945, John von Neumann, noting that instructions are pieces of information just like data, proposed a new computer architecture, in which instructions and data are stored alongside one another. We call this idea the stored-program concept. All modern computers are stored-program computers.

The figure below shows the components of a stored-program computer. They include a central processing unit (CPU), a clock, primary memory and a set of devices. Buses interconnect these components and are part of the motherboard. The CPU, primary memory and a clock are also part of the motherboard. The clock controls the rate at which the CPU executes the instructions.



Primary Memory

Primary memory is memory directly accessible by the CPU. Primary memory includes read-only memory (ROM), random-access memory (RAM) and memory within the CPU itself.

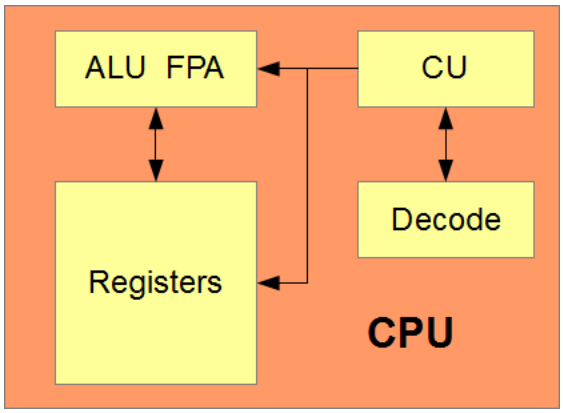
ROM holds the instructions for starting the system. ROM is not volatile: it persists if we turn off power.

RAM holds the program instructions and the program data. RAM is volatile: its contents are lost if we turn off power.

Central Processing Unit

The CPU is the work-horse of any modern computer. The CPU executes program instructions serially (one at a time). A modern CPU consists of:

- registers
- a decode unit
- a control unit (CU)
- an arithmetic and logic unit (ALU)
- a floating-point accelerator (FPA)



Registers are the CPU's internal memory. They hold the data used by the ALU and FPA and any new data that the ALU and FPA produce. Register data is volatile: we lose the contents of each register as soon as we turn off power.

The Decode unit extracts each incoming instruction from the instruction queue and decodes that instruction. The CU moves data between registers, RAM and the devices, and passes the decoded instruction to the ALU or the FPA for processing. The CU manages the data, but does not change it.

The ALU performs the comparisons and integer calculations, changes the data and creates new data as directed by the CU. The FPA performs the calculations on floating-point data. The ALU and FPA work solely with register memory inside the CPU.

Devices

The devices of a modern computer include peripheral and other devices. Peripheral devices include a keyboard, a mouse, and a monitor, which provide user interfaces for input and output. Hard drives, USB keys and DVD/CD-ROMs constitute secondary memory, which provides persistent storage of program instructions and program data.

Memory Comparison

Secondary memory is inexpensive compared to primary memory, but considerably slower. Compare the following data transfer rates:

- Registers ~10 nanoseconds
- ROM and RAM ~60 nanoseconds
- Hard disk ~12,000,000 nanoseconds

A nanosecond is 1e-9 seconds. To appreciate the differences, consider the following analogy. The ratio of the time that the CPU takes to transfer data between registers to the time that a hard disk takes to transfer that same information is the ratio of the width of an average-sized room to the distance once around the earth along the equator.

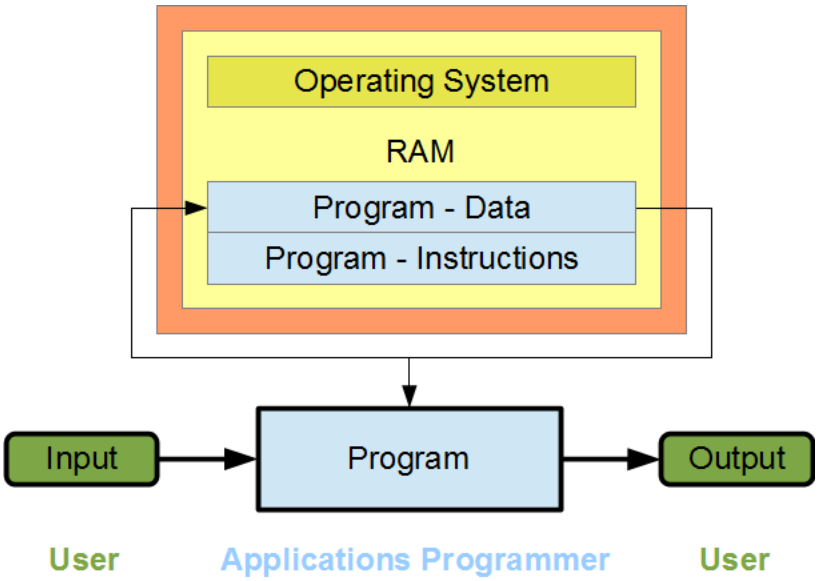
Software

The software that controls a modern computer includes the programs that are currently executing and the operating system that manages them. The operating system is a program that executes as long as power is on. The operating system resides in RAM along with the other currently executing programs.

When a user starts an application program, the operating system loads that program's instructions into part of the RAM and transfers control to the program. The program starts executing, requests data from the user, sends output to the user, and eventually terminates its own execution and returns control to the operating system.

An application program transforms raw data from the user (the input) into equivalent data stored in RAM, operates on the data in RAM and transforms the resultant data into some user-readable form (the output). Programming languages define how internal data is stored in RAM.

As application developers, we focus on input and output processing and transformation of input data into output data. The figure below relates this focus to the software that we code.



Outline

These notes introduce the fundamental concepts of software development. The chapters are grouped into six parts:

1. Introduction

2. Computations
3. Data Structures
4. Modularity
5. Secondary Storage
6. Refinements

The introductory part describes modern computers, the storage of information in memory, and how to create your first program.

The computations part introduces the concept of type, shows how to describe program variables using types, shows how to handle basic input and output, how to calculate a new value from existing values, how to create optional paths through a program, how to write code in a friendly readable style, and how to test and debug sets of program instructions, including how to determine the output produced by those instructions.

The data structures part describes how to organize groups of values into data types in memory. This part introduces the grouping concepts of arrays and structures.

The modules part describes how to organize program instructions into self-contained cohesive units, called functions, where each function implements a single algorithm. This part shows how to divide a program into independent modules and how to pass information from one module to another.

The secondary storage part describes how to move text data between an installed device and RAM. This part introduces files and describes the syntax for working with secondary memory.

The refinements part elaborates on concepts introduced in the other parts. It covers character strings as specialized versions of arrays and shows how to work with the library functions that handle them. This part describes the relation between pointers and arrays, including arrays of structures. This part also covers two-dimensional arrays and shows how to store tabular data using strings. This part concludes with introductions to some of the standard algorithms and the guidelines for portability of program code.

Optional Sections

Some chapters include sections or sub-sections marked optional. These sections contain information that elaborates specific details related to the topic covered in these notes. Feel free to skip these sections on first reading, without disrupting presentation flow. Subsequent mandatory sections do not rely on any information covered in these optional sections. These optional sections simply add depth to the material.

On this page

Information

Learning Outcomes

After reading this section, you will be able to:

- Define the units for storing information on a modern computer
- Introduce the memory model for programming a modern computer
- Introduce the addressing system for accessing the memory of a modern computer

Introduction

The information stored in a computer includes program instructions and program data. This information is stored in bits in RAM. The instructions and data take the form of groups of bits. The two most common systems for interpreting information stored in RAM are the binary and hexadecimal numbering systems.

This chapter defines these numbering systems and their units and describes the memory model for addressing different groups of bits stored in the part of RAM associated with a program.

Fundamental Units

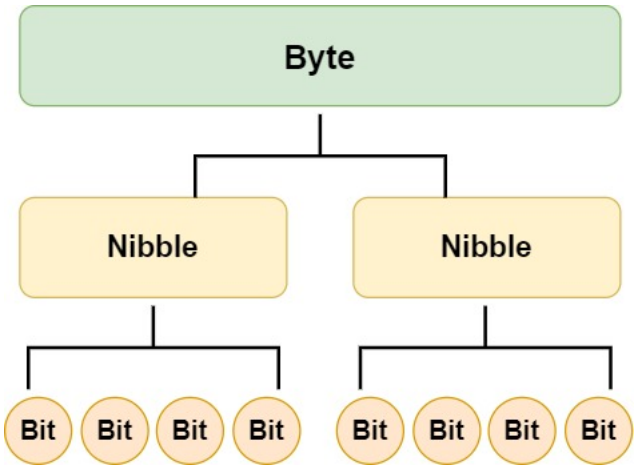
Bits

The most fundamental unit of a modern computer is the binary digit or bit. A bit is either on or off. One (1) represents on, while zero (0) represents off.

Since bits are too numerous to handle individually, modern computers transfer and handle information in larger units. As programmers, we define some of those units.

Bytes

The fundamental addressable unit of RAM is the byte. One byte consists of 2 nibbles. Each nibble consists of 4 bits.



One byte can store any one of 256 (2^8) possible values in the form of a bit string:

Bit Value	Decimal Value
00000000	0
00000001	1

Bit Value	Decimal Value
00000010	2
00000011	3
00000100	4
...	...
00111000	56
...	...
11111111	255

The bit strings are on the left. The equivalent decimal values are on the right. Note that our counting system starts from 0, not from 1.

Words

We call the natural size of the execution environment a word. A word consists of an integral number of bytes and is typically the size of the CPU's general registers. Word size may vary from CPU to CPU. On a 16-bit CPU, a word consists of 2 bytes. On a Pentium 4 CPU, the general registers contain 32 bits and a word consists of 4 bytes. On an Itanium 2 CPU, the general registers contain 64 bits, but a word still consists of 4 bytes.

Hexadecimal

The decimal system is not the most convenient numbering system for organizing information. The hexadecimal system (base 16) is much more convenient.

Two hexadecimal digits holds the information stored in one byte. Each digit holds 4 bits of information. The digit symbols in the hexadecimal number system are {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. The characters A through F denote the values that correspond to the decimal values 10 through 15 respectively. We use the **0x** prefix to identify a number as hexadecimal (rather than decimal - base 10).

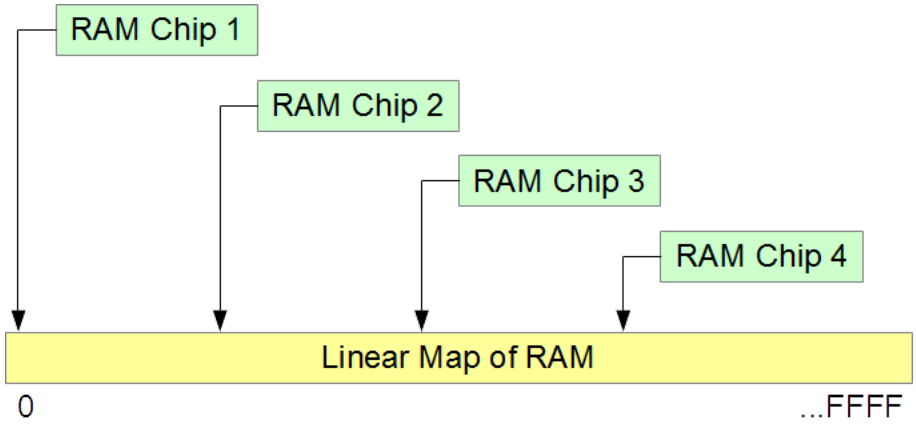
Bit Value	Hexadecimal Value	Decimal Value
00000000	0x00	0
00000001	0x01	1
00000010	0x02	2
00000011	0x03	3
00000100	0x04	4
...	...	
00111000	0x38	56
...	...	
11111111	0xFF	255

For example, the hexadecimal value 0x5C is equivalent to the 8-bit value 01011100_2, which is equivalent to the decimal value 92.

To learn how to convert between hexadecimal and binary refer to the chapter entitled Data Conversions in the Appendices.

Memory Model

The memory model for organizing information stored in RAM is linear. Any byte in memory is accessible through a map that treats each actual physical memory location as a position in a continuous sequence of locations aligned next to one another.

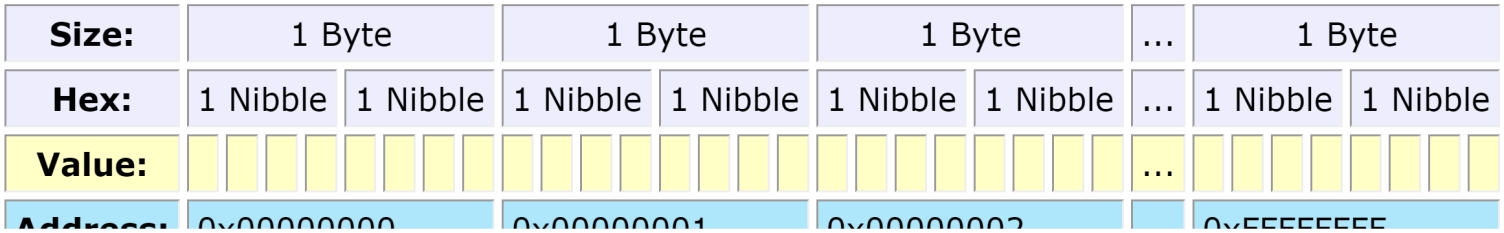


Addresses

Each byte of RAM has a unique address. Addressing starts at zero, is sequential, and ends at the address equal to the size of RAM less 1 unit.

For example, 4 Gigabytes of RAM

- consists of 32 (= 4 * 8) Gigabits
- starts at a low address of `0x00000000`
- ends at a high address of `0xFFFFFFFF`



> **NOTE**
>
> Each byte, and not each bit, has its own address. We say that RAM is byte-addressable.

Sets of Bytes

The abbreviations for sets of bytes are:

- Kilo or k (=1024): 1 Kilobyte = 1024 bytes ~ 10^3 bytes
- Mega or M (=1024k): 1 Megabyte = 1024 * 1024 bytes ~ 10^6 bytes
- Giga or G (=1024M): 1 Gigabyte = 1024 * 1024 * 1024 bytes ~ 10^9 bytes
- Tera or T (=1024G): 1 Terabyte = 1024 * 1024 * 1024 * 1024 bytes ~ 10^{12} bytes
- Peta or P (=1024T): 1 Petabyte = 1024 * 1024 * 1024 * 1024 * 1024 bytes ~ 10^{15} bytes
- Exa or E (=1024P): 1 Exabyte = 1024 * 1024 * 1024 * 1024 * 1024 * 1024 bytes ~ 10^{18} bytes

> **NOTE**
>
> The multiplying factor is 1024, not 1000. 1024 bytes is 2^{10} bytes, which is approximately 10^3 bytes.

Limit on Addressability (Optional)

The maximum size of the memory that the CPU can access depends on the size of its address registers. The highest accessible address is the largest address that an address register can hold:

- 32-bit address registers can address up to 4 GB (Gigabytes) (addresses can range from 0 to 2^{32}-1, that is 0 to 4,294,967,295).
- 36-bit address registers can address up to 64 GB (Gigabytes) (addresses can range from 0 to 2^{36}-1, that is 0 to 68,719,476,735).
- 64-bit address registers can address up to 16 EB (Exabytes) (addresses can range from 0 to 2^{64}-1, that is 0 to 18,446,744,073,709,551,615).

Segmentation Faults

The information stored in RAM consists of information that serves different purposes. We expect to read and write data, but not to execute it. We expect to execute program instructions but not to write them. So, certain architectures assign the data read and write permissions, while assigning instructions read and execute permissions. Such permission system helps trap errors while a program is executing. An attempt to execute data or to overwrite an instruction reports an error. Clearly, the access has been to the wrong segment. We call such errors a segmentation faults.

Compilers

Learning Outcomes

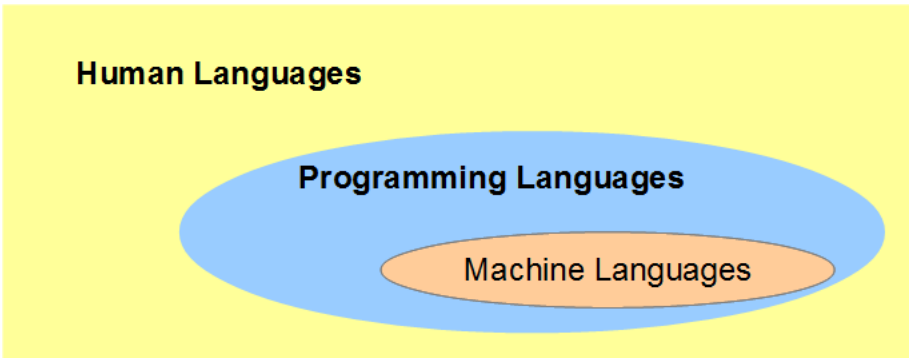
After reading this section, you will be able to:

- Describe the generations of programming languages
- Highlight the features of the C language
- Describe the general compilation process of a C compiler
- Edit, compile and run a computer program written in the C language
- Self-document a computer program using comments

Introduction

We transform the program instructions and program data into the bits and bytes that a computer understands using a compiler. We write the instructions and provide the data in a programming language that the compiler understands.

Programming languages demand completeness and greater precision than human languages. The ultimate interpreter of any computer program is the hardware. Hardware cannot interpret intent or nuance, and we need to provide much more detail in our instructions than we do in casual conversations amongst ourselves. In this sense, programming is a more arduous task than writing a formal report that someone else will read.



This chapter describes the generations of programming languages, identifies some key features of the C language, describes the compilers that we use to convert programs written in C into binary instructions that hardware can execute and explains the basic syntax found in any C program.

Programming Languages

Generations

There are well over 2500 programming languages and their number continues to increase. The different generations of programming languages include:

1. **Machine languages**. These are the native languages that the CPU processes. Each manufacturer of a CPU provides the instruction set for its CPU.
2. **Assembly languages**. These are readable versions of the native machine languages. Assembly languages simplify coding considerably. Each manufacturer provides the assembly language for its CPU.
3. **Third-generation languages**. These languages are procedural: they identify the instructions or procedures involved in reaching a result. The instructions are NOT tied to any particular machine. Examples include C, C++ and Java.
4. **Fourth-generation languages**. These languages describe what is to be done without specifying how it is to be done. These instructions are NOT tied to any particular machine. Examples include SQL, Prolog, and Matlab.
5. **Fifth-generation languages**. We use these languages for artificial intelligence, fuzzy sets, and neural networks.

The third, fourth and fifth generation languages are high-level languages. They exhibit no direct connection to any machine language. Their instructions are more human-like and less machine-like. A program written in a high-level language is relatively easy to read and relatively easy to port across different platforms.

> **NOTE**
>
> - Eric Levenez maintains an up-to-date map of 50 of the more popular languages.
> - TIOBE Software tracks the most popular languages and the long-term trends based on world-wide availability of software engineers, courses and third-party vendors as calculated from Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu search engines.

Features of C

C is one of the more popular third-generation languages. As a procedural language, C requires systematically ordered sets of instructions to perform a computational task and supports the collection of sets of instructions into so-called functions, which can be accessed from multiple points in the same program as required.

C serves as an excellent, first programming language for several reasons:

- C is English-like
- C is quite compact - has a small number of keywords
- C is the lowest in level of the high-level languages
- C can be faster and more powerful than other high-level languages
- C programs that need to be maintained are large in number
- C is used extensively in high-performance computing
- UNIX, Linux and Windows operating systems are written in C and C++

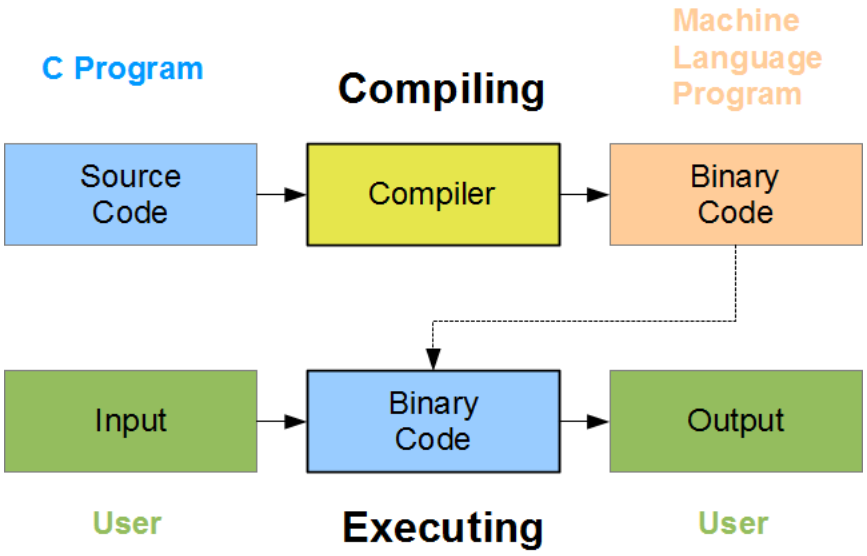
C programs execute quickly. Comparative times for a standard test (Sieve of Eratosthenes) are:

Language	Time to Run
Assembler	0.18 seconds
C	2.7 seconds
Basic	10 seconds

The C Compiler

A C compiler is an operating system program that converts C language statements into machine language equivalents. A compiler takes a set of program instructions as input and outputs a set of machine language instructions. We call the input to the compiler our source code and the output from the compiler the binary code. Once we compile our program, we do not need to recompile it, unless we have changed the source code in the interim.

To run our program, we direct the operating system to load the binary code into RAM and start executing that code. The user of our program provides input while that code is executing and receives output from it.



Compilers can optimize the program instructions that we provide to improve execution times.

Examples

Let us write a program that displays the phrase "This is C" and name our source file `hello.c`. Source files written in the C language end with the extension `.c`.

Copy and paste the following statements into a txt editor of your choice:

```
/* My first program           // comments introducing the source file
    hello.c          */

 #include <stdio.h>           // information about the printf identifier

 int main(void)               // the starting point of the program
 {
    printf("This is C"); // send output to the screen

    return 0;            // return control to the operating system
 }
```

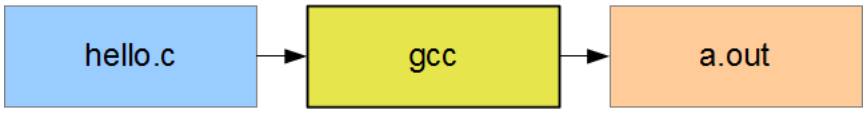
Each line is explained in more detail in the following section.

Linux

The C compiler that ships with Linux operating systems is called gcc.

To create a binary code version of our source code, enter the following command:

```
gcc hello.c
```



By default, the gcc compiler produces an output file named `a.out`. `a.out` contains all of the machine language instructions needed to execute the program.

To execute these machine language instructions, enter the command:

```
a.out
```

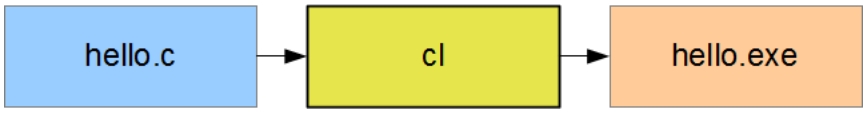
The output of the executed binary will display:

```
This is C
```

Windows

The C compiler that runs on Windows platforms is called cl. To access this compiler, open a Developer command prompt for Visual Studio window. To create a binary code version of our source code, enter the following command:

```
cl hello.c
```



By default, the cl compiler produces a file named `hello.exe`. `hello.exe` contains all of the machine language instructions needed to execute the program.

To execute these machine language instructions, enter the command:

```
hello
```

The output of the executed binary will display:

```
This is C
```

Basic C Syntax

The source code listed above includes syntax found in the source code for nearly every C program.

Documentation

The comments self-document our source code and enhance its readability. Comments are important in the writing of any program. C supports two styles: multi-line and inline. C compilers ignore all comments.

Multi-Line Comments

```
/* My first program
    hello.c             */
```

`/*` and `*/` delimit comments that may extend over several lines. Within these delimiters, we may include any character, except the `/*` and `*/` pair.

Inline Comments

```
int main(void)              // the starting point of the program
```

`//` indicates that the following characters until the end of the line.

Whitespace

Whitespace enhances program readability, for instance, by displaying the structure of a program. C compilers ignore whitespace altogether. Whitespace refers to any of the following:

- blank space

- newline
- horizontal tab
- vertical tab
- form feed
- comments

We may introduce whitespace anywhere except within an identifier or a pair of double-quotes. In the sample above, `main` and `printf` are identifiers. The blank spaces within "This is C" are not whitespace but characters within the literal string.

We preface our source code with comments to identify the program and provide distinguishing information.

Indentation

By indenting the **`printf("This is C")`** statement and placing it on a separate line, we show that **`printf("This is C")`** is part of something larger, which we have called **`int main(void)`**

Program Startup

Every C program includes a clause like `int main(void)`. Program execution starts at this line. We call it the program's entry point. The pair of braces that follow this clause encompasses the program instructions.

```
int main(void)           // program startup
{
        return 0;        // return to operating system
}
```

When the users or we load the executable code into RAM (`a.out` or `hello.exe`), the operating system transfers control to this entry point. The last statement (`return 0;`) before the closing brace transfers control back to the operating system.

Program Output

The following statement outputs "This is C" to the standard output device (for example, the screen).

```
printf("This is C");
```

The line before **`int main(void)`** includes information that tells the compiler that **`printf`** is a valid identifier.

```
#include <stdio.h>          // information about the printf identifier
```

Case Sensitivity

The C programming language is case sensitive. That is, the compiler treats the character 'A' as different from the character 'a'. If we change the identifier **`printf()`** to **`PRINTF()`** and recompile, the compiler will report a syntax error. However, if we change `"This is C"` to `"THIS IS C"` and recompile the source code, the compiler will accept the change and not report any error.

On this page

Types

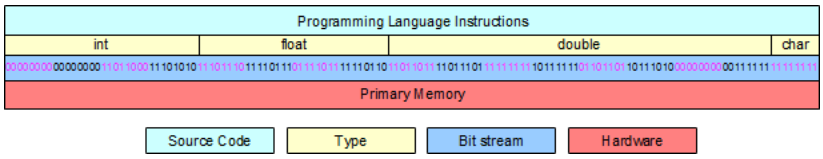
Learning Outcomes

After reading this section, you will be able to:

- Select appropriate types for storing program variables and constants
- Describe the internal representations defined by different types

Introduction

A typed programming language uses a type system to interpret the bit streams in memory. C is a typed programming language. A type is the rule that defines how to store values in memory and which operations are admissible on those values. A type defines the number of bytes available for storing values and hence the range of possible values. We use different types to store different information. The relation between types and raw memory is illustrated in the figure below.



This chapter describes the four most common types in the C language and the ranges of values that these types allow. This chapter concludes by describing how to allocate memory for variables by identifying their contents using a type.

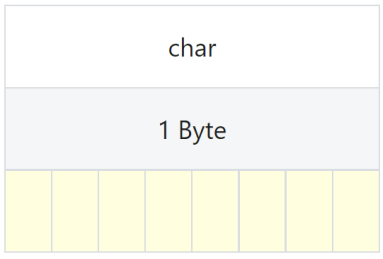
This chapter describes the four most common types in the C language and the ranges of values that these types allow. This chapter concludes by describing how to allocate memory for variables by identifying their contents using a type.

Arithmetic Types

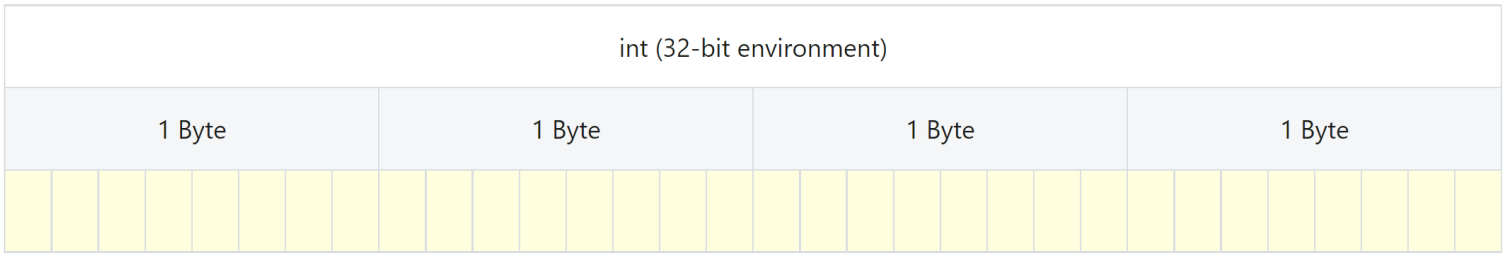
The four most common types of the C language for performing arithmetic calculations are:

- `char`
- `int`
- `float`
- `double`

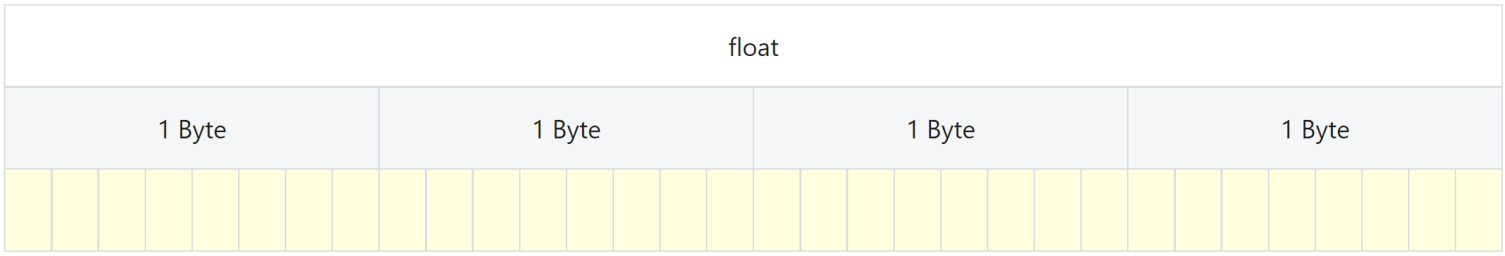
A **`char`** occupies one byte and can store a small integer value, a single character or a single symbol:



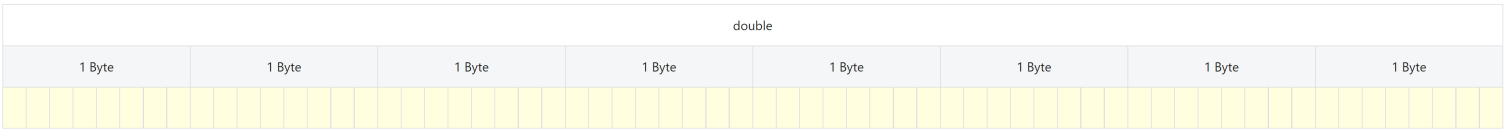
An **`int`** occupies one word and can store an integer value. In a 32-bit environment, an **`int`** occupies 4 bytes:



A `float` typically occupies 4 bytes and can store a single-precision, floating-point number:



A `double` typically occupies 8 bytes and can store a double-precision, floating-point number:



Size Specifiers

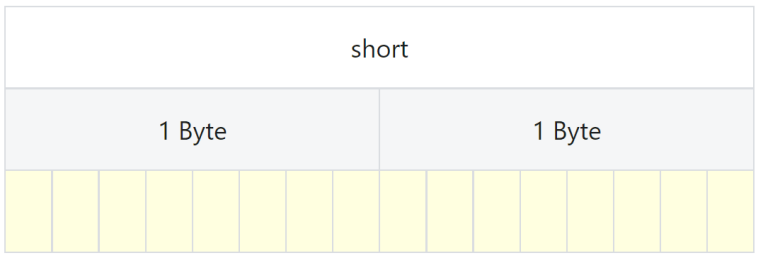
Size specifiers adjust the size of the `int` and `double` types.

`int` Type Size Specifiers

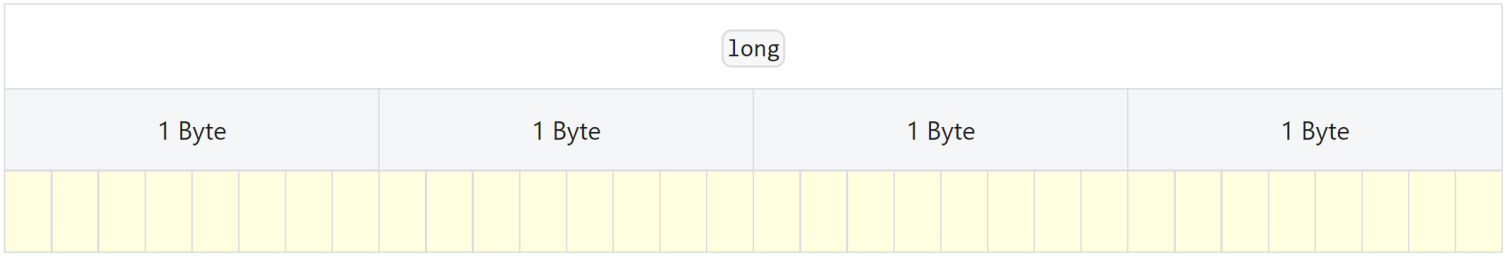
Specifying the size of an int ensures that the type contains a minimum number of bits. The three specifiers are:

- `short`
- `long`
- `long long`

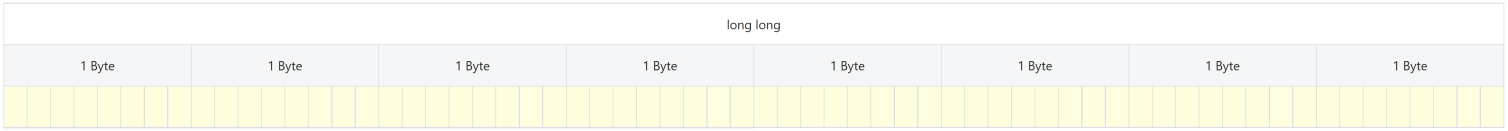
A `short int` (or simply, a `short`) contains at least 16 bits:



A `long int` (or simply, a `long`) contains at least 32 bits:



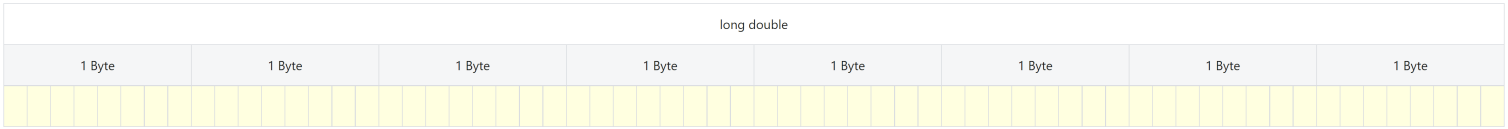
A `long long int` (or simply, a `long long`) contains at least 64 bits:



The size of a simple `int` is no less than the size of a `short`.

`double` Type Size Specifier

The size of a `long double` depends on the environment and is typically at least 64 bits:



Specifying the `long double` type only ensures that it contains at least as many bits as a `double`. The C language does not require a `long double` to contain a minimum number of bits.

`const` Qualifier

Any type can hold a constant value. A constant value cannot be changed. To qualify a type as holding a constant value we use the keyword const. A type qualified as `const` is unmodifiable. That is, if a program instruction attempts to modify a `const` qualified type, the compiler will report an error.

Representing Values

Hardware manufacturers distinguish integral types from floating-point types and represent integral data and floating-point data differently.

- integral types: `char` `int`
- floating-point types: `float` `double`

Integral Types

C stores the `char` and `int` data in equivalent binary form. Binary form represents the value stored exactly. To learn how to convert between decimal and binary representation refer to the appendix entitled Data Conversions.

Characters and Symbols

C stores characters and symbols in `char` types. Since characters and symbols have no intrinsic binary representation, the host platform provides the collating sequence for associating each character and symbol with a unique integer value. C stores the integer value from this sequence as the representative of the character or symbol.

The two popular collating sequences are ASCII and EBCDIC. ASCII is more popular. ASCII represents the letter A by the bit pattern 010000012, that is the hexadecimal value 0x41, that is the decimal value 65. EBCDIC represents the letter A by the bit pattern 110000012, that is the hexadecimal value 0xC1, that is the decimal value 193.

ASCII and EBCDIC are not compatible. The symbol order in ASCII differs from that in EBCDIC. In ASCII, the digits precede the letters, while in EBCDIC, the letters precede the digits. If we sort symbolic information that contains digits and letters, we will obtain different results under each sequence.

Neither ASCII nor EBCDIC contain enough values to represent most of the characters and symbols in the world languages. The Unicode standard, which is compatible with ASCII, provides a much more comprehensive collating system. We use the ASCII collating sequence throughout these notes.

Negative Values (Optional)

There are three schemes for storing negative integers:

- 2's complement notation (most popular)
- 1's complement notation
- sign magnitude notation

All three represent non-negative values identically. Under the 2's complement rule, there is only one representation of 0 and separate addition and subtraction circuits in the ALU are unnecessary.

To obtain the 2's complement of an integer, we

- flip the bits
- add one

For example, we represent the integer -92 by 10100100_2

Bit #	7	6	5	4	3	2	1	0
92 =>	0	1	0	1	1	1	0	0
Flip Bits	1	0	1	0	0	0	1	1
Add 1	0	0	0	0	0	0	0	1
-92 =>	1	0	1	0	0	1	0	0

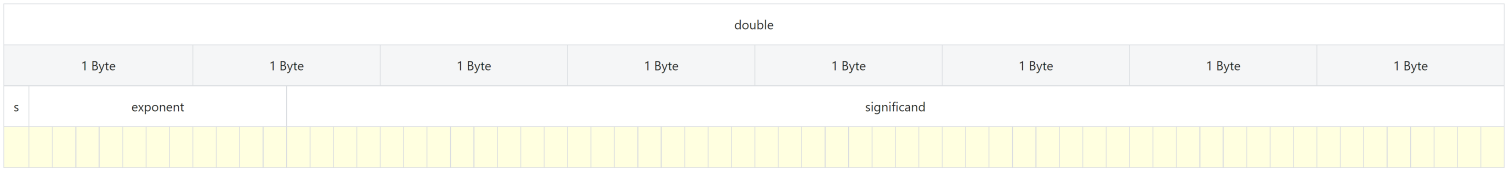
Floating-Point Data

Floating-point types store tiny as well as huge values by decomposing the values into three distinct components: a sign, an exponent and a significand. The C language leaves the implementation details to the hardware manufacturer.

The most popular model is the IEEE (I-triple-E or Institute of Electrical and Electronics Engineers) Standard 754 for Binary and Floating-Point Arithmetic. Under IEEE 754, a float has 32 bits, consisting of one sign bit, an 8-bit exponent and a 23-bit significand (or mantissa):

<table>
<tr><td colspan="32">float</td></tr>
<tr><td colspan="8">1 Byte</td><td colspan="8">1 Byte</td><td colspan="8">1 Byte</td><td colspan="8">1 Byte</td></tr>
<tr><td>s</td><td colspan="7">exponent</td><td colspan="24">significand</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Under IEEE 754, a double occupies 64 bits, has one sign bit, an 11-bit exponent and a 52-bit significand:



Since the number of bits in the significand is limited, the `float` and `double` types cannot store all possible floating-point values exactly. That is, the floating-point types store values approximately.

Value Ranges

The number of bytes allocated for a type determines the range of values that that type can store.

Integral Types

The ranges of values for the integral types are shown below. Ranges for some types depend on the execution environment:

Type	Size	Min	Max
`char`	8 bits	-128	127
`char`	8 bits	0	255
`short`	>= 16 bits	-32,768	32,767
`int`	2 bytes	-32,768	32,767
`int`	4 bytes	-2,147,483,648	2,147,483,647
`long`	>= 32 bits	-2,147,483,648	2,147,483,647
`long long`	>= 64 bits	-9,233,372,036,854,775,808	9,233,372,036,854,775,807

Floating-Point Types

The limits on a float and double depend on the execution environment:

Type	Size	Significant	Min Exponent	Max Exponent
`float`	minimum	6	-37	37
`float`	typical	6	-37	37
`double`	minimum	10	-37	37
`double`	typical	15	-307	307
`long double`	typical	15	-307	307

> **NOTE**
>
> Both the number of significant digits and the range of the exponent are limited. The limits on the exponent are in base 10.

Variable Declarations

We store program data in variables A declaration associates a program variable with a type. The type identifies the properties of the variable.

In C, a declaration takes the form:

```
[const] type identifier [= initial value];
```

The brackets denote an optional part of the syntax.

We select a meaningful name for the identifier and optionally set the variable's initial value. We conclude the declaration with a semi-colon, making it a complete statement.

For example:

```
char  children;
int   nPages;
float cashFare;
const double pi = 3.14159265;
```

> **GOOD PRACTICE**
>
> It is good practice to:
>
> - declare all variables at the **beginning** of the function
> - group related variables together
>
> Following this practice will contribute towards optimal variable organization and easier to maintain code.

Multiple Declarations

We may group the identifiers of variables that share the same type within a single declaration by separating the identifiers by commas.

For example,

```
char   children, digit;
int    nPages, nBooks, nRooms;
float  cashFare, height, weight;
double loan, mortgage;
```

Naming Conventions

We may select any identifier for a variable that satisfies the following naming conventions:

- starts with a letter or an underscore (_)
- contains any combination of letters, digits and underscores (_)
- contains less than 32 characters (some compilers allow more, others do not)
- is not a C reserved word

> **GOOD VARIABLE NAMING TECHNIQUES**
>
> Variable names (identifiers) should...
>
> - be self-documenting (should not require comments to describe what they are used for)
> - be concise but not so short that it is cryptic
> - accurately describes the data being stored

- help with the reading of the code
- use "camelNotation" (first letter of each word is capitalized with the exception of the first word)
- avoid underscore characters which are commonly used in system libraries to avoid conflicts
- **not** use underscore (_) characters in order to avoid conflicts with system libraries

Reserved Words

The C language reserves the following words for its own use:

```
auto        _Bool      break     case
char        _Complex   const     continue
default     restrict   do        double
else        enum       extern    float
for         goto       if        _Imaginary
inline      int        long      register
return      short      signed    sizeof
static      struct     switch    typedef
union       unsigned   void      volatile
while
```

For upward compatibility with C++, we avoid using the following C++ reserved words:

```
asm             friend            template
bool            mutable           this
catch           namespace         throw
class           new               true
const_cast      operator          try
delete          private           typeid
dynamic_cast    protected         typename
explicit        public            using
export          reinterpret_cast  virtual
false           static_cast       wchar_t
```

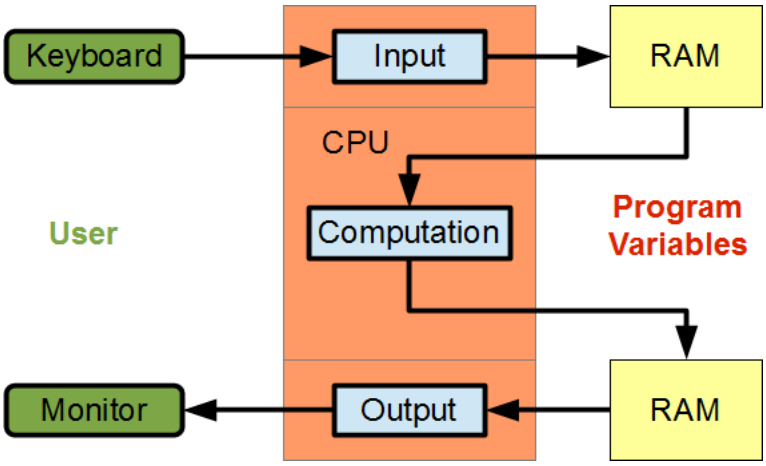
A Simple Calculation

Learning Outcomes

- Create a computer program to solve a basic programming task

Introduction

Application programs receive input from the user, convert that input into output and display the output to the user. The user enters the input through a keyboard or similar device and receives converted output through a monitor or similar device. Program instructions store the input data in RAM, retrieve that input data from RAM, convert it using the ALU and FPA in the CPU and store the result in RAM as illustrated in the figure below.



A program that directs these operations consists of both program variables and program instructions that operate on data stored in those variables.

This chapter describes how to write an interactive program for calculating the area of a circle. The chapter covers the syntax for defining a program constant, accepting the radius from the user, calculating the area, storing the result in a variable and displaying a copy of that result to the user.

Constant Values

Constant values in a program can be numbers, characters, or string literals. Each constant is of a specific type. We define the type of a constant, like the type of a variable, when we declare the constant.

Numeric Constants

We specify the type of a numeric constants by a suffix, if any, on the value itself and possibly a decimal point.

Type	Suffix	Example
int		1456234
long	L or l	75456234L
long long	LL or ll	75456234678LL
float	F or f	1.234F
double		1.234
long double	L or l	1.234L

To define a numeric constant in hexadecimal representation, we prefix the value with `0x`.

```
const int x = 0x5C;  // same as     const int x = 92;
```

Example

To define the constant `pi` (π) to 8 significant digits, we select the `float` type and write

```
int main(void)
{
    const float pi = 3.14159f; // pi is a constant float

    // ... completed below

    return 0;
}
```

The `const` keyword qualifies the value stored in the 'variable' `pi` as unmodifiable.

Character Constants

All character constants are of `char` type. The ways of defining a character constant include:

- the digit or letter enclosed in single quotes - for example `'A'`
- the decimal value from the collating sequence - for example `65` for `'A'` (ASCII)
- the hexadecimal value from the collating sequence - for example `0x41` for `'A'` (ASCII)

The single-quotes form is the preferred form, since it is independent of the collating sequence of the execution environment.

Escape Sequences

Character constants include special actions and symbols. We define special actions and symbols by escape sequences. The backslash (`\`) before each symbol identifies the symbol as part of an escape sequence:

Character	Sequence	ASCII	EBCDIC
alarm	\a	7	47
backspace	\b	8	22
form feed	\f	12	12
newline	\n	10	37
carriage return	\r	13	13
horizontal tab	\t	9	5
vertical tab	\v	11	11
backslash	\\	92	*

> **INFO**
>
> In the IBM reference card, System /370 Architecture Reference Summary, the `\` does not have an EBCDIC code. Its value may vary from machine to machine.

Escape sequences are relatively independent of the execution environment. Their decimal values however vary with the collating sequence of the execution environment and should be avoided.

String Literals

A string literal is a sequence of characters enclosed within a pair of double quotes. For example,

```
"This is C\n"
```

The `\n` character constant adds a new line to the end of the string.

Simple Input

The `scanf(...)` instruction accepts data from the user (that is, the standard input device) and stores that data in memory at the address of the specified program variable. The instruction takes the form:

```
scanf(format, address);
```

This statement calls the `scanf()` procedure, which performs the input operation. We say that format and address are the arguments in our call to `scanf()`.

Format

format is a string literal that describes how to convert the text entered by the user into data stored in memory. ***format*** contains the conversion specifier for translating the input characters. Conversion specifiers begin with a `%` symbol and identify the type of the destination variable. The most common specifiers are listed below.

Specifier	Input Text is a	Destination Type
%c	character	char
%d	decimal	int, short
%f	floating-point	float
%lf	floating-point	double

A more complete table is listed in the chapter entitled Input and Output.

Address

address contains the address of the destination variable. We use the prefix `&` to refer to the 'address of' of a variable.

Example (continued)

To accept the radius of the circle, we write

```
#include <stdio.h>              // for printf, scanf

int main(void)
{
    const float pi = 3.14159f;   // pi is a constant float
    float radius;                // radius is a float

    printf("Enter radius : ");   // prompt user for radius input
    scanf("%f", &radius);        // accept radius value from user

    // ... completed below

    return 0;
}
```

The argument in the call to `scanf()` is the address of radius, not the value of radius.

Coding the value radius as the argument is likely to generate a run-time error

```
scanf("%f", radius); //  ERROR possibly SEGMENTATION FAULT
```

Missing `&` in the call to `scanf()` is a common mistake for beginners and does not necessarily produce a compiler error or warning. Some compilers accept options (such as `-W`) to produce warnings, which may identify such errors.

Computation

We know that the area of a circle is given by the formula:

```
A = π r2
```

To store the area in memory involves 4 program instructions:

- define a variable to hold the area (a declaration)
- square the radius (an expression)
- multiply the square by π (another expression)
- assign the result to the defined variable (another expression)

Multiplication

The multiplication operation takes the form:

```
operand * operand
```

operand is a placeholder for the variable or constant being multiplied. `*` denotes the 'multiply by' operation. The value of this expression is equal to the result of the multiplication.

Assignment

The assignment operation stores the value of an expression in the memory location of the destination variable. Assignment takes the form:

```
destination = expression
```

destination is a placeholder for the destination variable. ***expression*** refers to the value to be assigned to the destination variable. `=` denotes the 'is assigned from' operation. We call `=` the assignment operator.

> **NOTE**
>
> Assignment is a unidirectional operation. ***destination*** must be a variable; that is, it must have a location in memory.

C compilers reject statements such as:

```
4 = age;  // *** ERROR cannot set 4 to the value in age ***
```

Example (continued)

Adding the statements to store the area in memory yields:

```
#include <stdio.h>              // for printf, scanf

int main(void)
{
    const float pi = 3.14159f;   // pi is a constant float
    float radius;                // radius is a float
    float area;                  // area is a float

    printf("Enter radius : ");   // prompt user for radius input
    scanf("%f", &radius);        // accept radius value from user

    area = pi * radius * radius; // calculate area from radius

    // ... completed below

    return 0;
}
```

Since the C language does not define an exponentiation operator, we need to calculate the square of the radius explicitly. Later, we will learn the `pow` procedure to perform exponentiation.

Simple Output

The `printf(...)` instruction reports the value of a variable or expression to the user (that is, copies the value to the standard output device). The instruction takes the form:

```
printf(format, expression);
```

This statement calls the `printf()` procedure, which performs the operation. We say that ***format*** and ***expression*** are arguments in our call to `printf()`.

Format

format is a string literal describing how to convert data stored in memory into text readable by the user. ***format*** contains the conversion specifier and any characters to be output directly. The conversion specifier begins with a `%` symbol and identifies the type of the source variable. The most common specifiers are listed below.

A more complete table is listed in the chapter entitled Input and Output.

The default number of decimal places displayed by `%f` and `%lf` is 6. To display two decimal places, we write `%.2f` or `%.2lf`.

Expression

expression is a placeholder for the source variable. The `printf()` procedure copies the variable and converts it into the output text.

Example (completed)

The complete program for calculating the area of a circle is:

```c
// Area of a Circle
// area.c

#include <stdio.h>              // for printf, scanf

int main(void)
{
    const float pi = 3.14159f;  // pi is a constant float
    float radius;               // radius is a float
    float area;                 // area is a float

    printf("Enter radius : ");  // prompt user for radius input
    scanf("%f", &radius);       // accept radius value from user

    area = pi * radius * radius; // calculate area from radius

    printf("Area = %f\n", area); // copy area to standard output

    return 0;
}
```

The input and output while executing this program is:

```
Enter radius : 1
Area = 3.141593
```

C rounds floating-point output to 6 decimal places by default.

Expressions

Learning Outcomes

After reading this section, you will be able to:

- Code various expressions that apply operations on operands of program type

Introduction

Programming languages support operators that combine variables and constants into expressions for transforming existing data into new data. These operators take one or more operands. The operands may be variables, constants or other expressions. The C language supports a comprehensive set of these operators for arithmetic, relational and logical expressions.

This chapter describes the supported operators in detail, what happens when the operands are of different types, how to change the type of an operand and the order of evaluation of sub-expressions within expressions. The introduction to this detailed description is a brief overview of the hardware components that evaluate expressions. These are the ALU and FPA inside the CPU.

Evaluating Expressions

The ALU evaluates the simplest of instructions on integer values: for instance, additions where the operands are of the same type. The FPA evaluates the simplest of instructions on floating-point values. C compilers simplify C language expressions into sets of hardware instructions that either the ALU or the FPA can process.

The ALU receives the expression's operator from the Control Unit, applies that operator to integer values stored in the CPU's registers and places the result in one of the CPU's registers. The FPA does the same but for floating-point values.

The expressions that the ALU can process on integer types are:

- arithmetic
- relational
- logical

The FPA can processes these same kinds of expressions on floating-point types.

Arithmetic Expressions

Arithmetic expressions consist of:

- integral operands - destined for processing by the **ALU**
- floating-point operands - destined for processing by the **FPA**

Integral Operands

The C language supports 5 binary and 2 unary arithmetic operations on integral (`int` and `char`) operands. Here, the term *binary* refers to two operands; *unary* refers to one operand.

Binary Operations

The ***binary*** arithmetic operations on integers are addition, subtraction, multiplication, division and remaindering. Expressions take one of the forms listed below:

Arithmetic Expression	Meaning
operand `+` operand	add the operands
operand `-` operand	subtract the right from the left operand
operand `*` operand	multiply the operands
operand `/` operand	divide the left by the right operand
operand `%` operand	remainder of the division of left by right

Division of one integer by another yields a whole number. If the division is not exact, the operation discards the remainder. The expression evaluates to the truncated integer result; that is, the whole number without any remainder. The expression with the modulus operator (%) evaluates to the remainder alone.

For example:

```
34 / 10    // evaluates to 3 (3 groups of 10 people)
34 % 10    // evaluates to 4 (4 person left without a group)
```

Unary Operations

The ***unary*** arithmetic operations are identity and negation. Expressions take one of the forms listed below:

Arithmetic Expression	Meaning
`+` operand	evaluates to the operand
`-` operand	changes the sign of the operand

The plus operator leaves the value unchanged and is present for language symmetry.

Floating-Point

Operands

The C language supports 4 binary and 2 unary arithmetic operations on the floating-point (`float` and `double`) operands.

Binary

The binary arithmetic operations on floating-point values are addition, subtraction, multiplication and division. Expressions take one of the forms listed below:

Arithmetic Expression	Meaning
operand `+` operand	add the operands
operand `-` operand	subtract the right from the left operand
operand `*` operand	multiply the operands
operand `/` operand	divide the left by the right operand

The division operator (/) evaluates to a floating-point result. There is no remainder operator for floating-point operands.

Unary

The unary operations are identity and negation. Expressions take the form listed below:

Arithmetic Expression	Meaning
`+` operand	evaluates to the operand
`-` operand	change the sign of the operand

The plus operator leaves the value unchanged and is present for language symmetry.

Limits (Optional)

The result of any operation is an expression of related type. Arithmetic operations can produce values that are outside the range of the expression's type.

Consider the following program, which multiplies two `int`'s and then two `double`'s

```c
// Limits on Arithmetic Expressions
// limits.c

int main(void)
{
        int i, j, ij;
        double x, y, xy;

        printf("Enter an integer : ");
        scanf("%d", &i);
        printf("Enter an integer : ");
        scanf("%d", &j);
        printf("Enter a floating-point number : ");
        scanf("%lf", &x);
        printf("Enter a floating-point number : ");
        scanf("%lf", &y);

        ij = i * j;
        xy = x * y;

        printf("%d * %d = %d\n", i, j, ij);
        printf("%le * %le = %le\n", x, y, xy);

        return 0;
}
```

Compile this program and execute it inputting different values. Try some very small numbers. Try some very large numbers. When does this program give incorrect results? When it does, explain why?

Relational Expressions

The C language supports 6 relational operations. A relational expression evaluates a condition. It compares two values and yields ***1*** if the condition is ***true*** and ***0*** if the condition is ***false***. The value of a relational expression is of type `int`. Relational expressions take one of the forms listed below:

Relational Expression	Meaning
operand `==` operand	operands are equal

Relational Expression	Meaning
operand `>` operand	left is greater than the right
operand `>=` operand	left is greater than or equal to the right
operand `<` operand	left is less than the right
operand `<=` operand	left is less than or equal to the right
operand `!=` operand	left is not equal to the right

The operands may be integral types or floating-point types.

Example

The following program, accepts two `int`'s and outputs 1 if they are equal; 0 otherwise:

```
// Relational Expressions
 // relational.c

 int main(void)
 {
        int i, j, k;

        printf("Enter an integer : ");
        scanf("%d", &i);
        printf("Enter an integer : ");
        scanf("%d", &j);

        k = i == j; // compare i to j and assign result to k

        printf("%d == %d yields %d\n", i, j, k);

        return 0;
 }
```

The first conversion specifier in the format string of the last `printf()` corresponds to `i`, the second corresponds to `j` and the third corresponds to `k`.

Logical Expressions

The C language does not have reserved words for true or false. It interprets the value 0 as false and any other value as true. C supports 3 logical operators. Logical expressions yield 1 if the result is true and 0 if the result is false. The value of a logical expression is of type int. Logical expressions take one of the forms listed below:

Relational Expression	Meaning
operand `&&` operand	both operands are true
operand `||` operand	one of the operands is true
`!` operand	the operand is not true

The operands may be integral types or floating-point types.

Example

The following program, accepts three `int`'s and outputs ***1*** if the second is greater than or equal to the first and less than or equal to the third; ***0*** otherwise:

```
// Logical Expressions
// logical.c

int main(void)
{
        int i, j, k, m;

        printf("Enter an integer : ");
        scanf("%d", &i);
        printf("Enter an integer : ");
        scanf("%d", &j);
        printf("Enter an integer : ");
        scanf("%d", &k);

        m = j >= i && j <= k; // store the value of this expression in m

        printf("%d >= %d and %d <= %d yields %d\n", j, i, j, k, m);

        return 0;
}
```

The conversion specifiers in the last `printf()` correspond to the arguments in the same order (first to ***j***, second to ***i***, etc.).

deMorgan's Law

deMorgan's law is a handy rule for converting conditions in logical expressions. The law states that:

> **NOTE**
>
> The opposite of a compound condition is the compound condition with all sub-conditions reversed, all &&'s changed to ||'s and all ||'s to &&'s.

Consider the following definition of an adult:

```
adult = !child && !senior;
```

This definition is logically identical to:

```
adult = !(child || senior);
```

The parentheses direct the compiler to evaluate the enclosed expression first.

By applying ***deMorgan's law***, we can often re-write a compound condition in a more readable form.

Shorthand Assignments

The C language also supports shorthand operators that combine an arithmetic expression with an assignment expression. These operators store the result of the arithmetic expression in the left operand.

Integral Operands

C has 5 binary and 2 unary shorthand assignment operators for integral (`int` and `char`) operands.

Binary Operands

The binary operators yield the same result as shown in the longhand expressions listed alongside:

Expression	Shorthand	Longhand	Meaning
operand `+=` operand	`i += 4`	`i = i + 4`	add 4 to i and assign to i
operand `-=` operand	`i -= 4`	`i = i - 4`	subtract 4 from i and assign to i
operand `*=` operand	`i *= 4`	`i = i * 4`	multiply i by 4 and assign to i
operand `/=` operand	`i /= 4`	`i = i / 4`	divide i by 4 and assign to i
operand `%=` operand	`i %= 4`	`i = i % 4`	remainder after i / 4 and assign to i

Unary Operands

The unary operators yield the same result as shown in the longhand expressions listed alongside:

Expression	Shorthand	Longhand	Meaning
++operand	`++i`	`i = i + 1`	increment i by 1
operand++	`i++`	`i = i + 1`	increment i by 1
--operand	`--i`	`i = i - 1`	decrement i by 1
operand--	`i--`	`i = i - 1`	decrement i by 1

We call the unary operator that precedes its operand a ***prefix*** operator and the unary operator that succeeds its operand a ***postfix*** operator.

The difference between the ***prefix*** and ***postfix*** expressions is in the value of the expression itself. The ***prefix*** operator changes the value of its operand and sets the expression's value to be the changed value. The ***postfix*** operator sets the expression's value to the operand's original value and then changes the operand's value. In other words, the ***prefix*** operator changes the value before using it, while the ***postfix*** operator changes the value after using it.

```c
// Prefix and Postfix Operators
// pre_post.c

int main(void)
{
        int age = 19;

        printf("Prefix:  %d\n", ++age);
        printf("         %d\n", age);
        printf("Postfix: %d\n", age++);
        printf("         %d\n", ++age);

        return 0;
}
```

Floating-Point Operands

C has 4 binary and 2 unary shorthand assignment operators for floating-point (`float` and `double`) operands.

Binary Operands

The binary operators yield the same result as in the longhand expressions listed alongside:

Expression	Shorthand	Longhand	Meaning

Expression	Shorthand	Longhand	Meaning
operand `+=` operand	`x += 4.1`	`x = x + 4.1`	add 4.1 to x and assign to x
operand `-=` operand	`x -= 4.1`	`x = x - 4.1`	subtract 4.1 from x and assign to x
operand `*=` operand	`x *= 4.1`	`x = x * 4.1`	multiply x by 4.1 and assign to x
operand `/=` operand	`x /= 4.1`	`x = x / 4.1`	divide x by 4.1 and assign to x

Unary Operands

Expression	Shorthand	Longhand	Meaning
++operand	`++x`	`x = x + 1`	increment i by 1.0
operand++	`x++`	`x = x + 1`	increment i by 1.0
--operand	`--x`	`x = x - 1`	decrement i by 1.0
operand--	`x--`	`x = x - 1`	decrement i by 1.0

The prefix and postfix operators operate on floating-point operands in the same way as on integral operands.

Ambiguities

Compact use of shorthand operators can yield ambiguous results across different platforms. Consider the following longhand statements:

```
int i = 5;
int j = i++ + i; // *** AMBIGUOUS ***
```

One compiler may increment the first ***i*** before the addition, while another compiler may increment ***i*** after the addition. The C language does not address this ambiguity and only stipulates that the value must be incremented before the semi-colon. To avoid ambiguity, we re-write this code to make our intent explicit:

```
int i = 5;
i++;           // ++ before
int j = i + i; // j is 12
int i = 5;
int j = i + i; // j is 10
i++;           // ++ after
```

Casting

The C language supports conversions from one type to another. To convert the type of an operand, we precede the operand with the target type enclosed within parentheses. We call such an expression a cast. Casting expressions take one of the forms listed below:

Cast Expression	Meaning
(`long double`) operand	`long double` version of operand
(`double`) operand	`double` version of operand
(`float`) operand	`float` version of operand

Cast Expression	Meaning
(`long long`) operand	`long long` version of operand
(`long`) operand	`long` version of operand
(`int`) operand	`int` version of operand
(`short`) operand	`short` version of operand
(`char`) operand	`char` version of operand

Consider the example below. To obtain the number of hours in fractional form, we cast minutes to a `float` type and then divide it by 60. The input and output are listed on the right:

```c
// From minutes to hours
// cast.c

int main(void)
{
        int minutes;
        float hours;

        printf("Minutes ? ");
        scanf("%d", &minutes);
        hours = (float)minutes / 60;
        printf("= %.2lf hours\n", hours);

        return 0;
}
```

Without the type cast, the output for the same input would have been 0.00 hours.

Mixed-Type Expressions

Since CPUs process integral expressions and floating-point expressions differently (using the ALU and the FPA respectively), they only handle expressions with operands of the same type. For expressions with operands of different types, we need rules for converting operands of one type to another type.

The C language using the following ranking:

`long double`	**higher**
`double`	...
`float`	...
`long long`	...
`long`	...
`int`	...
`short`	...
`char`	lower

There are two distinct kinds of expressions to consider with respect to type coercion:

- assignment expressions
- arithmetic and relational expressions

Assignment Expressions

Promotion

If the left operand in an assignment expression is of a higher type than the right operand, the compiler promotes the right operand to the type of the left operand. For the example below, the compiler promotes the right operand (loonies) to a `double` before completing the assignment:

```
// Promotion with Assignment Operators
// promotion.c

int main(void)
{
        int loonies;
        double cash;

        printf("Loonies ? ");
        scanf("%d", &loonies);
        cash = loonies; // promotion
        printf("Cash is $%.2lf\n", cash);

        return 0;
}
```

Narrowing

If the left operand in an assignment expression is of a lower type than the right operand, the compiler truncates the right operand to the type of the left operand. For the example below, the compiler truncates the type of the right operand (cash) to an `int`:

```
// Truncation with Assignment Operators
// truncation.c

int main(void)
{
        double cash;
        int loonies;

        printf("Cash ? ");
        scanf("%lf", &cash);
        loonies = cash; // truncation
        printf("%d loonies.\n", loonies);

        return 0;
}
```

Arithmetic and Relational Expressions

C compilers promote the operand of lower type in an arithmetic or relational expression to an operand of the higher type before evaluating the expression. The table below lists the type of the promoted operand.

Example

```
1034 * 10   evaluates to 10340    // an int result
1034 * 10.0 evaluates to 10340.0  // a double result
```

```
1034 * 10L  evaluates to 10340L   // a long result
1034 * 10.f evaluates to 10340.0f // a float result
```

Compound Expressions

A compound expression is an expression that contains an expression as one of its operands. C compilers evaluate compound expressions according to specific rules called rules of precedence. These rules define the order of evaluation of expressions based on the operators involved. C compilers evaluate the expression with the operator that has the highest precedence first.

The order of precedence, from highest to lowest, and the direction of evaluation are listed in the table below:

Operator	Evaluate From
`++ -- (postfix)`	left to right
`++ -- (prefix) + - & ! (all unary)`	right to left
`(type)`	right to left
`* / %`	left to right
`+ -`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&&`	left to right
`||`	left to right
`= += -= *= /= %=`	right to left

To change the order of evaluation, we introduce parentheses. C compilers evaluate the expressions within parentheses (( )) before applying the rules of precedence. For example:

```
  2 + 3   * 5 evaluates to 2 + 15, which evaluates to 17
( 2 + 3 ) * 5 evaluates to 5 * 5, which  evaluates to 25
3 / (double)2 evaluates to 3 / 2.0, which evaluates to 1.5
(double)(3 / 2) evaluates to (double)1, which evaluates to 1.0
```

On this page

Logic

Learning Outcomes

After reading this section, you will be able to:

- Design procedures using selection and iteration constructs to solve a programming task

Introduction

A complete programming language includes facilities to implement sequential constructs, in which one statement follows another and the statements are executed in order, and two other constructs, which represent modifications of sequential constructs. Selection constructs represent different paths through the set of instructions. Iteration constructs represent repetition of the same set of instructions until a specified condition has been met. The three classes of constructs required to complete a programming language are illustrated in the figure below.

Since programmers who maintain application software are typically not those who develop that software originally and the maintenance programmers may change throughout the lifetime of a software application, it is critical that the software is not only readable but also easy to upgrade and maintain. The principles of structured programming, which were developed in the 1960s, provide important coding guidelines that respect this objective.

This chapter introduces the selection and iteration constructs supported by the C language and describes how to implement structured programming principles in coding iterations.

Structured Programming

A structured program consists of sets of simple constructs, each of which has one entry point and one exit point. Any programmer may replace one construct with an upgraded construct without affecting the other constructs in the program or introducing errors ("bugs").

The simplest example of a structured construct is a ***sequence***. A sequence is either a simple statement or a code block. A ***code block*** is a set of statements enclosed in a pair of curly braces to be executed sequentially.

Example Simple Statement

```
// single statement
printf("I like pizza\n");
```

Example Code Block

```
// code block (upgrade)
{
    printf("I like pizza\n");
    printf("I want more pizza\n");
}
```

Unlike a single statement, a C code block does not require a terminating semi-colon of its own (after the closing brace).

Preliminary Design

During the design stage of a programming solution, it is helpful to outline the steps involved. Well-established techniques include:

- pseudo-coding
- flow charting

Clear and concise pseudo-code or flow charts improve chances are that our coding will also be clear and concise.

Pseudo-Code

Pseudo-code is a set of shorthand notes in a human (non-programming) language that itemizes the key steps in the sequence of instructions that produce a programming solution. For example, the pseudo code for calculating the change in a vending machine might look something like

1. Declare variables for quarters and nickels
2. Calculate the number of quarters in the change
3. Calculate the remainder to be returned in nickels
4. Output the change in quarters and nickels

Flow Charts

A ***flow chart*** is a set of conventional symbols connected by arrows that illustrate the flow of control through a programming solution. Popular sets of symbols for sequences, selections and iterations are shown below:

Usage of these sets with the C language is illustrated below.

Selection Constructs

The C language supports three selection constructs:

- optional path
- alternative paths
- conditional expression

The flow charts for these three constructs are shown below:

Optional Path

The simplest selection construct executes a sequence only if a certain condition is satisfied; that is, only if the condition is true. This optional selection takes the form:

```
if (condition)
   sequence
```

Parentheses enclose the condition, which may be a relational expression or a logical expression. The sequence may be a single statement or a code block.

Single Statement

```
if (likePizza == 1)
    printf("I like pizza\n");
```

Code Block (more than a single statement)

```
```

```c
if (likePizza == 1)
{
     printf("I like pizza\n");
     printf("I want more pizza\n");
}
```

The program executes the sequence only if `likePizza` is equal to `1`. Otherwise, the program bypasses the sequence altogether.

Alternative Paths

The C language supports two ways of describing alternative paths: an binary select construct and a multiple selection construct.

Binary Selection

The binary selection construct executes one of a set of alternative sequences. This construct takes the form

```c
 if (condition)
    sequence
 else
    sequence
```

Parentheses enclose the condition, which may be a relational expression or a logical expression. The sequences may be ***single statements*** or ***code blocks***. The program executes the sequence following the if only if the condition is true. The program executes the sequence following the ***else*** only if the condition is false.

Single Statement

```c
if (likePizza == 1)
    printf("I like pizza\n");
else
    printf("I hate pizza\n");
```

Code Block (more than a single statement)

```c
if (likePizza == 1)
{
    printf("I like pizza\n");
}
else
{
    printf("I hate pizza\n");
    printf("I don't want pizza\n");
}
```

> **READABILITY AND MAINTAINABILITY**
>
> Although it is not required to create a code block for single-line statements, it is suggested for maximum readability and maintainability of code that you create a code block.
>
> All code examples provided in these notes will assume this suggested style including the placement of the opening and closing curly braces be on their own dedicated lines (Allman).

Multiple Selection

For three alternative paths, we append an ***if else*** construct to the ***else*** keyword.

```c
 if (condition)
    sequence
 else if (condition)
```

```
    sequence
else
    sequence
```

If the first condition is true, the program skips the second and third sequences. If the first condition is false, the program skips the first sequence and evaluates the second condition. The program executes the second sequence only if the first condition is false and the second condition is true. The program executes the third sequence and skips the first two only if both conditions are false.

Compound Conditions

The condition in a selection construct may be a ***compound*** condition. A compound condition takes the form of a logical expression (see the section on ***Logical Expressions*** in the chapter on Expressions).

```
if (age > 12 && age < 16)
{
    printf("Student Fare - no id required\n");
}
else if (age > 15 && age < 20)
{
    printf("Student Fare - id is required\n");
}
else if (age < 13)
{
    printf("Child ride for free!\n");
}
else if (age >= 65)
{
    printf("Senior Fare - id is required\n");
}
else
{
    printf("Adult Fare\n");
}
```

Case-by-Case

The case-by-case selection construct compares a condition - simple or compound - against a set of constant values or constant expressions. This construct takes the form:

```
switch (condition)
{
case constant:
       sequence
       break;
case constant:
       sequence
       break;
default:
       sequence
}
```

If the condition matches a constant, the program executes the sequence associated with the case for that constant. The break; statement transfers control to the closing brace of the switch construct. Braces around the statements between case labels are unnecessary.

If a break statement is missing for a particular case, control flows through to the subsequent case and the program executes the sequence under that case as well.

The program executes the sequence following default only if the condition does not match any of the case constants. The `default` case is optional and this keyword may be omitted.

For example, the following code snippet compares the value of choice to `'A'` or `'a'`, `'B'` or `'b'`, and `'C'` or `'c'` until successful. If unsuccessful, the code snippet executes the statements under `default`.

```
char choice;
double cost;

printf("Enter your selection (a, b or c) ? ");
scanf("%c", &choice);

switch (choice)
{
case 'A' :
case 'a' :
    cost = 1.50;
    break;
case 'B' :
case 'b' :
    cost = 1.10;
    break;
case 'C' :
case 'c' :
    cost = 0.75;
    break;
default:
    choice = '?';
    cost = 0.0;
}

printf("%c costs %.2lf\n", choice, cost);
```

Conditional Expression

The ***conditional expression*** selection construct is shorthand for the ***alternative path*** construct. This ternary expression combines a condition and two sub-expressions using the `? :` operators:

```
condition ? operand : operand
```

If the condition is true, the expression evaluates to the operand between `?` and `:`. If the condition is false, the expression evaluates to the operand following `:`.

Example

```
int main()
{
    int minutes;
    char s;

    printf("How many minutes left ? ");
    scanf("%d", &minutes);

    s = minutes > 1 ? 's' : ' ';    // Conditional Expression

    printf("%d minute%c left\n", minutes, s);

    return 0;
}
```

If the operands in a conditional expression are themselves expressions, the conditional expression only evaluates the operand identified by the condition.

Iteration Constructs

The C language supports three iteration constructs:

- while
- do while
- for

Three instructions control the execution of an iteration: an initialization, a test condition, a change statement. The test condition may be simple or compound. The flow charts for the three constructs are shown below.

If the change statement is missing or if the test condition is always satisfied, the iteration continues without terminating and the program can never terminate. We say that such an iteration is an infinite loop.

while

The ***while*** construct executes its sequence as long as the test condition is true. This construct takes the form:

```
while (condition)
{
    sequence
}
```

Example

```
slices = 4;
while (slices > 0)
{
    slices--;
    printf("Gulp! Slices left %d\n", slices);
}
```

The above code produces the following output:

```
Gulp! Slices left 3
Gulp! Slices left 2
Gulp! Slices left 1
Gulp! Slices left 0
```

If the condition is never true (for example, if initially slice = 0), this construct never executes the sequence.

do while

he ***do while*** construct executes its sequence ***at least once*** and continues executing it as long as the test condition is true. This construct takes the form:

```
do {
    sequence
} while (condition);
```

Example

```
slices = 4;
do {
    slices--;
```

```
    printf("Gulp! Slices left %d\n", slices);
}while (slices > 0);
```

The above code produces the following output:

```
Gulp! Slices left 3
Gulp! Slices left 2
Gulp! Slices left 1
Gulp! Slices left 0
```

If we change the initial value to slices = 12 and the test condition to slices < 5, this iteration displays once and stops because the test condition is false.

```
slices = 12;
do {
    slices--;
    printf("Gulp! Slices left %d\n", slices);
} while (slices < 5);
```

The above code produces the following output:

```
Gulp! Slices left 11
```

This code contains a ***semantic error***: if the initial value was 5, the iteration would never end!

for

The ***for*** construct groups the initialization, test condition and change together, separating them with semi-colons. This construct takes the form:

```
for (initialization; condition; change)
{
    sequence
}
```

Example

```
for (slices = 4; slices > 0; --slices)
{
    printf("Gulp! Slices left %d\n", slices - 1);
}
```

Flags

Flagging is a method of coding iteration constructs within the ***single-entry single-exit principle*** of structured programming. Consider the flow-chart on the left side in the figure below. This design contains a path that crosses another path.

Flags are variables that determine whether an iteration continues or stops. A flag is either true or false. Flags helps ensure that no paths cross one another. By introducing a flag, we avoid the jump and multiple exit, obtain a flow chart where no path crosses any other and hence an improved design.

Example

The following code snippet uses a flag to terminate the iteration prematurely.

```
```

```
int done = 0;  // flag
int total = 0; // accumulator
for (i = 0; i < 10 && done == 0; i++)
{
    printf("Enter integer (0 to stop) ");
    scanf("%d", &value);
    if (value == 0)
    {
        done = 1;
    }
    else
    {
        total += value;
    }
}
printf("Total = %d\n", total);
```

Example execution of the code above:

```
Enter integer (0 to stop) 45
Enter integer (0 to stop) 32
Enter integer (0 to stop) 3
Enter integer (0 to stop) -6
Enter integer (0 to stop) 0
Total = 74
```

The test condition is compound (logical expression) due to the evaluation of both, the iterator `i` and the flag `done`. If `done == 1`, the iteration stops.

> **IMPORTANT**
>
> Until you learn how to evaluate and rationalize when to break the single-entry single-exit principle, you should apply the control flag approach to eloquently manage process flow. Listed below are some cases to avoid:
>
> - `break` (the `switch` construct should be the only construct using this, and only 1 per case)
> - `continue`
> - `exit`
> - `goto`
> - `return`
> - `exit`
> - iterator variable (should only be changed in a single place)

Avoid Jumps (Optional)

Designing a program with jumps or intersecting paths makes it more difficult to read. We refer to program code that contains paths that cross one another as spaghetti code. The roots of spaghetti coding lie in assembly languages (second-generation languages). Assembly languages include jump instructions. Jump instructions migrated to high-level languages as assembly language programmers started coding in higher-level languages. Spaghetti code was a serious problem in the 1960's. To improve readability, many programmers started to advocate complete avoidance of jump statements and introduced ***flags*** as the good-design alternative.

Nested Constructs

Enclosing one logic construct within another is called ***nesting***.

Nested Selections

A selection within another selection is called a ***nested selection***.

Example

```
if (grade < 50)
{
    if (sup == 1)
    {
        printf("Sup\n");
    }
    else
    {
        printf("Failed\n");
    }
}
else
{
    printf("Pass\n);
}
```

Dangling Else

An ambiguity arises in a ***nested if else*** construct that contains an optional sequence (***if***). Consider the following code snippet:

```
// Problem: Ambiguity
if (grade < 50)
    if (sup == 1)
        printf("Sup\n");
else // <-- Does this belong to 'if (grade < 50)' OR 'if (sup == 1)'?
    printf("Pass\n);
```

It is unclear as to which `if` the `else` belongs:

```
// Interpretation #1:
if (grade < 50)  // <-- The formatting suggests to this 'if'
    if (sup == 1)
        printf("Sup\n");
else
    printf("Pass\n);
```

... OR if the `else` is indented ...

```
// Interpretation #2:
if (grade < 50)
    if (sup == 1) // <-- now the formatting suggests this 'if'
        printf("Sup\n");
    else
        printf("Pass\n);
```

The C language always attaches the dangling `else` to the **innermost** `if` (regardless of how it is indented, interpretation #2 above is how it would be executed).

To guarantee the desired behaviour (interpretation #1 from above), we use code blocks (curly braces) to ensure the intended flow:

```
if (grade < 50)
{
    if (sup == 1)
    {
        printf("Sup\n");
    }
}
```

```
else
{
    printf("Pass\n);
}
```

Nested Iterations

An iteration within another iteration is called a ***nested iteration***.

The program below includes a nested iteration:

```
// Rows and Columns
// row_columns.c

#include <stdio.h>

int main(void)
{
    int i, j;
    for (i = 0; i < 5; i++)
    {
        for (j = 0; j < 5; j++)
        {
            printf("%d,%d  ", i, j);
        }
        printf("\n");
    }

    return 0;
}
```

The output of the code above:

```
0,0  0,1  0,2  0,3  0,4
1,0  1,1  1,2  1,3  1,4
2,0  2,1  2,2  2,3  2,4
3,0  3,1  3,2  3,3  3,4
4,0  4,1  4,2  4,3  4,4
```

Style Guidelines

Learning Outcomes

After reading this section, you will be able to:

- Self-document programs using comments and descriptive identifiers

Introduction

A well-written program is a pleasure to read. The coding style is consistent and clear throughout. The programmer looking for a bug sees a well-defined structure and finds it easy to focus on the portion of the code that is suspect. The programmer looking to upgrade the code sees how and where to incorporate changes. Although several programmers may have contributed to the code throughout its lifetime, the code itself appears to have been written by one programmer.

This chapter describes the coding style used throughout these notes and recommended for an introductory course in programming. The style is referred to as the Allman coding style.

Identifiers

All identifiers in a program should be self-descriptive. The reader should not have to search through the code for their meaning. It is better to embed the meaning in the name, rather than to explain it in a comment elsewhere in the code. By all means, avoid referring the reader to a document external to the code itself.

A program with short names is easier to read than one with long names. The human eye infers the meaning of a word from just a few letters that make up that word and the context within which the word is used. Reading long identifiers tires the eyes when searching through code. We follow the sophisticated conventions of our own languages and complying with them makes our programs all the more readable. Nouns describe objects, verbs describe actions.

Notations, such as Hungarian notation, that incorporate the type into the identifier will clutter source code unnecessarily. C compilers know the type of each identifier and readers do not need reminders in every place the identifier appears.

When selecting identifiers:

- adopt self-descriptive names, adding comments only if clarification is necessary
- prefer nouns for variable identifiers
- keep variable identifiers short - `temp`, rather than `temporary`, `id`, rather than `identification`,
- avoid cryptic identifiers - use just enough letters for the eye to infer the meaning from the context but no less (if you want to represent 'amount owed', `ao` is cryptic, while `amtOwed` is clear but concise)
- keep the identifiers of counters very short - use `i` rather than `loop_counter`, and `n` rather than `numberOfIterations`. This is context dependant and should only be applied to iterators and counters otherwise, the name becomes meaningless or cryptic.
- avoid decorating the identifier with Hungarian or similar notations (data type)
- use "camelNotation" (first letter of each word is capitalized with the exception of the first word)
- avoid underscore characters which are commonly used in system libraries to avoid conflicts

Layout

Professionals in the field of human-computer interaction confirm that layout and arrangement affects comfort and accessibility. Poorly laid out code frustrates and promotes misreading's.

Typographers, artists, and photographers know that negative space surrounding an image is as important as the image itself. Space itself can visually separate, making it easier to find something and draw attention to a certain part of a page.

Layout tools at our disposal include:

- indentation
- line length
- braces
- spaces
- comments

Indentation

Indentation helps define where a code block starts and ends, clearly showing the structure of our logic. The recommended indent in C programs is a tab of 4 or 8 characters.

Example:

```
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        for (k = 0; k < n; k++)
        {
            int ijk = i * j * k;
            if (ijk != 0)
            {
                printf(" %4d", ijk);
            }
            else
            {
                printf("    ");
            }
        }
        printf("\n");
    }
    printf("\n");
}
printf("That's all folks!!!\n");
```

Using tabs for indentation rather than spaces enables other programmers to adjust the indentation without difficulty in their own text editors. Using 8 characters per tab position heavily indents code to the far right and identifies code that may be a hog of compute cycles and a likely candidate for refactoring.

To minimize the effects of indentation with switch constructs, we align the subordinate case labels with the switch keyword:

```
switch(c)
{
case 'A' :
case 'a' :
        cost = 1.50;
        break;
case 'B' :
case 'b' :
        cost = 1.10;
        break;
case 'C' :
case 'c' :
        cost = 0.75;
        break;
default:
        c = '?';
```

```
        cost = 0.0;
}
```

Line Length

The practical limit on line length is 80 columns, including indentation. Many windows default to an 80-column width and break longer lines into chunks that are then difficult to read. Lines longer than 80 columns either truncate or wrap in hard copy printouts, which confuses readers.

String literals pose a special problem. We break them into a set of sub-string literals separated only by whitespace. C compilers discard the whitespace and concatenate the sub-string literals into a single string literal.

```
printf("%d substrings"
       " display as a"
       " single string"
       "\n", 3);
```

Produces the following output:

```
3 substrings display as a single string
```

Braces

The style of bracing used in these notes is that proposed by Eric Allman. We put the opening brace on its own line indented to the preceding statement and the closing brace on its own line in alignment with the opening brace.

```
if (i == 7)
{
    cost = 1.75;
    printf("Congrats!\n");
}
```

```
if (i == 7)
{
    cost = 1.75;
    printf("Congrats!\n");
}
else
{
    cost = 2.75;
    printf("Good luck next time!\n");
}
```

The opening and closing braces for a `do/while` construct is an exception:

```
do {
    printf("Guess i : ");
    scanf("%d", &i);
    if (i == 7)
    {
        cost = 1.75;
        printf("Congrats!\n");
    }
    else
    {
        cost = 2.75;
        printf("Good luck next time!\n");
    }
} while (i != 7);
```

Although braces are unnecessary with single statements, it is more clear to read and maintain when they are provided:

```
if (i == 7)
{
    printf("Congrats!\n");
}
else
{
    printf("Good luck next time!\n");
}
```

```
if (i == 7)
{
    printf("Congrats!\n");
    done = 1;
}
else
{
    printf("Good luck next time!\n");
}
```

Spaces

We add a single space after commas, semi-colons, most keywords and around most operators, except between parentheses and identifiers or constants, after unary operators and call identifiers.

For example:

```
int i; // space after keyword

scanf("%d", &i); // no space after unary operator

i = i * i; // single spaces around binary operators

if (i == 7) // no spaces between identifiers or constants and parentheses
{
    printf("Congrats!\n");
}

for (i = 0; i < 10; i++) // single space after ;
{
    printf("%d ", i);   // no space after call identifier
}
```

We avoid trailing spaces at the end of a line.

We add blank lines to distinguish the end of one construct from the start of another whenever either construct contains some complexity. However, we avoid superfluous blank lines.

Comments

We use comments to describe what is done, rather than how it is done. Comments introduce or summarize what follows. We keep them brief and avoid decoration or cuteness.

If it is important to comment data, we do so at the variable's declaration. Where units matter, we identify them. Where we comment variable declarations, we declare each variable on a separate line and use inline comments.

We preface every source file with a header comment that includes:

- the title of the program
- the source file name
- the name of the author(s)

For example:

```
 /* Payroll Deductions
  * payroll.c
  * Jane Doe
  */
```

Such header comments are helpful in locating the e-copy corresponding to a hard copy that we have in hand.

We align comments with the code they describe, indenting both identically, showing no preference for either comment or code.

```
// display even integers below 11
//
for (j = 0; j < 11; j += 2)
{
    printf(" %d", j);
}
```

Such comments summarize the code that follows and help the reader avoid studying that code in detail if it is not the target code for which they are searching.

Magic Numbers

We refer to values that appear out of nowhere in program code as magic numbers. These may be mathematical constants, standard rates or default values. We avoid magic numbers by identifying them with symbolic names and using those names throughout the code. We set their value in either of two ways:

- using an unmodifiable variable
- using a macro directive

Unmodifiable Variables

An **unmodifiable** variable takes the form

```
 const type SYMBOL = value;
```

For example:

```
const double PI = 3.14159;

int main(void)
{
    double radius, area;

    printf("Enter radius : ");
    scanf("%lf", &radius);

    area = PI * radius * radius;
    printf("The area of your circle is : %lf\n", area);

    return 0;
}
```

Macro Directive

A macro is **NOT** a variable but is used for substitution at compile-time. A macro directive takes the form:

```
#define SYMBOL value
```

We terminate this directive with an end of line character immediately following value.

The `define` directive instructs the C compiler to **substitute** every occurrence of SYMBOL with value throughout the code.

> **NOTE**
>
> Notice the absence of a semi-colon at the end of the directive. The substitution is a straightforward **search** and **replace**. The value itself may include whitespace.

For example,

```
#define PI 3.14159

int main(void)
{
        double radius, area;

        printf("Enter radius : ");
        scanf("%lf", &radius);

        area = PI * radius * radius;
        // At compile-time, the above statement becomes:
        // area = 3.14159 * radius * radius

        printf("The area of your circle is : %lf\n", area);

        return 0;
}
```

Miscellaneous

Other guidelines for enhancing and maintainability readability include:

- we avoid global variables (see Global Scope sub-section in the chapter on Information Hiding)
- we avoid variable identifiers that end in numbers
- we avoid using the character encodings for a particular environment (for example, ASCII or EBCDIC). Instead, we use escape sequences, which are universal. For example, to initialize c to the linefeed character (10 in ASCII and 37 in EBCDIC), use:

```
// prefer
//
char c = '\n';

// avoid
//
char c = 10; // ASCII
```

- we initialize iteration variables in the context of the iteration:

```
// prefer
//
for (i = 0; i < 10; i++)

// avoid
//
```

```
i = 0;
for (; i < 10; i++)
```

- we add special comments where code has been fine-tuned for efficient execution
- we avoid iterations with empty bodies
- we limit the initialization and iteration clauses in for statements to the iteration variables
- we avoid assignment expressions nested inside logical expressions
- we add an extra pair of parentheses where an assignment expression is also used as a condition
- we remove unreferenced variable declarations from our source code
- we remove all commented code and debugging statements from our release and production code

Testing & Debugging

Learning Outcomes

After reading this section, you will be able to:

- Trace the execution of a complete program to validate its correctness

Introduction

Testing and debugging skills are integral skills that a software developer refines throughout their career. Testing ensures that a program executes successfully for a well-defined range of values. Such a program might still crash for values outside this range. Each program needs to be thoroughly tested before release to a user community and with each patch to that release. Compilers identify syntactic errors with respect to the rules of the programming language, but cannot readily identify semantic errors; that is, errors in the meaning or intent of the code. Walkthroughs and code analysis help identify these errors.

Much of the time and effort involved in ensuring that a program executes correctly for all practical cases is spent on testing and debugging. Testing ensures that all of the paths through the program envisaged by the designer produce correct results. Debugging locates those 'bugs' that produce incorrect results. Over the years, computer scientists have developed sophisticated tools for testing and debugging. These tools are available in various development environments. The traditional walkthrough technique simulates instruction-by-instruction stepping of the CPU and its updating of program data in primary memory.

This chapter describes the kinds of errors that are common in source code, introduces testing and debugging techniques, and shows how to layout program variables in tabular form to facilitate comprehensive walks through the source code.

Errors

Programming errors are classified into either of two kinds:

- syntactic errors
- semantic errors

Syntactic Errors

Syntactic errors are errors that break the rules of the programming language. The most common syntactic errors in the C language are:

- missing semi-colon
- unnecessary semi-colon terminator in a #define directive
- undeclared variable name
- mismatched parentheses
- left-side of an assignment expression is not a defined memory location
- return statement is missing

Techniques for identifying syntactic errors include:

- reading code statements (walkthroughs)
- compiler error messages (compiler output)
- comparing error messages from different compilers - some are more cryptic than others

Semantic Errors

Semantic errors are errors that fail to implement the intent and meaning of the program designer. The more common semantic errors are:

- `=` instead of `==`
- iteration without a body (for/while followed by a semi-colon)
- uninitialized variable
- infinite iteration
- incorrect operator order in a compound expression
- dangling else
- off-by-one iteration
- integer division and truncation
- mismatched data types
- `&` instead of `&&`

Techniques for identifying semantic errors include:

- vocalization - use your sense of hearing to identify the error (compound conditions)
- intermediate output - `printf()` statements at critical stages
- walkthrough table
- interactive debugging using
  - Visual Studio IDE - integrated debugger for Windows OSs
  - Eclipse IDE - integrated debugger for Linux OSs
  - gdb - GNU debugger for gcc

Testing Techniques

The two categories of software testing techniques are:

- black box
- white box

Black Box Tests

The simplest type of test is a black box test. Black box tests are data-driven. We run the executable and treat it as a black box where all internal logic has been hidden from view. External factors alone determine the success or failure of our tests. We test against the specifications. Our tests are input-output driven.

Equivalence Classes

The number of possibilities to be tested in a comprehensive black box test regime is typically too large. To reduce this number to a manageable set, we introduce equivalence classes.

We create equivalence classes using boundary values. An equivalence class is a set where testwise any member is as good as any other (for example, i <1, =1...25, >25).

Experts suggest that semantic errors frequently exist at and on boundaries. We test either side of the boundaries of the equivalence class as well as the boundary itself (for example, i = 0, 1, 2, 17, 24, 25, 26).

We use equivalence classes for both input and output.

White Box Testing

The complementary test to black box tests is a white box test. White box testing is logic-driven. We treat the program as a glass box with all internal logic visible. Each white box test is path-oriented.

In white box testing, we execute each possible path through the code at least once. The number of paths may be too large to test. To reduce this number and still cover all paths through the code at least once, we prepare flow graphs.

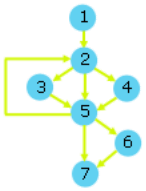
Flow Graphs

A flow graph models the sequences, selections and iterations in the source code. A flow graph consists of nodes and edges. Each node represents one or more sequence statements. Each edge represents the flow of control between two nodes.

Consider the following code:

```c
// Testing - Flow Graph
// flowGraph.c

#include<stdio.h>

int main(void)
{
  int total, value, count;

  // Start Node 1 ---
  total = 0;
  count = 0;
  // End   Node 1 ---

  do {
      // Start Node 2 ---
      scanf("%d", &value);
      // End   Node 2 ---

      if (value < 0)
      {
          // Start Node 3 ---
          total -= value;
          count++;
          // End   Node 3 ---
      }
      else if (value > 0)
      {
          // Start Node 4 ---
          total += value;
          count++;
          // End   Node 4 ---
      }

  // Start Node 5 ---
  } while (value != 0);
  // End   Node 5 ---

  if (count > 0)
  {
      // Start Node 6 ---
      printf("The average value is %.2lf\n",
             (double)total/count);
      // End   Node 6 ---
  }

  // Start Node 7 ---
  return 0;
}
```

The flow graph illustrating the above code would look like:



Test Criteria

To complete a white box test, we apply the following criteria:

- statement coverage - every elementary statement is executed at least once
- edge coverage - every edge is traversed at least once
- condition coverage - all possible values of the constituents of each compound condition are exercised at least once
- path coverage - all paths from the initial node to the final node are traversed at least once.
- iteration coverage
  - skip the iteration entirely
  - pass through the iteration once
  - pass through the iteration less than the specified number of times
  - pass through the iteration the specified number of times
  - pass through the iteration once more than the specified number of times
- compound condition coverage
  - break each compound condition into simple conditions

Debugging Techniques

The tools available for debugging in this course include:

- an integrated development environment (IDE)
- a command-line debugger

Use the source code listed below in the following examples:

```
// Debugging Example
// debug.c

#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>

int main(void)
{
  int total, value, count;

  total = 0;
  count = 0;
  do {
      printf("Enter a value (0 to stop) ");
      scanf("%d", &value);

      if (value < 0)
      {
          total -= value;
          count++;
      }
      else if (value > 0)
      {
          total += value;
          count++;
      }
```

```
  } while (value != 0);

  if (count > 0)
  {
      printf("The average value is %.2lf\n",
             (double)total/count);
  }
  return 0;
}
```

IDE Debugging

Integrated Development Environments (IDEs) are elaborate programs that support text editing, coding, compiling, testing and debugging in a unified application. The IDE used in this course is Microsoft's Visual Studio.

Build and Execute

To build and execute a C program in Visual Studio 2013 (or newer) we:

- Start Visual Studio
- Select New Project
- Select Visual C++ -> Win32 -> Console Application
- Enter Debugging Example as the Project Name | Select OK
- Press Next
- Check Empty Project | Press Finish
- Select Project -> Add New Item
- Select C++ File | Enter debug.c as File Name | Press Add
- Paste in a copy of debug.c (see above)
- Select Build | Build Solution
- Select Debug | Start without Debugging
- Enter 3 at the input prompt
- Enter 2 at the input prompt
- Enter 0 at the input prompt

The input prompts and results of execution appear in a Visual Studio command prompt window.

Tracing

To trace through execution of our program using the built-in debugger we:

- Move the cursor to the left-most column of the total = 0; statement and left-click | This places a red dot in that column, which identifies a breakpoint
- Move the cursor to the left-most column of the closing brace for the do while iteration and left-click | This places a red dot in the column, which identifies another breakpoint
- Move the cursor to the left-most column of the return statement and left-click | This places a red dot in the column, which identifies another breakpoint
- Select Debug -> Start Debugging | Execution should pause at the first breakpoint
- Observe the values under the Locals tab in the Window below the source code
- Press F10 until the input prompt appears and answer the prompt by entering 3
- Observe the values under the Locals tab in the window below the source code
- Press F5, note the position of the arrow identifying the next statement to be executed, and observe the value of total
- Press F5 and answer the prompt by entering a value of 2
- Press F5, note the position of the arrow identifying the next statement to be executed, and observe the value of total
- Press F5 and answer the prompt by entering a value of 0
- Press F5, note the position of the arrow identifying the next statement to be executed, and observe the value of total
- Press F5 and
- Select the command prompt window and observe the program output

- Select the source code window
- Press F5 again to exit

The keystrokes for the various debugging options available in this IDE are listed next to the sub-menu items under the Debug menu.

- F5 continue executing until the next breakpoint
- F10 execute the next statement

Command-Line Debugging

The GNU debugger is a command-line debugging tool called gdb that ships with the gcc compiler for Linux platforms.

Compile and Run

To be able to use gdb, we compile our source code with the -g option:

```
gcc -g myProgram.c
```

To debug the executable (a.out), we enter:

```
gdb a.out
```

The gdb prompt will appear:

```
(gdb)
```

When we start gdb, our program pauses. This is our first opportunity to set breakpoints. We enter the `run` command only once you are ready to execute.

Debugging Commands

The `gdb` commands that we may enter at the prompt include:

- `list` - lists the 10 lines of source code in the vicinity of where execution has stopped. Each call advances the current line by 10
- `list m, n` - where m and n are line numbers - lists lines m through n inclusive of the source code. This call advances the current line to n+1
- `break n` - where n is a line number - sets a breakpoint at line number n
- `clear n` - where n is a line number - clears any breakpoint or trace at line number n
- `delete` - clears all breakpoints
- `run` - starts the execution of your program from line 1
- `print varname` - where varname is a variable name - displays the value of varname
- `cont` - continues execution until either your program ends or encounters a breakpoint
- `step` - executes one line of your program
- `help` - displays the full set of commands available
- `quit` - quits

`gdb` is case-sensitive.

Crashes

If our program crashes and produces a core dump, `gdb` can help locate the crash point. We enter:

```
gdb a.out core
```

and use the following commands:

- **where** - displays the procedure and line number at the time of the crash
- **up** - moves up one procedure in the stack (towards `main()`)
- **down** - moves down one procedure in the stack (away from `main()`)

Help

For online help with a particular command while debugging, we enter:

```
help commmand
```

where *command* is the command in question.

For online help with the `gdb` command while not debugging, we enter:

```
man gdb
```

Walkthrough Table

Walkthroughs are an important technique for understanding the control flow and the memory changes of a source code snippet. A walkthrough emulates the CPU stepping through the code. A walkthrough solution consists of two parts:

- a record of every change in the value of every program variable
- a listing of the output, if any, produced by the program

The record of changes lists all changes that have occurred in RAM during the execution of the program.

When the operating system loads a program into RAM, the program instructions occupy one part of memory while the program variables occupy another part. The operating system transfers control to the program's first instruction. The program executes one instruction at a time until it returns control to the operating system. Some instructions accept input from the user, some change the values stored in the program variables, and others send output to the user.

To track each change in RAM, we construct a table of the program variables. We list their identifiers and their types across its top line and enter their values in the rows below. We insert mock addresses below the identifiers, picking convenient address values; the actual addresses do not matter here. In other words, our walkthrough table is a simplified representation of RAM throughout the program's lifetime.

Consider the program below:

```c
// Walkthrough
// walkthrough.c

#include <stdio.h>
#define ADULT_FARE 3.25

int main(void)
{
  int riders;
  double total;

  printf("Number of riders : ");
  scanf("%d", &riders);

  total = riders * ADULT_FARE;
  printf("Total fare is %.2lf\n\n", total);

  printf("riders' address %x\n", &riders);
  printf("total's address %x\n", &total);
```

```
  return 0;
}
```

The output of the program above displays the following:

```
Number of riders : 3
Total fare is 9.75

riders' address 0xbf9cf5bc
total's address 0xbf9cf5b0
```

The instructions part of the table is optional and may be replaced by the line numbers corresponding to these instructions. The walkthrough table is shown below:

A style that is sufficient for programs discussed in this set of notes is shown below. Note that the table header includes the type and the address of each variable.

Example

Consider the following code:

```
// by Evan Weaver

int main(void)
{
    int a;
    double b, c;

    a = 6;
    b = 0.7;

    while (a < 10 && b < 3.0
    {
        if (a < 8)
        {
            a = a + 1;
            b = b * 2;
            c = a - b; // careful: mixed types
        }
        else
        {
            a = a - 2;
            b = b + 0.8;
        }

        c = a - b;
        printf("%.2lf-%d-%.2lf\n", c, a, b);
    }
}
```

We prepare the walkthrough table by:

- inserting the name of the program (done)
- showing the type of each variable (done)
- showing each variable's identifier (done)
- showing a unique address for each variable (done)

Complete the rest of this table as an exercise!

On this page

Arrays

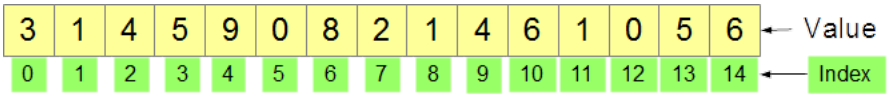
Learning Outcomes

After reading this section, you will be able to:

- Design data collections using arrays to manage information efficiently
- Introduce character strings as terminated collections of byte information

Introduction

Programs can process extremely large amounts of data much faster than well-established manual techniques. Whether this processing is efficient or not depends in large part on how that data is organized. For example, large collections of data can be organized in structures if each variable shares the same type with all other variables and the variables are stored contiguously in memory. Not only can structured data be processed efficiently but the programming of tasks performed on structured data can be simplified considerably. Instead of coding a separate instruction for each variable, we code the instruction that is common to all variables and apply that instruction in an iteration across the data structure.



The simplest data structure in the C language is a list of variables of the same type. We call such a list an array and the variables in that array its elements. We refer to any element by its index.

This chapter introduces the syntax for defining arrays, initializing them and accessing their elements directly. The chapter also demonstrates how to construct a table of values using the concept of a parallel array. This chapter concludes by introducing character strings as arrays with a special terminator.

Definition

An array is a data structure consisting of an ordered set of elements of common type that are stored contiguously in memory. Contiguous storage is storage without any gaps. An array definition takes the form

```
type identifier [ size ];
```

`type` is the type of each element, `identifier` is the array's name, the brackets `[ ]` identify the data structure as an array and `size` specifies the number of elements in the array.

For example, to define an array of 8 grades, we write:

```
int grade[8];
```

This statement allocates contiguous storage in RAM for an array named grade that consists of 8 int elements.

We can specify the size of the array using `#define` or `const int`:

```
#define NGRADES 8  // or const int NGRADES = 8;

int grade[NGRADES];
```

```
// ... record the grades

printf("Your last grade was %d\n", grade[NGRADES - 1]);
printf("Your second last grade was %d\n", grade[NGRADES - 2]);
```

This coding style facilitates modifiability. If we change the size, we need to do so in only one place.

Elements

Each element has a unique index and holds a single value. Index numbering starts at 0 and extends to one less than the number of elements in the array. To refer to a specific element, we write the array name followed by bracket notation around the element's index.

```
identifier[index]
```

For example, to access the first element of grade, we write:

```
grade[0]
```

To display all elements of grade, we iterate:

```
for (int i = 0; i < NGRADES; i++)
{
    printf("%d" , grade[i]);
}
```

Check Array Bounds

C compilers do not introduce code that checks whether an element's index is within the bounds of its array. It is our responsibility as programmers to ensure that our code does not include index values that point to elements outside the memory allocated for an array.

Initialization

We can initialize an array when we define it in the same way that we initialize variables. We suffix the declaration with an assignment operator followed by the set of initial values. We enclose the values in the set within a pair of braces and separate them with commas. Initialization takes the form:

```
type identifier[ size ] = { value, ... , value };
```

For example, to initialize grade, we write:

```
int grade[NGRADES] = {10,9,10,8,7,9,8,10};
```

If our initialization fills all elements in the array, C compilers infer the size of the array from the initialization set and we do not need to specify the size between the brackets. We may simply write:

```
int grade[] = {10,9,10,8,7,9,8,10};
```

If we specify fewer initial values than the size of the array, C compilers fill the uninitialized elements with zero values:

```
int grade[NGRADES] = {0};
```

This will initializes all 8 elements of `grade` to zero.

Specifying a size that is less than the number of initial values generates a syntax error.

Parallel Arrays

A convenient way to store tabular information is through two parallel arrays. One arrays holds the key, while the other hold values. The arrays are parallel because the elements at the same index hold data that are related to the same entity.

In the following example, `sku[i]` holds the stock keeping unit (sku) for a product, while `price[i]` holds its unit price.

```
// Parallel Arrays
// parallel.c

#include <stdio.h>

int main(void)
{
    int i;
    int sku[]      = { 2156, 4633, 3122, 5611};
    double price[] = { 2.34, 7.89, 6.56, 9.32};
    const int n    = 4;

    printf("  SKU Price\n");
    for (i = 0; i < n; i++)
    {
        printf("%5d $%.2lf\n", sku[i], price[i]);
    }

    return 0;
}
```

Output of the above program:

```
  SKU Price
 2156 $2.34
 4633 $7.89
 3122 $6.56
 5611 $9.32
```

The `sku[]` array holds the key data, while the `price[]` array holds the value data. Note how the elements of parallel arrays with the same index make up the fields of a single record of information.

Parallel arrays are simple to process. For example, once we find the index of the element that matches the specified sku, we also have the index of the unit price for that element.

Character Strings

The topic of character strings is covered in depth in the chapter entitled Character Strings. The following section introduces this topic at a high level.

Introduction

A ***string*** is a `char` array with a special property: a **terminator element** follows the last meaningful character in the string. We refer to this terminator as the ***null terminator*** and identify it by the escape sequence `'\0'`.

The null terminator has the value 0 on any host platform (in its collating sequence). All of its bits are 0's. The null terminator occupies the first position in the ASCII and EBCDIC collating sequences.

The value of the index identifying the null terminator element is the number of meaningful characters in the string.

The number of memory locations occupied by a string is one more than the number of meaningful characters in the string.

Syntax

We need to allocate memory for one additional byte to provide room for the null terminator:

```
const int NCHAR = 17;
char name[NCHAR + 1] = "My Name is Arnold";
```

We use the `"%s"` conversion specifier and the address of the start of the character string to send its contents to standard output:

```
printf("%s", name);
```

Formatting and handling syntax is covered later.

On this page

Structures

Learning Outcomes

After reading this section, you will be able to:

- Design data collections using structures to manage information efficiently

Introduction

The most commonly used data structure in C language programs aside from the array is the struct or structure. A structure type is a collection of not necessarily identical types. We use the structure type to define a group of variables as a single object.

This chapter reviews the primitive types and presents the syntax for declaring a structure type, defining an object of structure type, and accessing the data values within that object. This chapter includes an example of how to walkthrough a program that includes structure types.

Types

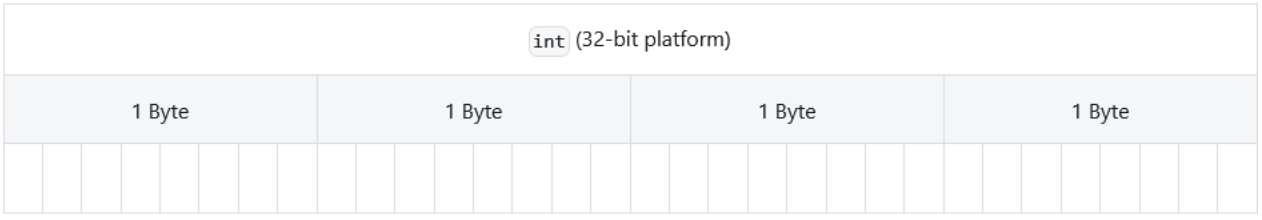
A type describes how to interpret the information stored in a region of memory. In the C language, a type may be a primitive type or a derived type. A derived type is a collection of other types.

Primitive Types

The core language defines the primitive types. We cannot redefine these types or introduce new primitive types. The C language primitive types include:

- `char`
- `int`
- `float`
- `double`

Each type defines how a value of that type is stored in a region of memory. Consider the `int` type. A value of `int` type is stored in equivalent binary representation in 4 bytes on a 32-bit platform:



To define an object of `int` type called `noSubjects`, we write:

```
int noSubjects;
```

Derived Types

The declaration of a derived type in the C language takes the form

```
struct Tag
{
    //... declarations here
};
```

where the keyword `struct` identifies a derived type or structure. `Tag` is the name by which we call the structure (just like `int` above). The declaration concludes with a semicolon.

We list the types that belong to the structure along with their identifiers within the curly braces.

```
struct Tag
{
    // [type] [identifier];
    // ... other types
};
```

`type` is the member's type. `identifier` is the name by which we access the member's value.

Example

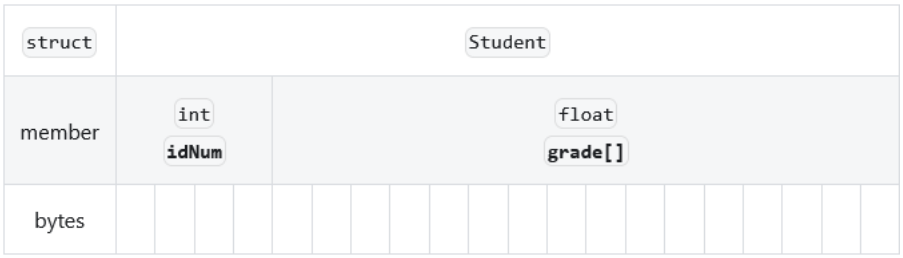
Consider a structure type that consists of two pieces of information:

- the student's ID number
- the student's grades (up to 4 individual grades)

Let us call this structure type `Student`. To declare the type, we write:

```
struct Student
{
    int idNum;       // student number
    float grade[4];  // grades
};
```

The members occupy memory in the order in which we have listed them in the declaration of our structure:



> **NOTE**
>
> This declaration does not allocate any memory for any object; it only defines the structure and the rules for objects of that type.

Declaration

We declare our structure globally and may store its declaration in a separate file called a header file (say, with the name `Student.h`):

```
// Student.h

struct Student
{
    int idNum;       // student number
```

```
    float grade[4];  // grades
};
```

When we place source code in a header (`.h`) file, we insert that header file's code into the source file that requires that information, as shown below. In such cases, our complete source code is stored in more than one file. When compiling multi-file source code, we only pass the `.c` file(s) to the compiler. The code in a header file is duplicated inside each C file in which it is included, which allows us to write code, like a `struct`, in one spot and edit it in that one spot alone.

Allocating Memory

When we define an object of a structure, we allocate memory for that object. Our definition takes the form:

```
struct Tag identifier;
```

where `Tag` is the name of the structure and `identifier` is the name of the object.

Example

To allocate memory for a `Student` named `harry`, we write:

```
// main.c

#include "Student.h" // includes the description of a Student

int main(void)
{
    struct Student harry; // allocates memory for harry

    // ...

    return 0;
}
```

The object name `harry` refers to the collection of members in `Student harry` taken together.

Initialization

To initialize an object of a structure we add a braces-enclosed, comma-separated list of values. We organize the initial values in the same order as the member listing in the declaration of the structure. The initialization takes the form:

```
struct Tag identifier = { value, ... , value };
```

> **NOTE**
>
> Structure initialization is similar to one of an array.

Example

To initialize `harry` with student number `975` and grades of `75.6`, `82.3` and `68.9`, we write:

```
struct Student harry = { 975, { 75.6f, 82.3f, 68.9f } };
```

Member Access

To access a member of an object of a structure, we use the dot operator (`.`). Dot notation takes the form:

```
object.member
```

To access `harry`'s student number, we write:

```
harry.idNum
```

To retrieve the address of a non-array member of an object, we use the address of operator (`&`):

```
&instance.member
```

To access the address of `harry`'s student number, we write:

```
&harry.idNum
```

> **NOTE**
>
> We may omit the parentheses here - `&(harry.idNum)` - they are unnecessary because the dot (`.`) operator binds tighter than the address-of operator (see the precedence table).

To access an array member, we refer to its name without brackets. For example, to access the address of `harry`'s grades, we write:

```
harry.grade
```

To access an element of an array member, we use subscript notation

```
object.member[index]
```

To access `harry`'s ***third*** grade, we write:

```
harry.grade[2]
```

To retrieve the address of an element of an array member, we use the address of operator (`&`):

```
&object.member[index]
```

To access the address of `harry`'s ***third*** grade, we write:

```
&harry.grade[2]
```

Example

A convenient alternative to parallel arrays for storing tabular information is an array of structures. One member holds the key, while the other member holds the data.

In the following example, the `sku` member holds the stock keeping unit (sku) for a product, while `price` holds its unit price. The header file with the declaration of the `Product` structure contains:

```
// Structure Example
// product.h

struct Product
{
    int sku;
    double price;
};
```

The program that uses the `Product` structure is listed below.

```
// Structure Example
// structure.c

#include <stdio.h>
#include "product.h"

int main(void)
{
    int i;
    struct Product product[] = { {2156, 2.34}, {4633, 7.89},
                                 {3122, 6.56}, {5611, 9.32} };
    const int n = 4;

    printf("  SKU Price\n");
    for (i = 0; i < n; i++)
    {
        printf("%5d $%.2lf\n", product[i].sku, product[i].price);
    }

    return 0;
}
```

The output produced from the above sample is shown below:

```
  SKU Price
 2156 $2.34
 4633 $7.89
 3122 $6.56
 5611 $9.32
```

Walkthrough

A walkthrough table for a program with structure types includes lists of the member types below the object identifiers. The table for the example above is shown below.

The table includes:

- the structure type of each object
- the identifier of each object
- the type of each member
- the identifier of each member

> **NOTE**
>
> Each object is broken down into its members in the head of the table. We reserve a separate line for the addresses of the different objects:

Output:

On this page

Functions

Learning Outcomes

After reading this section, you will be able to:

- Design procedures using selection and iteration constructs to solve a programming task
- Connect procedures using pass-by-value semantics to build a complete program
- Trace the execution of a complete program to validate its correctness

Introduction

Procedural programming involves separating source code into self-contained components that can be accessed multiple times from different locations in a complete program. This approach enables separate coding of each component and assembly of various components into a complete program. We call this approach to programming solutions modular design.

This chapter introduces the principles of modular design, describes the syntax for defining a module in the C language, shows how to pass data from one module to another, suggests a walkthrough table structure for programs composed of several modules and includes an example that validates user input.

Modular Design

Modular design identifies the components of a programming project that can be developed separately. Each module consists of a set of logical constructs that are related to one another. A module may refer to other modules. A trivial example is the program described in the chapter on compilers:

```c
/* My first program
   hello.c          */

#include <stdio.h>           // information about the printf identifier

int main(void)               // program startup
{
    printf("This is C"); // output

    return 0;            // return to operating system
}
```

The module named `hello.c` starts executing at statement `int main(void)`, outputs a string literal and returns control to the operating system.

- `main()` transfers control to a module named `printf()`
- `printf()` executes the detailed instructions for outputting the string literal
- `printf()` returns control to `main()`

Design Principles

We can sub-divide a programming project in different ways. We select our modules so that each one focuses on a narrower aspect of the project. Our objective is to define a set of modules that simplifies the complexity of the original problem.

Some general guidelines for defining a module include:

1. the module is easy to upgrade

2. the module contains a readable amount of code
3. the module may be used as part of the solution to some other problem

For a structured design, we stipulate that:

1. each module has one entry point and one exit point
2. each module is highly cohesive
3. each module exhibits low coupling

Cohesion

Cohesion describes the focus: a highly cohesive module performs a single task and only that task.

In creating a cohesive module, we ask whether our tasks belong to that module: a reason to include a task is its relation to the other tasks within the module. A reason to exclude a task is its independence from other tasks within the module.

For example, the following tasks are related:

- receives a date and the number of days to add
- converts the date into a format for adding days
- adds the number of days received
- converts the result to a new date
- returns the new date

The following tasks are unrelated:

- calculates Federal tax on bi-weekly payroll
- calculates the value of π
- outputs an integer in hexadecimal format

We allocate unrelated tasks to separate modules in the program design.

Coupling

Coupling describes the degree of interrelatedness of a module with other modules. The less information that passes between the module and the other modules the better the design. We prefer designs in which each module completely control its own computations and avoids transferring control data to any other module.

Consider a module that receives a flag from another module and performs a calculation based on that flag. Such a module is highly coupled: another module controls its execution. To improve our design, we transfer data to the module and let it create its own flags before completing its task.

Functions

The C language is a procedural programming language. It supports modular design through function syntax. Functions transfer control between one another. When a function transfers control to another function, we say that it ***calls*** the other function. Once the other function completes its task and transfers control to the caller function, we say that that other function ***returns*** control to its ***caller***.

In the example from the introductory chapter on compilers listed above:

1. the `main()` function calls the `printf()` function
2. the `printf()` function outputs the string
3. the `printf()` function returns control to its caller `main()`

Definition

A function consists of a header and a body. The body is the code block that contains the detailed instructions to be performed by the function. The header immediately precedes the body and includes the following ***in order***:

1. the type of the function's return value
2. the function's identifier
3. a parentheses-enclosed list of parameters that receive data from the caller

```
type identifier(type parameter, ..., type parameter)
{
   // function instructions

   return x; // x denotes the value returned by this function
}
```

`type` specifies the type of the return value or the function's parameter, while the `identifier` specifies the name of the function, and `parameter` is a variable that holds data received from the caller function.

For example:

```c
/* Raise an integer to an integer
 * power.c
 */

#include <stdio.h>

int power(int base, int exponent)
{
    int i, result;

    result = 1;
    for (i = 0; i < exponent; i++)
        result = result * base;

    return result;
}

int main(void)
{
    int base, exp, answer;

    printf("Enter base : ");
    scanf("%d", &base);
    printf("Enter exponent : ");
    scanf("%d", &exp);

    answer = power(base, exp);

    printf("%d^%d = %d\n", base, exp, answer);
}
```

Outputs the following:

```
Enter base : 3
Enter exponent : 4
3^4 = 81
```

The first function returns a value of `int` type, while `power` identifies the function, and `base` and `exponent` are the function's parameters; both are of `int` type.

Special Cases

void Functions

A function that does not have to return any value has no return type. We declare its return type as `void` and exclude any expression from the return statement. For example:

```
void countDown(int n)
{
    while (n > 0)
    {
        printf("%d ", n);
        n--;
    }

    return; // optional
}
```

> **NOTE**
>
> In such cases, the return statement is **optional** and is usually not included.

No Parameters

A function that does not have to receive any data does not require parameters. We insert the keyword `void` between the parentheses. For example:

```
void alphabet(void)
{
    char letter = 'A';

    do {
            printf("%d ", letter);
            letter++;
    } while (letter != 'Z');
}
```

> **NOTE**
>
> The iteration changes `letter` to the next character in the alphabet, assuming the collating sequence arranged them contiguously.

main

The `main()` function is a function itself. It is the function to which the operating system transfers control after loading the program into RAM.

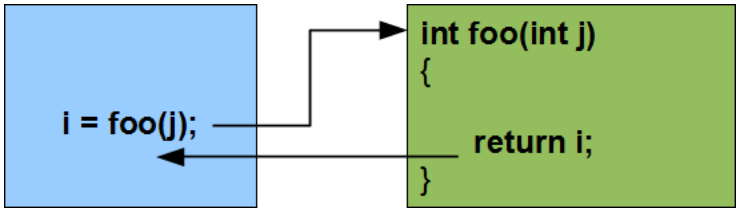
`main()` returns a value of `int` type to the operating system once it has completed execution. A value of 0 indicates success to the operating system.

Function Calls

A function call transfers control from the caller to function being called. Once the function being called has executed its instructions, it returns control to the caller. Execution continues at the point immediately following the call statement. A function call takes the form:

```
identifier(argument, ..., argument)
```

`identifer` specifies the function being called, while `argument` specifies a value being passed to the function being called.



An argument may be a constant, a variable, or an expression (with certain exceptions). The number of arguments in a function call should match the number of parameters in the function header.

Pass By Value

The C language passes data from a caller to a function by value. That is, it passes a copy of the value and not the value itself. The value passed is stored as the inital value is the parameters that corresponds to the argument in the function call.

Each parameter is a variable with its own memory location. We refer to the mechanism of allocating separate memory locations for parameters and using the arguments in the function call to initialize these parameters as ***pass by value***. Pass by value facilitates modular design by localizing consequences. The function being called may change the value of any of its parameters many times, but the values of the corresponding arguments in the caller remain unchanged. In other words, a function cannot change the value of an argument in the call to the function. This language feature ensures the variables in the caller are relatively secure.

Mixing Types

If the type of an argument does not match the type of the corresponding parameter, the compiler coerces (narrows or promotes) the value of the argument into a value of the type of the corresponding parameter. Consider the `power` function listed above and the following call to it:

```
int answer;
answer = power(2.5, 4);
```

The compiler converts the first argument into a value of type `int`

```
int answer;
answer = power((int)2.5, 4);
```

The C compiler evaluates the cast (coercion) of 2.5 before passing the value of type `int` to `power` and initializing the parameter to the cast result (2).

Walkthroughs

The structure of a walkthrough table for a modular program is a simple extension of the structure of the walkthrough table shown in the chapter entitled Testing and Debugging. The table for a modular program groups the variables under their parent functions.

int			type		
main(void)			--function identifier here--		
type	...	type	type	...	type
variable z	...	variable a	variable z	...	variable a
&z	...	&a	&z	...	&a
initial value	...	initial value	initial value	...	initial value

int			type		
main(void)			--function identifier here--		
next value	...	next value	next value	...	next value
next value	...	next value	next value	...	next value
			next value	...	next value
initial value	...	initial value	initial value	...	initial value
next value	...	next value	next value	...	next value
next value	...	next value	next value	...	next value
			next value	...	next value
			next value	...	next value

Output:

(record output here line by line)

Example

The completed walkthrough table for the power.c program listed above is shown below:

- **& represents the address.**

int			int			
main(void)			power(int base, int exponent)			
int	int	int	int	int	int	int
base	exp	answer	base	exponent	result	i
&100	&104	&108	&10C	&110	&114	&118
3						
3	4					
3	4		3	4		
3	4		3	4	1	
3	4		3	4	1	0
3	4		3	4	3	0
3	4		3	4	3	1
3	4		3	4	9	1

int			int			
main(void)			power(int base, int exponent)			
3	4		3	4	9	2
3	4		3	4	27	2
3	4		3	4	27	3
3	4		3	4	81	3
3	4		3	4	81	4
3	4	81				

> **NOTE**
>
> Each parameter occupies a memory location that is distinct from any other location in the caller. For example, the parameter `base` in `power()` occupies a different memory location than the variable `base` in `main()`.

Validation (optional)

Ensuring that programming assumptions are not breached by the user is part of good program design. Our function to raise an integer base to the power of an exponent is based on the assumption that the exponent is non-negative. Accordingly, we need to validate the user input to ensure that our assumption holds.

Let us introduce a function named `getNonNegInt()` that only accepts non negative integer values from the user:

```c
/* Raise an Integer to the Power of an Integer
 * power.c
 */

#include <stdio.h>

// getNonNegInt returns a non-negative integer
//
// getNonNegInt assumes that the user enters only
// integer values and no trailing characters
//
int getNonNegInt(void)
{
    int value;

    do {
            printf(" Non-negative : ");
            scanf("%d", &value);
            if (value < 0)
                    printf(" * Negative! *\n");
    } while(value <= 0);

    return value;
}

// power returns the value of base raised to
//  the power of exponent (base^exponent)
//
//  power assumes that base and exponent are
//  integer values and exponent is non-negative
//
int power(int base, int exponent)
```

```
{
    int result, i;

    result = 1;
    for (i = 0; i < exponent; i++)
            result = result * base;

    return result;
}

int main(void)
{
    int base, exp, answer;

    printf("Enter base : ");
    scanf("%d", &base);

    printf("Enter exponent\n");
    exp = getNonNegInt();
    answer = power(base, exp);

    printf("%d^%d is %d\n", base, exp, answer);

    return 0;
}
```

Code Output

```
Non-negative : -2
* Negative! * //error message from do-while
Non-negative : 4

Enter base : 3
Enter exponent //exponent becomes "4"

3^4 is 81
```

Walkthrough

The table below lists the values of the local variables in this source file at different stages of execution:

int			int				int
main(void)			power(int base, int exponent)				getNonNegInt(void)
int	int	int	int	int	int	int	int
base	exp	answer	base	exponent	result	i	value
?	?	?					
3	?	?					
3	?	?					-2
3	?	?					4
3	4	?					

int			int				int
main(void)			power(int base, int exponent)				getNonNegInt(void)
3	4	?	3	4	?	?	
3	4	?	3	4	1	?	
3	4	?	3	4	1	0	
3	4	?	3	4	3	0	
3	4	?	3	4	3	1	
3	4	?	3	4	9	1	
3	4	?	3	4	9	2	
3	4	?	3	4	27	2	
3	4	?	3	4	27	3	
3	4	?	3	4	81	3	
3	4	?	3	4	81	4	
3	4	81					

The shaded areas show the stages in their lifetimes at which the variables are visible. The unshaded areas identify the stages at which the variables are out of scope of the function that has control. The values marked ? are undefined.

On this page

Pointers

Learning Outcomes

After reading this section, you will be able to:

- Design procedures using selection and iteration constructs to solve a programming task
- Connect procedures using pass-by-address semantics to build a complete program
- Trace the execution of a complete program to validate its correctness

Introduction

Programming languages set different rules for passing data from one module to another. The C programming language was designed from the outset to safeguard data in each module from corruption by another module. The language's pass-by-value mechanism prevents one function from making any direct change to any variable outside that function. A function's parameters receive copies of its caller's arguments so that any changes that the function makes to the parameter values only affect those copies. The calling function's arguments remain unaltered.

Cases arise that require changing the value of an external variable from within a function. The C language enables this through the variable's address.

This chapter describes how to receive the address of a variable in a function parameter, how to change the value stored in that address from within the function and how to walkthrough code that accesses addresses.

Addresses

Every program variable occupies a unique address in memory throughout its lifetime. The 'address of' operator (`&`) applied to a variable's identifier evaluates to the address of that variable in memory.

The following program fills the address of `x` (written as `&x`) with user supplied input. The program then displays the value stored and its address in memory:

```c
/* Working with Addresses
 * addresses.c
 */

#include <stdio.h>

int main(void)
{
    int x;

    printf("Enter x : ");
    scanf("%d", &x);
    printf("Value stored in x   :%d\n", x);
    printf("Address of x        :%x\n", &x);

    return 0;
}
```

The above program produces the following output:

```
Enter x : 45
Value stored in x   :45
Address of x        :12f9a0
```

`%x` is the conversion specifier for integer output in hexadecimal format.

Pointer

A variable that holds an address is called a pointer. To store a variable's address, we define a pointer of the variable's type and assign the variable's address to that pointer. A pointer definition takes the form:

```
type *identifier;
```

`type *` is the type of the pointer, and `identifier` is the name of the pointer.

The `*` operator stands for 'data at address' or simply 'data at' and is called the *dereferencing* or *indirection* operator. This operator applied to a pointer's identifier evaluates to the value in the address that that pointer holds.

The following program stores the address of variable `x` in pointer `p` and displays the value in that address using the pointer `p`:

```c
/* Working with Pointers
 * pointers.c
 */

#include <stdio.h>

int main(void)
{
    int x;
    int *p = &x; // store address of x in p

    printf("Enter x : ");
    scanf("%d", &x);
    printf("Value stored in x : %d\n", *p);
    printf("Address of x      : %x\n", p);

    return 0;
}
```

The above program produces the following output:

```
Enter x : 45
Value stored in x : 45
Address of x      : 3cf760
```

Pointer Types

The C language supports a pointer type for every primitive or derived type:

Type	Pointer Type
`char`	`char *`
`short`	`short *`
`int`	`int *`
`long`	`long *`
`long long`	`long long *`

Type	Pointer Type
`float`	`float *`
`double`	`double *`
`long double`	`long double *`
`struct Product`	`struct Product *`

C compilers typically store addresses in 4 bytes of memory.

NULL Address

Each pointer type has a special value called its null value. The constant `NULL` is an implementation defined constant that contains this null value (typically, 0). This constant is defined in the `<stdio.h>` and `<stddef.h>` header files.

It is good style to initialize the value of a pointer to `NULL` before the address is known. For example:

```
int *p = NULL;
```

Parameters

A function can receive in its parameters not only data values but also addresses of program variables.

Pass-by-Value

Consider a function named `internal_swap()` that swaps the values stored in two memory locations. We call this function from `main()` and note that the swap remains completely within the function itself:

```
/* Internal swap
 *  internal_swap.c
 */

#include <stdio.h>

void internal_swap (int a, int b)
{
    int c;

    printf("a is %d, b is %d\n", a, b);

    c = a;
    a = b;
    b = c;

    printf("a is %d, b is %d\n", a, b);
}

int main(void)
{
    int a, b;

    printf("a is ");
    scanf("%d", &a);
    printf("b is ");
    scanf("%d", &b);
```

```
    internal_swap(a, b);

    printf("After internal_swap:\n"
           "a is %d\n"
           "b is %d\n", a, b);

    return 0;
}
```

The above program produces the following output:

```
a is 5
b is 6
a is 5, b is 6
a is 6, b is 5
After internal_swap:
a is 5
b is 6
```

> **INFO**
>
> Although `internal_swap()` does exchange the values in variables `a` and `b`, the pass-by-value mechanism preserves the original values in `main().`

Walkthrough Table

The walkthrough table shows how the changes remain completely within `internal_swap()` function scope:

`void`			`int`	
`local_swap(int a, int b)`			`main(void)`	
`int`	`int`	`int`	`int`	`int`
`a`	`b`	`c`	`a`	`b`
`0x0012FF78`	`0x0012FF7C`	`0x0012FF6C`	`0x0012FF88`	`0x0012FF84`
			`5`	`6`
`5`	`6`	?	`5`	`6`
`5`	`6`	`5`	`5`	`6`
`6`	`6`	`5`	`5`	`6`
`6`	`5`	`5`	`5`	`6`
`6`	`5`	`5`	`5`	`6`

The hexadecimal values below the variable identifiers are their addresses in memory.

> **NOTE**
>
> The addresses of `a` and `b` in `internal_swap()` are different from those in `main()`.

The program ***copies*** the argument values (`a` and `b`) as initial values into parameters `a` and `b`. The swapping only affects `a` and `b` in the `internal_swap()` function.

Pass-by-Address

To change the original values, we pass the addresses of their variables instead of their values. We use these addresses to access the original values and change them from within the function.

Consider the following program:

```c
/* Swapping values using a function
 *  swap.c
 */

#include <stdio.h>

void swap(int *p, int *q)
{
    int c;

    c = *p;
    *p = *q;
    *q = c;
}

int main(void)
{
    int a, b;

    printf("a is ");
    scanf("%d", &a);
    printf("b is ");
    scanf("%d", &b);

    swap(&a, &b);

    printf("After swap:\n"
           "a is %d\n"
           "b is %d\n", a, b);

    return 0;
}
```

The above program produces the following output:

```
a is 5
b is 6
After swap:
a is 6
b is 5
```

Walkthrough Table

The walkthrough table shows how the changes carry over to `main()`:

`void`			`int`	
`swap(int *p, int *q)`			`main(void)`	
`int *`	`int *`	`int`	`int`	`int`
`p`	`q`	`c`	`a`	`b`
`0x0012FF78`	`0x0012FF7C`	`0x0012FF6C`	`0x0012FF88`	`0x0012FF84`

			`5`	`6`
`0x0012FF88`	`0x0012FF84`	?	`5`	`6`
`0x0012FF88`	`0x0012FF84`	`5`	`5`	`6`
`0x0012FF88`	`0x0012FF84`	`5`	`6`	`6`
`0x0012FF88`	`0x0012FF84`	`5`	`6`	`5`
`0x0012FF88`	`0x0012FF84`	`5`	`6`	`5`

Some programmers prefer symbolic notation instead of address values. For example, they use the symbol `main::a` to refer to the local variable `a` in the function `main()`. A walkthrough table using symbolic notation looks something like this:

`void`			`int`	
`swap(int *p, int *q)`			`main(void)`	
`int *`	`int *`	`int`	`int`	`int`
`p`	`q`	`c`	`a`	`b`
`0x0012FF78`	`0x0012FF7C`	`0x0012FF6C`	`0x0012FF88`	`0x0012FF84`
			`5`	`6`
`main::a`	`main::b`	?	`5`	`6`
`main::a`	`main::b`	`5`	`5`	`6`
`main::a`	`main::b`	`5`	`6`	`6`
`main::a`	`main::b`	`5`	`6`	`5`
`main::a`	`main::b`	`5`	`6`	`5`

Multiple Return Values

C function syntax only allows for the `return` of a single value. If program design requires a function that returns more than one value, we do so through ***parameter pointers*** that hold the addresses of the variables that receive the multiple return values.

The following program converts day of year to month and day of month by calling function `day_to_dm()`:

```
/* Day of Year to Day of Month and Month
 * day_to_dm.c
 */

#include <stdio.h>

// day_to_dm return day and month of given day in year
// assumes not leap year
//
```

```c
void day_to_dm(int day, int *d, int *m)
{
    if (day < 32)
    {
        *m = 1;
        *d = day;
    }
    else if (day < 60)
    {
        *m = 2;
        *d = day - 31;
    }
    else if (day < 91)
    {
        *m = 3;
        *d = day - 59;
    }
    else if (day < 121)
    {
        *m = 4;
        *d = day - 90;
    }
    else if (day < 152)
    {
        *m = 5;
        *d = day - 120;
    }
    else if (day < 182)
    {
        *m = 6;
        *d = day - 151;
    }
    else if (day < 223)
    {
        *m = 7;
        *d = day - 181;
    }
    else if (day < 254)
    {
        *m = 8;
        *d = day - 222;
    }
    else if (day < 284)
    {
        *m = 9;
        *d = day - 253;
    }
    else if (day < 305)
    {
        *m = 10;
        *d = day - 283;
    }
    else if (day < 335)
    {
        *m = 11;
        *d = day - 304;
    }
    else if (day < 366)
    {
        *m = 12;
        *d = day - 334;
    }
}

int main(void)
{
    int day, d, m;
```

```
    printf("Day of Year : ");
    scanf("%d", &day);

    day_to_dm(day, &d, &m);

    printf("Day/Month is %d/%d\n", d, m);

    return 0;
}
```

The above program produces the following output:

```
Day of Year : 357
Day/Month is 23/12
```

> **DESIGN CONSIDERATION:**
>
> A function that returns values through the parameters can reserve its single `return` value for reporting an error code produced by the function.

On this page

Functions, Arrays and Structs

Learning Outcomes

After reading this section, you will be able to:

- Design procedures using selection and iteration constructs to solve a programming task
- Connect procedures using pass-by-value and pass-by-address semantics to build a complete program
- Design data collections using arrays and structures to manage information efficiently
- Trace the execution of a complete program to validate its correctness

Introduction

Procedural programming scopes information. Scoping is an essential feature of modular design. A complete program consists of a variety of scopes. Each module limits the visibility of the program data and instructions in that module. Each code block limits the visibility of the data in that block.

This chapter describes how to identify a function type, describes the different scopes within a program, describes the passing of arrays and structures to a function, lists style guidelines for function coding and provides a sample walkthrough with functions, pointers and structures.

Prototypes

A function prototype identifies a function type. It provides the information that the compiler requires to validate a function call. The prototype is similar to the function header. A prototype takes the following form

```
type identifier(type [parameter], ..., type [parameter]);
```

A prototype ends with a semi-colon and may exclude the parameter identifiers. The identifier, the return type, and the parameter types are sufficient to validate a function call. The parameter types determine any coercion that may be necessary in passing values to the function.

For example, the prototype for our power() function in the chapter entitled Functions is:

```
int power(int, int);
```

We insert prototypes near the head of our source file and before any function calls. Once we have declared the prototype for each function in a source file, we can arrange our function definitions in any order.

For example:

```
/* Raise an integer to the power of a integer
 * power.c
 */

#include <stdio.h>

int power(int base, int exponent);

int main(void)
{
    int base, exp, answer;
```

```
    printf("Enter base : ");
    scanf("%d", &base);
    printf("Enter exponent : ");
    scanf("%d", &exp);

    answer = power(base, exp);
    printf("%d^%d = %d\n", base, exp, answer);

    return 0;
}

int power(int base, int exponent)
{
    int result, i;

    result = 1;
    for (i = 0; i < exponent; i++)
    {
        result = result * base;
    }

    return result;
}
```

The compiler interprets the call to `power()` in the `main()` function as a valid call based on the prototype (not the header in the function definition below the call).

include

We can define a function used by our host application in a file separate from the source file of our application. If we do, we also store its prototype in a separate file called a header file. Typically, this header file has the extension `.h`. We insert the contents of the header file into the source file of our application using a `#include` directive.

The `#include` directive takes either of two forms:

```
#include <filename>  // filename is in the system directories
#include "filename"  // filename is in the current directory
```

The compiler searches for the header file in the current directory, the system directory or both and, if found, inserts the contents of the file in place of the directive.

stdio.h

The header file that contains the prototypes for the `printf()` and `scanf()` functions is called `stdio.h` and is stored in a **system directory**.

We include this header file in our source code whenever we call either of these functions:

```
#include <stdio.h>

int main(void)
{
    printf("This is C\n");

    return 0;
}
```

Current Directory(Optional)

We may store the prototype for our `power()` function in its own **header file (.h)** and insert that header file in our source code. For example:

```c
/* Raise an integer to the power of a integer
 * power.h
 */

int power(int base, int exponent);
```

The corresponding **implementation file (.c)** will look like:

```c
/* Raise an integer to the power of a integer
 * power.c
 */

#include <stdio.h>

#include "power.h"

int main(void)
{
    int base, exp, answer;

    printf("Enter base : ");
    scanf("%d", &base);
    printf("Enter exponent : ");
    scanf("%d", &exp);

    answer = power(base, exp);
    printf("%d^%d = %d\n", base, exp, answer);

    return 0;
}

int power(int base, int exponent)
{
    int result, i;

    result = 1;
    for (i = 0; i < exponent; i++)
    {
        result = result * base;
    }

    return result;
}
```

Scope

The scope of a program identifier determines its visibility. Its scope depends on where we have placed its definition. A program variable may have

- global scope
- function scope
- local scope
- block scope

Global Scope

A variable **declared outside all function definitions** has *global scope*. We call such a variable a *global variable*. Any program instruction can access a global variable. The compiler allocates memory for a global variable alongside the string literals at startup and releases that memory at termination; that is, after the having executed the `return` statement of `main()`.

Global variables introduce a high degree of coupling. For instance, if we change the name of the variable in its definition, we need to change it in all functions that reference that variable. Because of this high degree of coupling, we avoid global variables altogether. Their presence complicates maintainability: if 1000s of functions reference the variable, changing its name proves to be a nightmare.

Function Scope

A variable that we **declare within a function header** has *function scope*. We call such variables *function parameters*. The scope of the parameter extends from the function header to the closing brace of the function body. We say that the parameter goes out of scope at this closing brace.

The compiler allocates memory for a function parameter when the function is called and releases that memory when the function return control to its caller. C compilers initialize the values of the function parameters to the values of the arguments in the function call.

Local Scope

A variable that we **declare within a function body** has *local scope*. We call such variables *local variables*. The scope of the variable extends from its definition to the end of the code block within which we declared the variable. We say that the variable goes out of scope at the closing brace of its code block.

The compiler allocates memory for a local variable where it is defined (or first used) and releases that memory when the variable goes out of scope. C compilers do not initialize the values of local variables.

Block Scope

A variable that we **declare within a code block** has *block scope*. The scope of the variable extends from its definition to the end of the code block within which we declared the variable. We say that the variable goes out of scope at the closing brace of its code block.

Overlapping Scope (Optional)

A variable occupies its own memory location from its declaration to the end of its parent code block. We refer to this period as the variable's *lifetime*. A variable is visible throughout its lifetime as long as it is not hidden by another variable of the same name.

Consider the following program. The variable `x` within the code block hides the local parameter `x` until the end of the iteration.

```c
/* Avoid Variables of the Same Name
 * lifetime.c
 */

#include <stdio.h>

void foo(int x)
{
    int i = 4;

    do {
        int x = i;
        printf("%d ", x);
        i--;
    } while(i > 0);

    printf("%d ", x);
}

int main(void)
{
    foo(6);
    return 0;
}
```

The program above will produce the following output:

```
4 3 2 1 6
```

We say that the `x` declared within the code block ***shadows*** the parameter `x` declared in the function header.

> Using the same name for two distinct variables with overlapping lifetimes only introduces confusion. Although compilers accept such code, it is poor style and best avoided altogether.

Passing Arrays

The elements of an array occupy contiguous locations in memory. Because of this accessing any element from within a function only requires the address of the start of the array and the element index.

The name of an array without the brackets holds the address of the start of the array.

Array Arguments

To grant a function access to an array, we pass the array's address as an argument in the function call. The call takes the form

```
function_identifier(array_identifier, ... )
```

By passing the address, we avoid copying the entire array. The decision to pass arrays in this way was made when the C language was designed.

For example:

```
// Passing an Array to a Function
// passArray.c

#include <stdio.h>
#define NGRADES 8

// definition of display() ...

int main(void)
{
    int grade[] = {10,9,10,8,7,9,8,10};

    display(grade, NGRADES);

    return 0;
}
```

Parameters

A function header that receives an array's address takes the form:

```
type function_identifier(type array_identifier[], ... )
```

or

```
type function_identifier(type *array_identifier, ... )
```

The empty brackets following `identifier` in the first alternative tell the compiler that the parameter holds the address of a one-dimensional array. For example:

```
// array using []
void display(int g[], int n)
{
```

```
    for(i = 0; i < n; i++)
    {
        printf("%d ", g[i]);
    }
}
```

OR:

```
// array using * (pointer)
void display(int *g, int n)
{
    for(i = 0; i < n; i++)
    {
        printf("%d ", g[i]);
    }
}
```

Because we have passed the address of the array and not a copy of all of its elements, any change to an element within the function will change the array element in the caller.

Barring Changes

To prevent a function from changing any element of an array identified by a function parameter, we qualify the parameter as `const`. The function header takes the form:

```
type function_identifier(const type array_identifier[], ... )
```

OR:

```
type function_identifier(const type *array_identifier, ... )
```

For example:

```
// array using []
void display(const int g[], int n)
{
    for(i = 0; i < n; i++)
    {
        printf("%d ", g[i]);
    }
}
```

OR:

```
// array using * (pointer)
void display(const int *g, int n)
{
    for(i = 0; i < n; i++)
    {
        printf("%d ", g[i]);
    }
}
```

> Any attempt to modify the value of an element of `g` will generate a compiler error. Without the `const` keyword, we could reset the value of the first element to 10 by adding a statement like `g[0] = 10;`.

Passing Structures

We can pass an object of structure type to a function in either of two ways:

- pass by value
- pass by address
- Pass By Value

Consider the following program. Note that the `Student` structure includes a member that identifies the number of grades filled. We pass `harry` as a single argument to `display()` and access its member within the function:

```
// Passing a structure to a function
// pass_by_value.c

#include <stdio.h>

struct Student
{
    int no;
    int no_grades_filled;
    float grade[4];
};

void display(const struct Student s);  // pass by value

int main(void)
{
    struct Student harry = {975, 3,
         {75.6f, 82.3f, 68.9f, 0.0f}};

    display(harry);
}

void display(const struct Student st)
{
    int i;
    printf("Grades for %d\n", st.no);

    for (i = 0; i < st.no_grades_filled; i++)
    {
        printf("%.1f\n", st.grade[i]);
    }
}
```

The above program produces the following output:

```
Grades for 975
75.6
82.3
68.9
```

The declaration of `Student` precedes the prototype for `display()`. The compiler needs this declaration to interpret the parameter type in the prototype.

The C compiler passes objects of structure type by value. It copies the value of the argument in the function call into the parameter, as its initial value. Any change within the function affects only the copy and not the original value.

In the following example, the data stored in `harry` does not change after the function `set()` returns control to `main()`:

```
// Pass by Value
// pass_by_value.c
```

```c
#include <stdio.h>

struct Student
{
    int no;
    int no_grades_filled;
    float grade[4];
};

void set(struct Student st);
void display(const struct Student st);

int main(void)
{
    struct Student harry = { 975, 2, {50.0f, 50.0f}};

    set(harry);
    display(harry);
}

void set(struct Student st)
{
    struct Student harry = {306, 2, {78.9, 91.6}};

    st = harry;
}

void display(const struct Student st)
{
    int i;
    printf("Grades for %d\n", st.no);

    for (i = 0; i < st.no_grades_filled; i++)
    {
        printf("%.1f\n", st.grade[i]);
    }
}
```

The above program produces the following output:

```
Grades for 975
50.0
50.0
```

The values in the original object, its copy and the local object are shown in the table below:

`int main()`	`void set()`	
`struct Student harry`	`struct Student st`	`struct Student harry`
975 2 50.0f 50.0f	975 2 50.0f 50.0f	
975 2 50.0f 50.0f	975 2 50.0f 50.0f	306 2 78.9f 91.6f
975 2 50.0f 50.0f	306 2 78.9f 91.6f	306 2 78.9f 91.6f

Copying

The C compiler performs member-by-member copying automatically whenever we:

- pass an object by value

- assign an object to an existing object
- initialize a new object using an existing object
- return an object by value
- pass by address

To change the data within an original object passed to the `set()` function, we require the address of the original object. In the call to `set()` we pass its address. `set()` receives this address in its pointer parameter.

In the following program, we pass the address of harry to `set()`:

```
// Pass by Address
// pass_by_address.c

#include <stdio.h>

struct Student
{
        int no;
        int no_grades_filled;
        float grade[4];
};

void set(struct Student* st);
void display(const struct Student st);

int main(void)
{
    struct Student harry = { 975, 2, {50.0f, 50.0f}};

    set(&harry);
    display(harry);
}

void set(struct Student* st)
{
    struct Student harry = {306, 2, {78.9, 91.6}};

    *st = harry;
}

void display(const struct Student st)
{
    int i;
    printf("Grades for %d\n", st.no);

    for (i = 0; i < st.no_grades_filled; i++)
    {
        printf("%.1f\n", st.grade[i]);
    }
}
```

The above program produces the following output:

```
Grades for 306
78.9
91.6
```

The values in the original object and the local object are shown in the table below:

`int main()`	`void set()`	`void display()`

`struc`	`int main()`	`struct Student *`	`void set()`	`s`	`void display()`
`harry`		`t Student harry`		`t`	

`struct Student harry`	`struct Student *st`	`struct Student harry`	`struct Student st`
Address: 22ff2b8d4	22ff2b8ec	22ff2b8f0	22ff2b908
975 2 50.0f 50.0f			
306 2 78.9f 91.6f	2ff2b8d4	306 2 78.9f 91.6f	
306 2 78.9f 91.6f			306 2 78.9f 91.6f

Efficiency

Passing an object ***by address*** is efficient. It avoids copying all member values, saving time and space especially in cases where a member is an array with a large number of elements. Passing an object by address only copies the address, which typically occupies 4 bytes.

Consider passing `harry` by address to function `display()` as well:

```c
// Pass by Address 1
// pass_by_address_1.c

#include <stdio.h>

struct Student
{
    int no;
    int no_grades_filled;
    float grade[4];
};

void set(struct Student* st);
void display(const struct Student* st);

int main(void)
{
    struct Student harry = { 975, 2, {50.0f, 50.0f}};

    set(&harry);
    display(&harry);
}

void set(struct Student* st)
{
    struct Student harry = {306, 2, {78.9, 91.6}};

    *st = harry;
}

void display(const struct Student* st)
{
    int i;
    printf("Grades for %d\n", (*st).no);

    for (i = 0; i < (*st).no_grades_filled; i++)
    {
        printf("%.1f\n", (*st).grade[i]);
    }
}
```

The above program produces the following output:

```
Grades for 306
78.9
91.6
```

`display()` dereferences the address before selecting the members. Since the dot operator binds tighter than the dereferencing operator, the parentheses are necessary. Omitting them would generate a compiler error (the data member after the dot operator is not of type `Student*`).

If we **pass by address** with no intention of changing that object within the function, we add the `const` qualifier to safeguard against accidental modifications.

The values in the original object and the local object are shown in the table below:

`int main()`	`void set()`		`void display()`
`struct Student harry`	`struct Student *st`	`struct Student harry`	`const struct Student *st`
Address: 22ff2b8d4	22ff2b8ec	22ff2b8f0	22ff2b908
975 2 50.0f 50.0f			
306 2 78.9f 91.6f	2ff2b8d4	306 2 78.9f 91.6f	
306 2 78.9f 91.6f			2ff2b8d4

Arrow Notation

The syntax `(*s).no` is awkward to read. ***Arrow notation*** provides cleaner alternative. It takes the form:

```
address->member
```

The arrow operator takes a pointer to an object on its left and a member identifier on its right.

For example:

```c
// Pass by Address 3
// arrow_notation.c

#include <stdio.h>

struct Student
{
        int no;
        int no_grades_filled;
        float grade[4];
};

void set(struct Student* st);
void display(const struct Student* st);

int main(void)
{
    struct Student harry = { 975, 2, {50.0f, 50.0f}};

    set(&harry);
    display(&harry);
}
```

```
void set(struct Student* st)
{
    struct Student harry = {306, 2, {78.9, 91.6}};

    *st = harry;
}

void display(const struct Student* st)
{
    int i;
    printf("Grades for %d\n", st->no);

    for (i = 0; i < st->no_grades_filled; i++)
    {
        printf("%.1f\n", st->grade[i]);
    }
}
```

The above program produces the following output:

```
Grades for 306
78.9
91.6
```

Style

The rules of structured programming extend directly to functions. A structured program does not have multiple returns. We replace multiple return statements with flags logic and a single return statement.

It is good programming style to:

- place the opening brace on a separate line
- declare a prototype for each function definition
- include parameter identifiers in the prototype declaration as documentation
- use generic comments and variables names to enable future use in different applications without having to modify any of the function code
- avoid calling the `main()` function recursively
- limit the number of local variables to below 10, if possible

Documentation

We document each function at one level of abstraction above the caller. We precede the function header in the function definition with comments that describe:

- what the function does (not how)
- what the function needs (in terms of values for its parameters)
- what the function returns (if anything)

We state any assumption or constraint that applies to using the function.

For example:

```
// power returns the value of base raised to
//  the power of exponent (base^exponent)
//
//  power assumes that base and exponent are
//  integer values and exponent is non-negative
//
int power(int base, int exponent)
```

```
{
    int result, i;

    result = 1;
    for (i = 0; i < exponent; i++)
    {
        result = result * base;
    }

    return result;
}
```

Structure Walkthrough

The following program contains several objects of type A. The walkthrough table is shown below.

```
// Structure Types - Walkthrough
// struct_walk.c

#include <stdio.h>

struct A
{
    int x;
    double r;
};

void foo(struct A* c);
struct A goo(struct A d);

int main(void)
{
    struct A a = {4, 6.67}, b;

    foo(&a);

    printf("00%d.%.3lf.111\n", a.x, a.r);

    b = goo(a);

    printf("00%d.%.3lf.112\n", a.x, a.r);
    printf("%d.%.3lf.113\n", b.x, b.r);
}

void foo(struct A* c)
{
    int i;

    i = c->x;
    c->x = c->r;
    c->r = c->x % i + 202.134;
}

struct A goo(struct A d)
{
    struct A e;

    d.x = d.r - 62;
    e = d;
    return e;
}
```

The table includes:

- the return type for each function
- the name of each function
- the structure type of each object
- the name of each object
- the type of each member
- the name of each member

> **NOTE**
>
> The breakdown of each object into its members in the head of the table. We reserve a separate line for the addresses that are pointed to:

`int`				`void`		`struct A`			
`main()`				`foo()`		`goo()`			
`struct A`		`struct A`		`struct A*`		`struct A`		`struct A`	
a		b		c		d		e	
Address: 1000		100C		1018	101C	1020		102C	
int	double	int	double		int	int	double	int	double
x	r	x	r		i	x	r	x	r
				1000					
				1000					
				1000					
				1000					
				1000					

On this page

Input Functions

Learning Outcomes

After reading this section, you will be able to:

- Invoke standard library procedures to stream data from users

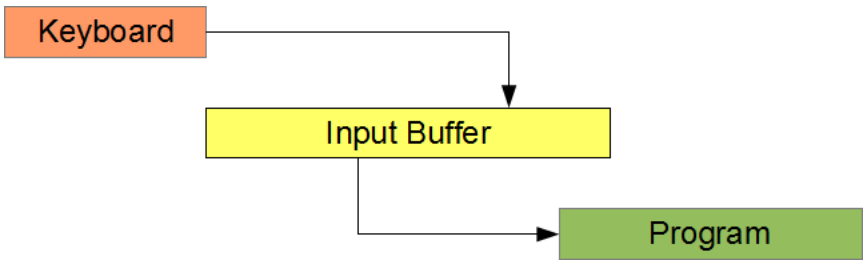
Introduction

Some programming languages leave input and output support to the libraries developed for the languages. For instance, the core C language does not include input and output specifications. These facilities are available in a set of functions, which are defined in the `stdio` module. This module ships with the C compiler. Its name stands for standard input and output. Typically, standard input refers to the system keyboard and standard output refers to the system display. The system header file that contains the prototypes for the functions in this module is `<stdio.h>`.

This chapter describes some of the input facilities supported by the `stdio` module, introduces buffered input, describes two library functions that accept formatted and unformatted buffered input and demonstrates how to validate user input.

Buffered Input

A buffer is a small region of memory that holds data temporarily and provides intermediate storage between a device and a program. The system stores each keystroke in the input buffer, without passing it to the program. The user can edit their data before submitting it to the program. only by pressing the \n key, the user signals the program to start extracting data from the buffer. The program then only retrieves the data that it needs and leaves the rest in the buffer for future retrievals. The figure below illustrates the buffered input process.



Two functions accept buffered input from the keyboard (the standard input device):

- getchar() - unformatted input
- scanf() - formatted input

Unformatted Input

The function `getchar()` retrieves the next unread character from the input buffer.

The prototype for `getchar()` is

```
int getchar(void);
```

`getchar()` returns either:

- the character code for the retrieved character
- EOF

The character code is the code from the collating sequence of the host computer. If the next character in the buffer waiting to be read is `'j'` and the collating sequence is ASCII, then the value returned by `getchar()` is 106.

`EOF` is the symbolic name for end of data. It is assigned the value -1 in the `<stdio.h>` system header file. On Windows systems, the user enters the end of data character by pressing `Ctrl-Z`; on UNIX systems, by pressing `Ctrl-D`.

Clearing the buffer

To synchronize user input with program execution the buffer should be empty. The following function clears the input buffer of all unread characters:

```
// clear empties the input buffer
//
void clear(void)
{
    while (getchar() != '\n')
    {
        ;  // empty statement intentional
    }
}
```

The iteration continues until `getchar()` returns the newline (`'\n'`) character, at which point the buffer is empty and `clear()` returns control to its caller.

Pausing Execution

To pause execution at a selected point in a program, consider the following function:

```
// pause execution
//
void pause_(void)
{
    printf("Press enter to continue ...");
    while (getchar() != '\n')
    {
        ;  // empty statement intentional
    }
}
```

This function will not return control to the caller until the user has pressed `'\n'`.

Formatted Input

The `scanf()` function retrieves the next set of unread characters from the input buffer and translates them according to the conversion(s) specified in the format string. `scanf()` extracts only as many characters as required to satisfy the specified conversion(s).

The prototype for scanf() is:

```
int scanf(format, ... );
```

format consists of one or more conversion specifiers enclosed in a pair of double quotes. The ellipsis (**...**) refers to one or more addresses.

`scanf()` extracts characters from the input buffer until `scanf()` has either:

- interpreted and processed data to match all conversion specifiers in the format string
- found a character that fails to match the next conversion specified in the format string
- emptied the buffer completely

In a mismatch between the conversion specifier and the next character in the buffer, `scanf()` **leaves the offending character in the buffer** and returns to the caller. In the case of an emptied buffer, `scanf()` waits until the user adds more data to the buffer.

Each conversion specifier describes how `scanf()` is to interpret the next set of characters in the buffer. Once `scanf()` has completed a conversion, it stores the result in the address passed to the corresponding parameter.

We provide as many conversion specifiers in the format string as there are address arguments in the call to `scanf()`.

Conversion Specifiers

Each conversion specifier begins with a `%` symbol and ends with a conversion character. The conversion character describes the type to which `scanf()` is to convert the next set of text characters.

Specifier	Input Text	Convert to Type...	Most Common (*)
%c	character	char	*
%d	decimal	char, int, short, long, long long	*
%o	octal	int, char, short, long, long long	
%x	hexadecimal	int, char, short, long, long long	
%f	floating-point	float, double, long double	*

The following program converts two input fields into data values of int type and float type respectively:

```c
/* scanf conversion specifiers
* scanf.c
*/

#include <stdio.h>

int main(void)
{
    int items;
    float price;

    printf("Enter items, price : ");
    scanf("%d%f", &items, &price);

    return 0;
}
```

The above program produces the following output:

```
Enter items, price : 4 39.99
```

Whitespace

`scanf()` treats the whitespace between text characters of the user's input as a separator between input values. There is no need to place a blank character between the conversion specifiers.

Conversion Control

We may insert control characters between the `%` and the `conversion character`. The general form of a conversion specification is

```
% * width size conversion_character
```

The three control characters are:

- ***** - suppresses storage of the converted data (discards it without storing it)
- ***width*** - specifies the maximum number of characters to be interpreted
- ***size*** - specifies the size of the storage type

For integer values:

Size Specifier	Convert to Type
-none-	int
hh	char
h	short
l	long
ll	long long

For floating-point values:

Size Specifier	Convert to Type
-none-	float
l	double
L	long double

A conversion specifier that includes an ***** does not have a corresponding address in the argument list. This is an exception to the matching conversion-specifier/argument rule.

Problems with %c (Optional)

Because `scanf()` only extracts the characters that it needs from the input buffer, problems arise with `%c` conversions. If you encounter such difficulty, see the section with this title in the chapter entitled More Input and Output.

Plain Characters (Optional)

Plain characters in the format string - those not preceded by the conversion symbol - serve a special purpose. Each such character requires exact duplication on input. If the user enters a plain character other than that specified in the format string, scanf() abandons further interpretation.

To input `%` as a plain character (and distinguish it from the symbol identifying a conversion specifier), we insert `%%` into the format string.

Return Value

`scanf()` returns either the number of addresses successfully filled or `EOF`. A return value of:

- 0 indicates that `scanf()` did not fill any address

- 1 indicates that `scanf()` filled the first address successfully
- 2 indicates that `scanf()` filled the first and second addresses successfully
- `EOF` indicates that `scanf()` did not fill any address **AND** encountered an end of data character

The return code from `scanf()` does not reflect success of **%*** conversions or any successful reading of plain characters in the format string.

Validation (Optional)

Since we can never predict that all users will never make mistakes in inputting data to our programs, input validation is an important part of our programming tasks.

To validate the input data that a program receives, we can perform many checks. We localize our validation in special functions that trap erroneous input and request corrections to that input. Erroneous input may include:

- invalid characters
- trailing characters
- out-of-range input
- incorrect number of input fields

The following program includes a special function (`getInt()`), which provides robust validation for integer input. This function tests for no input, trailing characters and out-of-range input.

```c
/* Robust Input Validation
*  getInt.c
*/

#include <stdio.h>
#define MIN 3
#define MAX 15

int getInt(int min, int max);
void clear(void);

int main(void)
{
    int input;

    input = getInt(MIN, MAX);
    printf("\nProgram accepted %d\n", input);

    return 0;
}

// getInt accepts an int between min and max
// inclusive, returns the value of the int accepted
//
int getInt(int min, int max)
{
    int value, keeptrying = 1, rc;
    char after;

    do {
        printf("Enter an integer in range [%d,%d] : ", min, max);
        rc = scanf("%d%c", &value, &after);

        if (rc == 0)
        {
            printf("**Bad char(s)!**\n");
            clear();
        }
        else if (after != '\n')
        {
```

```
            printf("**Trail char(s)!**\n");
            clear();
        }
        else if (value < min || value > max)
        {
            printf("**Out of range!**\n");
        }
        else
        {
            keeptrying = 0;
        }
    } while (keeptrying == 1);

    return value;
}

// clear empties the input buffer
//
void clear(void)
{
    while (getchar() != '\n')
    {
        ;  // empty statement intentional
    }
}
```

The above program produces the following output:

```
Enter an integer in range [3,15] : we34
**Bad char(s)!**
Enter an integer in range [3,15] : 34.4
**Trail char(s)!**
Enter an integer in range [3,15] : 345
**Out of range!**
Enter an integer in range [3,15] : 14

Program accepted 14
```

On this page

Output Functions

Learning Outcomes

After reading this section, you will be able to:

- Invoke standard library procedures to stream data to users

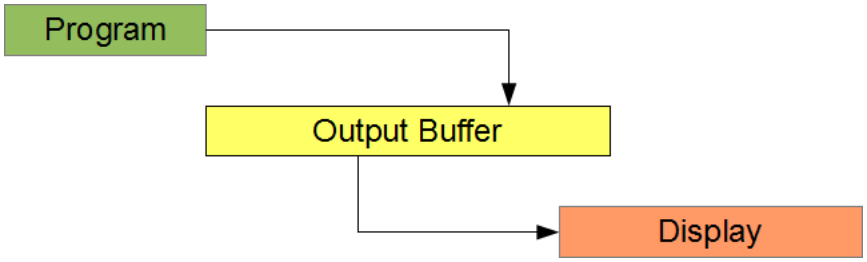
Introduction

The adequate provision of a user interface is an important aspect of software development: an interface that consists of user-friendly input and user-friendly output. The output facilities of a programming language convert the data in memory into a stream of characters that is read by the user. The `stdio` module of the C language provides such facilities.

This chapter describes two functions in the `stdio` module that provide formatted and unformatted buffered support for streaming output data to the user and demonstrates in detail how to format output for a user-friendly interface.

Buffering

Standard output is line buffered. A program outputs its data to a buffer. That buffer empties to the standard output device separately. When it empties, we say that the buffer flushes.



Output buffering lets a program continue executing without having to wait for the output device to finish displaying the characters it has received.

The output buffer flushes if:

- it is full
- it receives a newline (`\n`) character
- the program terminates

Two functions in the `stdio` module that send characters to the output buffer are

- `putchar()` - unformatted
- `printf()` - formatted

Unformatted Output

The `putchar()` function sends a single character to the output buffer. We pass the character as an argument to this function. The function returns the character sent or `EOF` if an error occurs.

The prototype for `putchar()` is:

```
int putchar (int);
```

To send the character 'a' to the display device, we write:

```
// Single character output
// putchar.c

#include <stdio.h>

int main(void)
{
    putchar('a');
    return 0;
}
```

The above program produces the following output:

```
a
```

Formatted Output

The `printf()` function sends data to the output buffer under format control and returns the number of characters sent.

The prototype for the `printf()` function is:

```
int printf(format, argument, ... );
```

format is a set of characters enclosed in double-quotes that may consist of any combination of plain characters and conversion specifiers. The function sends the plain characters as is to the buffer and uses the conversion specifiers to translate each value passed as an argument in the function call. The ellipsis indicates that the number of arguments can vary. Each conversion specifier corresponds to one argument.

Conversion Specifiers

A conversion specifier begins with a `%` symbol and ends with a ***conversion character***. The conversion character defines the formatting as listed in the table below:

Specifier	Format As	User With Type ...	Common(*)
%c	character	char	*
%d	decimal	char, int, short, long, long long	*
%o	octal	char, int, short, long, long long	
%x	hexadecimal	char, int, short, long, long long	
%f	floating-point	float, double, long double	*
%g	general	float, double, long double	

Specifier	Format As	User With Type ...	Common(*)
%e	exponential	float, double, long double	

For example:

```
int i = 15;
float x = 3.141593f;
printf("i is %d; x is %f\n", i, x);
```

The above code snippet produces the following output:

```
i is 15; x is 3.141593
```

Conversion Controls

We refine the output by inserting control characters between the `%` symbol and the conversion character. The general form of a conversion specification is:

```
% flags width . precision size conversion_character
```

The five control characters are:

1. ***flags***
   - Prescribes left justification of the converted value in its field
   - 0 pads the field width with leading zeros
2. ***width*** sets the minimum field width within which to format the value (overriding with a wider field only if necessary). Pads the converted value on the left (or right, for left alignment). The padding character is space or 0 if the padding flag is on
3. `.` separates the field's width from the field's precision
4. ***precision*** sets the number of digits to be printed after the decimal point for `f` conversions and the minimum number of digits to be printed for an integer (adding leading zeros if necessary). A value of `0` suppresses the printing of the decimal point in an `f` conversion
5. ***size*** identifies the size of the type being output

Integral values

Size Specifier	User with Type
none	int
hh	char
h	short
l	long
ll	long long

Floating-point values

Size Specifier	User with Type
none	float
l	double

Size Specifier	User with Type
L	long double

Special Characters

To insert the special characters `\`, `'`, and `"`, we use their escape sequences. To insert the special character `%` into the format, we use the `%` symbol:

```
// Outputting special characters
// special.c

int main(void)
{
    printf("\\ \' \" %%\n");
    return 0;
}
```

The above program produces the following output:

```
 \ ' " %
```

Reference Example

The following program produces the output listed on the right for the ASCII collating sequence:

```
// Playing with output formatting
// printf.c
#include <stdio.h>

int main(void)
{
    /* integers */
    printf("\n* ints *\n");
    printf("00000000011\n");
    printf("12345678901\n");
    printf("------------------------\n");
    printf("%d|<--        %%d\n",4321);
    printf("%10d|<--  %%10d\n",4321);
    printf("%010d|<--  %%010d\n",4321);
    printf("%-10d|<--  %%-10d\n",4321);
    /* floats */
    printf("\n* floats *\n");
    printf("00000000011\n");
    printf("12345678901\n");
    printf("------------------------\n");
    printf("%f|<-- %%f\n",4321.9876546);
    /* doubles */
    printf("\n* doubles *\n");
    printf("00000000011\n");
    printf("12345678901\n");
    printf("------------------------\n");
    printf("%lf|<-- %%lf\n",4321.9876546);
    printf("%10.3lf|<--  %%10.3lf\n",4321.9876);
    printf("%010.3lf|<--  %%010.3lf\n",4321.9876);
    printf("%-10.3lf|<--  %%-10.3lf\n",4321.9876);
    /* characters */
    printf("\n* chars *\n");
    printf("00000000011\n");
    printf("12345678901\n");
    printf("------------------------\n");
```

```
    printf("%c|<--           %%c\n",'d');
    printf("%d|<--         %%d\n",'d');
    printf("%x|<--          %%x\n",'d');
    return 0;
}
```

The above program produces the following output:

```
* ints *
00000000011
12345678901
------------------------
4321|<--        %d
    4321|<--  %10d
0000004321|<--  %010d
4321      |<--  %-10d

* floats *
00000000011
12345678901
------------------------
4321.987655|<-- %f

* doubles *
00000000011
12345678901
------------------------
4321.987655|<-- %lf
4321.988|<--  %10.3lf
004321.988|<--  %010.3lf
4321.988  |<--  %-10.3lf

* chars *
00000000011
12345678901
------------------------
d|<--           %c
100|<--         %d
64|<--          %x
```

> **NOTE**
>
> - `doubles` and `floats` **round** to the requested precision before being displayed
> - `double` data may be displayed using `%f` (`printf()` converts float values to doubles for compatibility with legacy programs)
> - `char`acter data can be displayed in various formats including:
>   - character
>   - decimal
>   - hexadecimal

Portability Note (Optional)

Character data is encoded on many computers using the ASCII standard, but not all computers use this sequence. A program is portable across sequences if it refers to character data in its **symbolic** form (`'A'`) and to special characters - such as newline, tab, and formfeed - by their escape sequences (`'\n'`, `'\t'`, `'\f'`, etc.) rather than by their decimal or hexadecimal values.

On this page

Library Functions

Learning Outcomes

After reading this section, you will be able to:

- Implement algorithms using standard library procedures to incorporate existing technology

Introduction

The standard libraries that support programming languages perform many common tasks. C compilers ship with the libraries that include functions for mathematical calculations, generation of random events and manipulation and analysis of character data. To access any function within any library we simply include the appropriate header file for that library and call the function in our source code.

This chapter introduces some of the more common functions in the libraries that ship with C compilers. The GNU Documentation includes a comprehensive description of each function in each library.

Mathematical Functions

The mathematics related libraries that contain the more common mathematical functions are:

- `stdlib` - the standard library
- `math` - the math library

Standard Library

The header file `<stdlib.h>` contains prototypes for the functions that perform the more general mathematical calculations. These calculations include **absolute** values of **integers** and **random** number generation.

Integer Absolute Value

`abs()`, `labs()`, `llabs()` return the absolute value of the argument. Their prototypes are:

```
int abs(int);
long labs(long);
long long llabs(long long);
```

Example:

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

int main(void)
{
    int x = -12;
    long y = -24L;

    printf("|%d| is %d\n", x, abs(x));
    printf("|%ld| is %ld\n", y, labs(y));

    return 0;
}
```

The above program produces the following output:

```
|-12| is 12
|-24| is 24
```

Random Numbers

`rand()` returns a pseudo-random integer in the range `0` to `RAND_MAX`. `RAND_MAX` is implementation-dependent but **no less** than `32767`. The prototype for `rand()` is:

```
int rand(void);
```

The following program outputs the same set of 10 pseudo-random integers for each successive run:

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

int main(void)
{
    int i;

    for (i = 0; i < 10 ; i++)
    {
        printf("Random number %d is %d\n", i+1, rand());
    }

    return 0;
}
```

The following program outputs the same set of 10 pseudo-random integers between `6` and `100` inclusive:

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

int main(void)
{
    int i, n, a = 6, b = 100;

    for (i = 0; i < 10 ; i++)
    {
        n = a + rand() % (b + 1 - a);
        printf("Random number %d is %d\n", i+1, n);
    }

    return 0;
}
```

The following program outputs the same set of 10 pseudo-random floating-point numbers between `3.0` and `100.0` inclusive:

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

int main(void)
{
    int i;
    double x, a = 3.0, b = 100.0;

    for (i = 0; i < 10 ; i++)
```

```
    {
        x = a + ((double) rand() / RAND_MAX * (b - a));
        printf("Random number %d is %.2lf\n", i+1, x);
    }

    return 0;
}
```

`rand()` generates the **same set of random numbers** for every run of its host application. This is very useful during the debugging stage. To generate a different set of random numbers for every run we add a call to the function `srand()`. `srand()` sets the **seed** for the random number generator. The prototype for `srand()` is:

```
int srand(unsigned seed);
```

`unsigned` is a type that only holds **non-negative** integer values. We call `srand()` once with `time(NULL)` as the argument (see below) before the first call to `rand()`, typically at the start of our program. This provides a **unique seed for each run** and hence different pseudo-random numbers.

The following program outputs a different set of 10 pseudo-random numbers with every run:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h> // prototype for time(NULL)

int main(void)
{
    int i;

    srand(time(NULL)); // will set a unique seed for each run of the program
    for (i = 0; i < 10 ; i++) // iterate with “i” as index
    {
        printf("Random number %d is %d\n", i+1, rand());
    }

    return 0;
}
```

math (Optional)

The math library contains many functions that perform mathematical calculations. Their prototypes are listed in `<math.h>`. To compile a program that uses one of these functions with the gcc compiler, we add the -lm option to the command line:

```
gcc myProgram.c -lm
```

Floating-Point Absolute Value

`fabs()`, `fabsf()`, `fabsl()` return the absolute value of the argument. Their prototypes are:

```
double fabs(double);
float fabsf(float);
long double fabsl(long double);
```

Example

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

int main(void)
```

```
{
    float w = -12.5;
    double x = -12.5;

    printf("|%f| is %f\n", w, fabsf(w));
    printf("|%lf| is %lf\n", x, fabs(x));

    return 0;
}
```

The above program produces the following output:

```
|-12.500000| is 12.500000
|-12.500000| is 12.500000
```

Floor

`floor()`, `floorf()`, `floorl()` return the largest integer value not greater than the argument. Their prototypes are:

```
double floor(double);
float floorf(float);
long double floorl(long double);
```

For example, `floor(16.3)` returns a value of `16.0`.

Ceiling

`ceil()`, `ceilf()`, `ceill()` return the smallest integer value not less than the argument. Their prototypes are:

```
double ceil(double);
float ceilf(float);
long double ceill(long double);
```

For example, `ceil(16.3)` has a value of `17.0`.

Rounding

`round()`, `roundf()`, `roundl()` return the integer value closest to the argument. Their prototypes are:

```
double round(double);
float roundf(float);
long double roundl(long double);
```

For example, `round(16.3)` has a value of `16.0`, while `round(-16.3)` returns a value of `-16.0`.

Truncating

`trunc()`, `truncf()`, `truncl()` return the integer part of the argument. Their prototypes are:

```
double trunc(double);
float truncf(float);
long double truncl(long double);
```

For example, `trunc(16.7)` has a value of `16.0`, while `trunc(-16.7)` returns a value of `-16.0`.

Square Root

`sqrt()`, `sqrtf()`, `sqrtl()` return the square root of the argument. Their prototypes are:

```
double sqrt(double);
float sqrtf(float);
long double sqrtl(long double);
```

For example, `sqrt(16.0)` returns a value of `4.0`.

Powers

`pow()`, `powf()`, `powl()` return the result of the first argument raised to the power of the second argument. Their prototypes are:

```
double pow(double base, double exponent);
float powf(float base, float exponent);
long double powl(long double base, long double exponent);
```

Example:

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

int main(void)
{
    double base = 12.5;

    printf("%lf^3 is %lf\n", base,
    pow(base,3));

    return 0;
}
```

The above program produces the following output:

```
12.500000^3 is 1953.125000
```

This set of functions was designed for floating-point arguments and each one is computationally intensive. Avoid using them for simpler integer arguments.

Logarithms

`log()`, `logf()`, `logl()` return the natural logarithm of the argument. Their prototypes are:

```
double log(double);
float logf(float);
long double logl(long double);
```

For example, `log(2.718281828459045)` returns a value of `1.0`.

Powers of e

`exp()`, `expf()`, `expl()` return the natural anti-logarithm of the argument. Their prototypes are:

```
double exp(double);
float expf(float);
long double expl(long double);
```

For example, `exp(1.0)` returns a value of `2.718281828459045`.

Time Functions (Optional)

The time library contains functions that return timing data. Their prototypes are listed in `<time.h>`.

Calendar Time

time

`time(NULL)` returns the current calendar time. The prototype is:

```
time_t time(time_t *);
```

`time_t` is a type that it is sufficiently large to hold time values - for example, `unsigned long`.

difftime

`difftime()` returns the difference in ***seconds*** between two calendar time arguments. The prototype is:

```
double difftime(time_t, time_t);
```

`time_t` is a type that it is sufficiently large to hold time values - for example, `unsigned long`.

The following program returns the time in seconds taken to execute the central iteration:

```
#include <time.h>
#include <stdio.h>
#define NITER 1000000000

int main(void)
{
    double x;
    int i, j, k;
    time_t t0, t1;

    x = 1;
    t0 = time(NULL);
    for (i = 0; i < NITER; i++)
    {
        x = x * 1.0000000001;
    }

    t1 = time(NULL);
    printf("Elapsed time is %.1lf secs\n", difftime(t1, t0));
    printf("Value of x is %.10lf\n", x);

    return 0;
}
```

The above program produces the following output:

```
Elapsed time is 40.0 secs
Value of x is 1.1051709272
```

Process Time

clock

`clock()` returns the approximate process time. The prototype is:

```
clock_t clock(void);
```

`clock_t` is a type that holds time values - for example, `unsigned long`. The value is in units of `CLOCKS_PER_SEC`. We divide by these units to obtain the process time in seconds.

The following program returns the process time in seconds taken to execute the central iteration:

```
#include <time.h>
#include <stdio.h>
#define NITER 400

int main(void)
{
    double x;
    int i, j, k;
    time_t t0, t1;
    clock_t c0, c1;

    x = 1;
    t0 = time(NULL);
    c0 = clock();

    for (i = 0; i < NITER; i++)
    {
        for (j = 0; j < NITER; j++)
        {
            for (k = 0; k < NITER; k++)
            {
                x = x * 1.0000000001;
            }
        }
    }

    t1 = time(NULL);
    c1 = clock();

    printf("Elapsed time is %.1lf secs\n", difftime(t1, t0));
    printf("Process time is %.3lf secs\n", (double)(c1-c0)/CLOCKS_PER_SEC);
    printf("Value of x is %.10lf\n", x);

    return 0;
}
```

The above program produces the following output:

```
Elapsed time is 1.0 secs
Process time is 0.040 secs
Value of x is 1.0001000050
```

> **NOTE**
>
> The cast to a `double` forces floating-point division.

Character

The `ctype` library contains functions for character manipulation and analysis. Their prototypes are listed in `<ctype.h>`.

Manipulation

tolower

`tolower()` returns the lower case of the character received, if possible; otherwise, the character received unchanged. The prototype is:

```
int tolower(int);
```

For example, `tolower('D')` returns `'d'`, while `tolower(';')` returns `';'`.

toupper

`toupper()` returns the upper case of the character received, if possible; otherwise, the character received unchanged. The prototype is:

```
int toupper(int);
```

For example, `toupper('d')` returns `'D'`, while `toupper(';')` returns `';'`.

Analysis

The character analysis functions return 'true' or 'false'. **C represents 'false' by the value 0** and 'true' by **any other value**.

islower

`islower()` returns a true value if the character received is lower case, a false value otherwise. The prototype is:

```
int islower(int);
```

For example, `islower('d')` returns a true value, while `islower('E')` returns a false value.

isupper

`isupper()` returns a true value if the character received is upper case, a false value otherwise. The prototype is:

```
int isupper(int);
```

For example, `isupper('d')` returns a false value, while `isupper('E')` returns a true value.

isalpha

`isalpha()` returns a true value if the character received is alphabetic, a false value otherwise. The prototype is:

```
int isalpha(int);
```

For example, `isalpha('d')` returns true, while `isalpha('6')` returns false.

isdigit

`isdigit()` returns a true value if the character received is a digit, a false value otherwise. The prototype is:

```
int isdigit(int);
```

For example, `isdigit('d')` returns false, while `isdigit('6')` returns true.

isspace

`isspace()` returns a true value if the character received is a whitespace character, a false value otherwise. The prototype is:

```
int isspace(int);
```

For example, `isspace('d')` returns false, while `isspace(' ')` returns true. Whitespace characters are:

- `' '`
- `'\t'`
- `'\n'`
- `'\v'`
- `'\f'`

isblank

`isblank()` returns a true value if the character received is a space or tab, a false value otherwise. The prototype is:

```
int isblank(int);
```

For example, `isblank('\t')` returns true, while `isblank('\n')` returns false.

On this page

Text Files

Learning Outcomes

After reading this section, you will be able to:

- Stream data using standard library functions to access persistent text

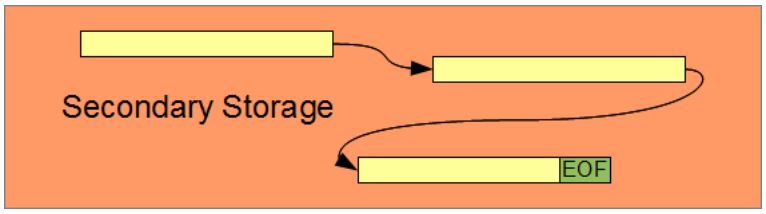
Introduction

Secondary storage retains its information when a computer is turned off and provides a mechanism for holding information beyond the execution of a program. Information in secondary storage can be accessed later by the same or a different program. This information resides in secondary memory in the form of files.

This chapter describes how to connect a program to a file, how to store information in that file and how to retrieve that information.

Files

A *file* is a named area of secondary storage. The file may be fragmented; that is, it may consist of several parts stored at different non-contiguous locations in secondary memory. A file does not necessarily occupy contiguous space on the storage device.



The byte is the fundamental storage unit of a file. The distinguishing feature of a file is the end-of-file mark. We refer to this mark as `EOF`. `EOF` typically has the value -1.

Text Format

A file holds information in either of two formats:

- text - readable and editable data
- binary - executable program code (beyond the scope of these notes)

Data stored in text format is suitable for displaying and modifying through a text editor. Files stored in text format are portable across platforms that share the same character set. A common standard is the IEC/ISO 646-1083 Invariant Code Set, which consists of:

- 52 upper and lower case alphabetic characters: A, B, ..., Z, a, b, ..., z
- 10 digits: 0, 1, ..., 9
- space
- null, line feed, carriage return, horizontal tab, vertical tab and form feed: \0, \l, \n, \t, \v, \f
- 29 graphic characters: ! # % ^ & * (_) - + = ~ [ ] ' | \ ; : " { } , . < > / ?

> **NOTE**
>
> This set excludes the $ and ` characters. The encoding for characters like $ and ` does not produce the same characters on all platforms (for more details see National Variants).

Sequential Access

The most common way to access data in a text file is sequentially, byte by byte. We process the file as a stream of bytes without skipping any byte until we reach the file's end-of-file (`EOF`) mark.

Connection

A C program connects to a file through an object of `FILE` type. The object holds information about the file and keeps track of the next position to be accessed. We use a library function to retrieve the address of the file object, store that address in a pointer and subsequently access the file through that pointer.

Allocating a pointer to a `FILE` object takes the form:

```
FILE *identifier;
```

Where `FILE` is the type of the FILE object and identifier is the name of the pointer to the `FILE` object. We call this pointer a handle to the object.

The structure type `FILE` is declared in the `<stdio.h>` header file. To allocate memory for a `FILE` pointer, we write:

```
#include <stdio.h>

FILE *fp = NULL;
```

We initialize the pointer `fp` to `NULL` as a precaution against premature dereferencing. If our program accesses data at `fp` before the connection to the file is open, our program may generate a **segmentation fault**.

> **NOTE**
>
> NULL is defined in the `<stdio.h>` header file

Opening a File

`fopen()` opens the named file and returns the address of the `FILE` object that connects to that file. The prototype for `fopen()` is:

```
FILE *fopen(const char file_name[], const char mode[]);
```

The first parameter holds the address of the file's name, which could be a ***string literal*** (a set of characters enclosed in a pair of **double quotes**). The second parameter holds the address of the connection mode, which could also be a string literal.

The most common connection modes are:

- **"r"** - read from the file
- **"w"** - write to the file: if the file exists, truncate its contents and then write; if the file does not exist, create a new file and then write to that file
- **"a"** - write to the end of the file: if the file exists, append to the end of the file; if the file does not exist, create it and then write to it

The less common connection modes for text files are:

- **"r+"** - opens the file for reading and possibly writing
- **"w+"** - opens the file for writing and possibly reading; if the file exists, truncates its contents and then writes to the file; if the file does not exist, creates a new file and then writes to that file
- **"a+"** - opens the file for writing to the end of the file and possibly reading; if the file exists, appends to the end of the file; if the file does not exist, creates it and then writes to the file

The mode parameter is enclosed in a pair of double quotes, not single quotes.

To open a file named alpha.txt for writing, we write:

```c
// Open a file
// openFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;

    fp = fopen("alpha.txt","w");

    if (fp != NULL)
    {
        // statements to be added later
    }
    else
    {
        printf("Failed to open file\n");
    }
    return 0;
}
```

`fopen()` returns `NULL` if it fails to connect to the file. `fopen()` can fail due to lack of permission, premature removal of the secondary storage medium or a full device.

Closing

`fclose()` disconnects the file from the host program. This library function takes as its only parameter the file pointer. The prototype for `fclose()` is:

```c
int fclose(FILE *);
```

If the file is open for writing or appending, `fclose()` writes any data remaining in the file's buffer to the file and appends the end of file mark after the last character written. If the file is open for reading, `fclose()` ignores any data left in the file's buffer and closes the connection.

To close a file named `alpha.txt` that is open for writing, we write:

```c
// Close an Opened file
// closeFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;

    fp = fopen("alpha.txt","w");

    if (fp != NULL)
    {
        // statements to be added later
        fclose(fp);
    }
    else
    {
        printf("Failed to open file\n");
    }
    return 0;
}
```

`fclose()` returns `0` if successful, `EOF` if unsuccessful. `fclose()` fails if the storage device is full, an I/O error occurs or the storage medium is prematurely removed.

Communication

The C library functions for communicating with an open file include:

- `fprintf()` - formatted write to file
- `fputc()` - write single character to file
- `fscanf()` - formatted read from file
- `fgetc()` - read single character from file

Writing

Formatted Writing

`fprintf()` writes data to an open file under format control. The prototype for this library function is:

```
int fprintf(FILE *, const char [], ...);
```

The first parameter receives the address of the `FILE` object. The second parameter receives the address of the string literal that specifies the format. This literal may contain text to be written directly to the file as well as conversion specifiers, if any, to be applied to the data values supplied as arguments.

For example:

```
// Writing to a File
// writeToFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    int sku = 4664;
    double price = 1.49;

    fp = fopen("alpha.txt","w");

    if (fp != NULL)
    {
        fprintf(fp, "sku = %d price = %10.2lf\n", sku, price);
        fclose(fp);
    }
    else
    {
        printf("Failed to open file\n");
    }
    return 0;
}
```

Unformatted Writing

`fputc()` writes a single character to an open file. The prototype for this library function is:

```
int fputc(int ch, FILE *fp);
```

`ch` receives a copy of the character to be written and `fp` receives the address of the `FILE` object. `fputc()` returns the character written, or `EOF` in the event of an error.

Reading

Formatted Reading

`fscanf()` reads a sequence of bytes from an open file under format control. The prototype for this library function is:

```
int fscanf(FILE *, const char [], ...);
```

The first parameter receives the address of the `FILE` object. The second parameter receives the address of the string literal that specifies the format. This literal contains the conversion specifiers to be used in translating the file data to data stored in memory.

For example:

```
// Reading from a File
// readFromFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    int sku;
    double price;

    fp = fopen("alpha.txt","r");

    if (fp != NULL)
    {
        fscanf(fp, "%d %lf", &sku, &price);
        printf("sku = %d price = %10.2lf\n", sku, price);
        fclose(fp);
    }
    else
    {
        printf("Failed to open file\n");
    }

    return 0;
}
```

Unformatted Reading

`fgetc()` reads a single character from an open file. The prototype for this library function is:

```
int fgetc(FILE *fp);
```

`fp` receives the address of the `FILE` object. `fgetc()` returns the character read; `EOF` in the event of an error.

State of a File Object

The C library functions for managing the state of a `FILE` object include:

- `rewind()` - rewind the file

- `feof()` - identify the end of the file

Rewind

`rewind()` resets the record pointer in the `FILE` object to the first byte in a file. The next byte to be accessed by the object will be the first byte in the file.

In other words, to jump to the beginning of a file, instead of disconnecting and re-connecting it, we simply rewind the file. The prototype for this library function is

```
void rewind(FILE *fp);
```

`fp` receives the address of the `FILE` object.

Consider a text file named produce.txt that contains:

```
 4664 1.49
 4419 1.29
 4011 0.59
```

The following program reads and displays this data, rewinds the file and reads and displays again:

```
// Reading from a file
// readFromFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    int sku;
    double price;

    fp = fopen("produce.txt","r");

    if (fp != NULL)
    {
        while( fscanf(fp, "%d%lf", &sku, &price) != EOF)
        {
            printf("%5d %6.2lf\n", sku, price);
        }

        rewind(fp);

        while (fscanf(fp, "%d%lf", &sku, &price) != EOF)
        {
            printf("%5d %6.2lf\n", sku, price);
        }

        fclose(fp);
    }
    else
    {
        printf("Failed to open file\n");
    }

    return 0;
}
```

End of File

`feof()` indicates whether or not the caller attempted to read the end-of-file mark; that is, read beyond the last character in the file. The prototype for this library function is:

```
int feof(FILE *fp);
```

`feof()` returns false (`0`) if the caller has not attempted to read the end-of-file mark; true if the caller attempted to read the end-of-file mark.

If the next byte to be read is the end-of-file mark, but the caller has not yet read the mark (that is, has only read the last character in the file), `feof()` returns false. In other words, to receive true, the caller must have attempted to read the end-of-file mark at least once.

Comparison

The library functions for communicating with files share many common properties with the functions for communicating with users directly. The functions belong to the same library, follow the same rules for format control and share a common syntax.

Return Type	Standard I/O	File I/O	Notes
`int`	`scanf(...)`	`fscanf(fp, ...)`	check to see if the return value is `EOF`
`int`	`printf(...)`	`fprintf(fp, ...)`	returns the number of characters written
`int`	`getchar()`	`fgetc(fp)`	check to see if the return value is `EOF` **before** converting it to `char` type
`int`	`putchar(ch)`	`fputc(ch, fp)`	check to see if the return value is `EOF`

On this page

Records and Files

Learning Outcomes

After reading this section, you will be able to:

- Stream data using standard library functions to access persistent text

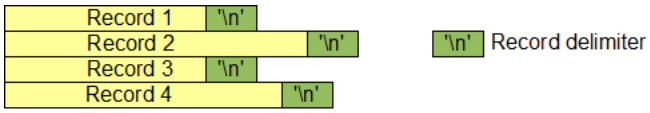
Introduction

Persistent data hierarchies typically consist of databases composed of files, which consist of records, which consist of fields, which consist of bytes, which are stored in bits. In files that hold a tabular structure, each record contains the same number of fields.

This chapter describes how to identify records and fields in a text file and how to retrieve tabular data.

Records

A **record** occupies a single line in a text file and holds all of the data associated with one chunk of information. The record is a sequence of characters that ends with a record delimiter. The typical record delimiter is the newline character (`\n`).



File = { Record 1, Record 2, Record 3, Record 4, ... , EOF }

Consider a text file named `produce.txt` containing information about items of produce in a grocery store. Each record consists of the **SKU** for a product and its **unit price**.

```
4664 1.49
4419 1.29
4011 0.59
```

To determine the number of records in this file, we count the number of newline (`'\n'`) characters:

```c
// Number of Records
// records.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    int c, nrecs;

    fp = fopen("produce.txt", "r");

    if (fp != NULL)
    {
        nrecs = 0;
        do {
            c = fgetc(fp);
```

```
            if (c != EOF)
            {
                if ((char)c == '\n')
                nrecs++;
            }
        } while (feof(fp) == 0);

        printf("%d records on file\n", nrecs);
        fclose(fp);
    }

    return 0;
}
```

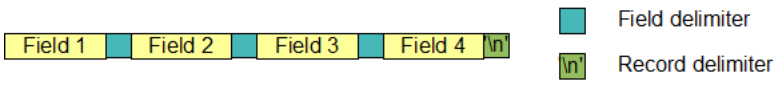
The above program produces the following output:

```
3 records on file
```

> **NOTE**
>
> Since this program determines the number of records in the file by counting the newline characters, to report the correct number of records, the last record in the file must end with a newline character. If the last record does not end with a newline character, the count will be off by one.

Fields

A **field** holds one element of information within a single record. We separate adjacent fields within a record by a **field delimiter**.



Record = { Field 1, Field 2, Field 3, Field 4, ... , Record Delimiter }

Consider the file named `produce.txt` (see above). Each record contains two fields: the first field holds the **SKU** and the second field holds the **unit price**. The field delimiter is a blank character.

The following program reads the fields of each record in the file and displays their contents:

```
// Record and Fields
// recordFields.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    int sku;
    double price;

    fp = fopen("produce.txt", "r");

    if (fp != NULL)
    {
        printf(" Produce Items\n"
        " =============\n\n"
        "SKU       Price\n"
        "---------------\n");

        while (fscanf(fp,"%d%lf\n", &sku, &price) == 2)
        {
            printf("%4d %10.2lf\n", sku, price);
```

```
        }

        fclose(fp);
    }

    return 0;
}
```

The above program produces the following output:

```
 Produce Items
 =============

SKU       Price
---------------
4664       1.49
4419       1.29
4011       0.59
```

Tables

A **table** is a set of records in which each record contains the same number of fields.

> **WARNING**
>
> If one of the fields in a record is a character field, the blank character might not be suitable as a field delimiter and we select a special character for that purpose.

Consider the file named `sale.txt` (its contents are listed below). Each record in this file contains three fields:

1. **SKU**
2. **price status** (a single character where a blank character represents the regular price and `*` represents a sale)
3. **unit price**

The field delimiter is the semi-colon character (`;`):

The following program reads each record from the file and displays the fields in a tabular format:

```
// Tabular Data
// table.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    int sku;
    char status;
    double price;

    fp = fopen("sale.txt","r");

    if (fp != NULL)
    {
        printf(" Produce Items\n"
        " =============\n\n"
        "SKU  Sale  Price\n"
        "----------------\n");
```

```
        while (fscanf(fp, "%d;%c;%lf", &sku, &status, &price) == 3)
        {
            printf("%4d %c %8.2lf\n", sku, status, price);
        }

        fclose(fp);
    }

    return 0;
}
```

The above program produces the following output:

```
 Produce Items
 =============

SKU  Sale  Price
----------------
4664 *      1.49
4419 *      1.29
4011        0.59
```

> **NOTE**
>
> We have included the field delimiters within `fscanf()`'s format string and these delimiter characters are discarded and not assigned to any variables.

On this page

Character Strings (C string)

Learning Outcomes

After reading this section, you will be able to:

- Design data collections using arrays to manage information efficiently
- Stream data using standard library functions to interact with users

Introduction

Although some original programming languages focused on processing numerical information, most languages include extensive features for processing textual data. Textual data involves sets of characters. These sets are often referred to as character strings. The C language libraries provide facilities for processing character strings, treated as arrays of characters with a special delimiter.

This chapter introduces these C-style strings, highlights their distinguishing feature, and notes the advantage of using character strings to pass textual data from one function to another. This chapter includes the conversion specifiers for the input and output of character strings.

Definition (review)

A string is a `char` array with a special property which is a terminator element that follows the last ***meaningful character*** in the string. We refer to this terminator as the **null terminator** and identify it by the escape sequence `'\0'`.

char																
																\0

> **TERM DEFINITION**
>
> The term "Meaningful Characters" in these notes refers to the actual data content you want to manage in the C string character array.

The **null terminator** has the integral value of `0` on any host platform (in its collating sequence). All of its bits are 0's. The null terminator occupies the first position in the ASCII and EBCDIC.

The index identifying the null terminator element is the same as the number of meaningful characters in the string (including spaces between words).

char name																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	**17**
M	y		n	a	m	e		i	s		A	r	n	o	l	d	**\0**

> **HINT**
>
> The number of memory locations occupied by a C string (`char`) is one more than the number of meaningful characters in the string so as to hold the null terminator.

Allocating Memory

We allocate memory for a C string in the same way that we allocate memory for an array. Since the **null terminator** is one of the elements in the array, we must allocate memory for one extra character than the number of meaningful characters.

For example, to allocate memory for a string with up to `30` meaningful characters, we write:

```
char name[31]; // 30 chars plus 1 char for the null terminator byte
```

Initializing Memory

To initialize a string at the time of memory allocation, we follow the definition with the assignment operator and the set of initial characters enclosed in braces.

```
const char name[31] = {'M','y',' ','n','a','m','e',' ','i','s',' ',
                       'A','r','n','o','l','d','\0'};
```

For a more compact form we enclosed the list of meaningful characters in double quotes.

```
const char name[31] = "My name is Arnold";  // null-byte is automatically appended
```

The C compiler copies the characters in the string literal into the character string and appends the null-byte terminator after the last copied character.

char name																										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	**17**	18	19	20	21	22	23	24	25	26	27
M	y		n	a	m	e		i	s		A	r	n	o	l	d	**\0**	0	0	0	0	0	0	0	0	0	0

Since the number of initializers (18) is less than the number of elements (31) available, the compiler fills the uninitialized elements with 0's.

String Handling

Arrays of numbers require a separate variable to hold the number of elements that are filled. Unlike arrays of numbers, character strings do not require a separate variable for sizing. In iterations on the characters in a string, we check for the presence of the null terminator in our test conditions.

Iterations

The following program displays the string stored in name[31] character by character:

```
// Iterations on character strings
// string_iterations.c

#include <stdio.h>

int main(void)
{
    int i;
    const char name[31] = "My name is Arnold";

    for (i = 0; name[i] != '\0'; i++)
    {
        printf("%c", name[i]);
    }

    putchar('\n');
```

```
    return 0;
}
```

The above program produces the following output:

```
My name is Arnold
```

Functions

Using a character string instead of an array of characters with a separate sizing variable achieves a more compact argument list for function calls. For example:

```c
// Strings To Functions
// string_to_function.c

#include <stdio.h>
void print(const char name[]);

int main(void)
{
    int i;
    const char name[31] = "My name is Arnold";

    print(name);
    return 0;
}

void print(const char name[])
{
    int i;

    for (i = 0; name[i] != '\0'; i++)
    {
        printf("%c", name[i]);
    }

    putchar('\n');
}
```

The above program produces the following output:

```
My name is Arnold
```

Formatted String Input

The `scanf()` and `fscanf()` library functions support conversion specifiers particularly designed for character string input. These specifiers are:

- `%s` - whitespace delimited set
- `%[]` - rule delimited set

The corresponding argument for these specifiers is the address of the string to be populated from the input stream.

%s

The `%s` conversion specifier

- reads all characters ***until*** the first whitespace character
- stores the characters read in the char array identified by the corresponding argument

- stores the null terminator in the char array after accepting the last character
- leaves the delimiting whitespace character and any subsequent characters in the input buffer

For example:

```
char name[31];
scanf("%s", name);  // <=== User enters: My name is Arnold
```

The `scanf()` function will stop accepting input after the character `y` and stores the following:

																char name										
0	1	**2**	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
M	y	**\0**																									

The characters `' name is Arnold'` remain in the input buffer.

A qualifier on the conversion specifier limits the number of characters accepted. For instance, %10s reads no more than 10 characters:

```
char name[31];
scanf("%10s", name); // <=== User enters: Schwartzenegger
```

The `scanf()` function will stop accepting input after the character `n` and stores the following:

																char name										
0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
S	c	h	w	a	r	t	z	e	n	**\0**																	

By specifying the maximum number of characters to be read at less than 31, we ensure that `scanf()` does not exceed the memory allocated for the string.

`%s` discards all leading whitespace characters.

For example, if enter many spaces before the Schwartzenegger value:

```
char name[31];
scanf("%10s", name);  // <=== User enters: '          Schwartzenegger'
```

Just as before, the `scanf()` function will stop accepting input after the character `n` but will also discard the leading spaces entered and stores the following:

																char name										
0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
S	c	h	w	a	r	t	z	e	n	**\0**																	

Because `%s` discards ***leading whitespace***, it cannot accept an empty string; that is, `%s` does treat a `'\n'` in an empty input buffer as an empty string. If the buffer only contains `'\n'`, `scanf("%10s", name)` discards the `'\n'` and waits for non-whitespace input followed by another `'\n'`.

%[]

The `%[]` conversion specifier accepts input consisting only of a set of pre-selected characters. The brackets contain the admissible and/or inadmissible characters. The symbol `^` prefaces the list of **inadmissible** characters. The symbol `-` identifies a range of characters in an **inclusive** set.

For example, the `%[^\n]` conversion specifier:

- reads all characters until the newline ('\n')
- stores the characters read in the char array identified by the corresponding argument
- stores the null terminator in the char array after accepting the last character
- leaves the delimiting character (`'\n'`) in the input buffer

For example:

```
char name[31];
scanf("%[^\n]", name);  // <=== User enters: My name is Arnold
```

The `scanf()` function accepts the full line an stores:

char name																									
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	**17**	18	19	20	21	22	23	24	25	26	27
M	y		n	a	m	e		i	s		A	r	n	o	l	d	**\0**										

A qualifier on this conversion specifier before the opening bracket limits the number of characters accepted. For instance, `%10[^\n]` reads no more than 10 characters:

```
char name[31];
scanf("%10[^\n]", name);  // <=== User enters: My name is Arnold
```

The `scanf()` function will store:

char name																									
0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
M	y		n	a	m	e		i	s	**\0**																	

We specify the maximum number of characters as the qualifier to ensure that `scanf()` does not store more characters than the allocated memory for the array size.

`%[ ]`, like `%s`, ignores any leading whitespace characters.

For example:

```
char name[31];
scanf("%10[^\n]", name);  // <=== User enters: '          My name is Arnold'
```

The `scanf()` function will store:

char name																									

0	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
M	y		n	a	m	e		i	s	**\0**																	

Caution

Because `%[ ]` ignores leading whitespace, it cannot accept an empty string; that is, `%[^\n]` does treat a `'\n'` in an empty input buffer as an empty string. If the input buffer only contains `'\n'`, `scanf("%[^\n]", name)`, unlike `%s`, returns `0` and leaves `name` unchanged.

Example:

Consider a text file named spring.dat that contains:

```
Light Jacket
Long-Sleeved Shirts
Large Skateboards
```

The following program reads and displays this data:

```c
// Reading from a file
// readFromFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    char phrase[61];

    fp = fopen("spring.dat","r");

    if (fp != NULL)
    {
        while (fscanf(fp, "%60[^\n]%*c", phrase) != EOF)
        printf("%s\n", phrase);

        fclose(fp);
    }
    else
    {
        printf("Failed to open file\n");
    }

    return 0;
}
```

The above program produces the following output:

```
Light Jacket
Long-Sleeved Shirts
Large Skateboards
```

String Output

Formatted Output

The `printf()` and `fprintf()` library functions support the `%s` conversion specifier for character string output. The corresponding argument is the **address** of the character string or strings literal. Under this specifier `printf()` displays all of the characters from the address provided up to but **excluding** the null

terminator byte. For example:

```c
// Displaying Strings
// displayStrings.c

#include <stdio.h>

int main(void)
{
    const char name[31] = "My name is Arnold";

    printf("%s\n", name);

    return 0;
}
```

The above program produces the following output:

```
My name is Arnold
```

```c
// Writing to a File
// writeToFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    const char phrase[] = "My name is Arnold";

    fp = fopen("alpha.txt","w");

    if (fp != NULL)
    {
        fprintf(fp, "%s\n", phrase);
        fclose(fp);
    }
    else
    {
        printf("Failed to open file\n");
    }

    return 0;
}
```

Qualifiers

Qualifiers on the `%s` specifier add detail control:

- `%20s` displays a string **right**-justified in a field of **20**
- `%-20s` displays a string **left**-justified in a field of **20**
- `%20.10s` displays the ***first* 10** characters of a string **right**-justified in a field of **20**
- `%-20.10s` displays the ***first* 10** characters of a string **left**-justified in a field of **20**

Unformatted Output

The `puts()` and `fputs()` library functions output a character string to the standard or specified output device respectively.

puts

The prototype for puts() is:

```
int puts(const char *);
```

The parameter receives the address of the character string to be displayed. For example:

```
// Displaying Lines
// puts.c

#include <stdio.h>

int main(void)
{
    const char name[31] = "My name is Arnold";

    puts(name);

    return 0;
}
```

The above program produces the following output:

```
My name is Arnold
```

fputs

`fputs()` writes a null-terminated string to a file. The prototype for `fputs()` is:

```
int fputs(const char *str, FILE *fp);
```

`str` receives the address of the string to be written and `fp` receives the address of the `FILE` object. `fputs()` returns a non-negative value if successful; `EOF` in the event of an error.

On this page

String Library

Learning Outcomes

After reading this section, you will be able to:

- Implement algorithms using standard library procedures to incorporate existing technology
- Stream data using standard library functions to interact with users

Introduction

The standard library that ships with C compilers and processes character strings is called the string library. The functions in this library perform fundamental operations on character strings, which include copying one string to another, adding one string to another and comparing one string to another.

This chapter describes four library functions that operate on character strings: determining the length, copying one to another, comparing one to another and concatenating one to another.

String Functions

The functions of the string library include:

- `strlen()` - returns the number of characters in a character string
- `strcpy()` - copies one character string to another
- `strcmp()` - compares one character string to another
- `strcat()` - concatenates one character string to another

> **NOTE**
>
> These are four of the most common string functions, but there are many available in the string library.

The header file that contains the prototypes for these library functions is:

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

String Length

The `strlen()` function returns the number of characters in the character string excluding the **null terminator byte**. That is, `strlen()` returns the index of the **null terminator byte**.

The following program finds the length of the input string and reverses its contents:

```
// Reverse a string
// reverse_string.c

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

int main(void)
{
    int i, len;
    char str[31], rev[31];
```

```
    printf("Enter a string   : ");
    scanf("%30[^\n]%*c", str);

    printf("In reverse order : ");
    len = strlen(str);

    for (i = len - 1; i >= 0; i--)
    {
        rev[len - 1 - i] = str[i];
    }

    rev[len] = '\0';
    puts(rev);

    return 0;
}
```

The above program produces the following output:

```
Enter a string   : strlen
In reverse order : nelrts
```

String Copy

The `strcpy()` function receives two addresses and copies the string at the second address into the memory locations starting at the first address. `strcpy()` returns the address of the destination string.

We are responsible for ensuring that there is sufficient room in the destination string to hold all of the characters in the source string **including the null terminator byte**.

The following program copies the input string only if it contains 20 characters or less:

```
// Copy a string
// copy_string.c

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

int main(void)
{
    char str[31], copy[21] = "?";
    int i, len;

    printf("Source : ");
    scanf("%30[^\n]%*c", str);

    len = strlen(str);

    if (len < 21)
    {
        strcpy(copy, str);
        printf("Copy : %s\n", copy);
    }
    else
    {
        printf("* No room *\n");
        printf("Copy : %s\n", copy);
    }

    return 0;
}
```

Sample run-through #1 of the above program:

```
Source : strcpy
Copy   : strcpy
```

Sample run-through #2 of the above program:

```
Source : this string is too long
* No room *
Copy   : ?
```

String Compare

The `strcmp()` function receives two addresses and compares the string at the first address to the string at the second address.

`strcmp()` returns:

- **0** if they are identical
- **a negative value** if the first non-matching character in the first string is, under the host computer's collating sequence, **lower** than the character with the same index in the second string
- **a positive value** if the first non-matching character in the first string is, under the host computer's collating sequence, **higher** than the character with the same index in the second string

For example:

```c
// Compare Two Strings
// compare_string.c

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

int main(void)
{
    char str1[31], str2[31];
    int i, len;

    printf("Enter first string  : ");
    scanf("%30[^\n]%*c", str1);

    printf("Enter second string : ");
    scanf("%30[^\n]%*c", str2);

    if (strcmp(str1, str2) < 0)
    {
        printf("%s precedes %s\n", str1, str2);
    }
    else if(strcmp(str1, str2) > 0)
    {
        printf("%s follows %s\n", str1, str2);
    }
    else
    {
        printf("%s matches %s\n", str1, str2);
    }

    return 0;
}
```

Sample run-through #1 of the above program:

```
Enter first string  : elephant
Enter second string : elephants
elephant precedes elephants
```

Sample run-through #2 of the above program:

```
Enter first string  : elephant
Enter second string : elephant
elephant matches elephant
```

Sample run-through #3 of the above program:

```
Enter first string  : elephants
Enter second string : elephant
elephants follows elephant
```

String Concatenate

The `strcat()` function receives two addresses and concatenates the string at the second address to the string at the first address. `strcat()` returns the address of the concatenated string.

We are responsible for ensuring that the destination string has enough room to hold the concatenated result **along with the null terminator byte**.

For example:

```c
// Concatenate two strings
// concatenate.c

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

int main(void)
{
    int i, len;
    char surname[31], fullName[62];

    printf("Given name  : ");
    scanf("%30[^\n]%*c", fullName);

    printf("Surname     : ");
    scanf("%30[^\n]%*c", surname);

    strcat(fullName, " ");
    strcat(fullName, surname);
    printf("Full name is %s\n", fullName);

    return 0;
}
```

Sample run-through of the above program:

```
Given name  : Jane
Surname     : Doe
Full name is Jane Doe
```

> **NOTE**
>
> We have allocated 62 characters to accommodate 30+30 characters plus the blank space separator and the null terminator byte.

More Input and Output

Learning Outcomes

After reading this section, you will be able to:

- Implement algorithms using standard library procedures to incorporate existing technology
- Stream data using standard library functions to interact with users and access persistent text

Introduction

The standard input and output library (stdio) that ships with C compilers provides comprehensive support for communicating with the user and with secondary storage. This support includes numerical as well as character string processing under format control and optionally line by line processing without format control. For platforms that don't support line-by-line input processing, we write our own custom procedures.

This chapter reviews the conversion specifiers for formatted input and output along with the library functions for line by line input and output. Specifiers not covered in previous chapters are included here. This chapter concludes with two custom functions for input that safeguard line mismatching and memory overflow.

Input

The stdio library functions for processing input are:

- `scanf()` - input from standard input under format control
- `fscanf()` - input from file under format control
- `getchar()` - character by character input from standard input (see Input Functions)
- `fgetc()` - character by character input from file (see Input Functions)
- `gets_s()` - line by line input from standard input (not universally implemented)
- `fgets()` - line by line input from file

> **NOTE**
>
> Typically, ***standard input*** refers to the **keyboard**.

Formatted Input

The `scanf(...)` and `fscanf(...)` functions accept data from the standard input device or secondary storage respectively and store that data in memory at the address specified in their argument list. Their prototypes are:

```
int scanf(const char *format, address);
int fscanf(FILE *, const char *format, address);
```

format

`format` receives a string literal that describes how to convert input text into data stored in memory. Calls to these functions can take multiple arguments. `format` contains the conversion specifier(s) for translating the input characters. Conversion specifiers begin with a `%` symbol and identify the type of the destination variable. The possible specifiers are listed below. Other specifiers may be found on the web.

Specifier	Input Text is a	Destination Type
`%c`	character	`char`, `char []`
`%d`	decimal	`int`, `short`, `long`, `long long`
`%i`	integer	`int`, `short`, `long`, `long long`
`%o`	unsigned octal	`unsigned int`, `short`, `long`, `long long`
`%x`	unsigned hexadecimal	`unsigned int`, `short`, `long`, `long long`
`%u`	unsigned decimal	`unsigned int`, `short`, `long`, `long long`
`%n`	--	`int`, `short`, `long`, `long long`
`%f` `%e` `%g` `%a`	floating-point	`float`, `double`, `long double`
`%s`	character string	`char []`
`%[ ]` `%[^ ]`	character string	`char []`
`%p`	address	any type

> **REMARKS**
>
> - `%n` does not read any characters but instead returns the number of characters processed.
> - `%f`, `%e`, `%g` and `%a` treat floating-point input identically.
> - *Size specifiers* also apply to `%i` , `%o`, `%x`, `%u` and `%n`, but are not listed here.

address

`address` receives the address of the destination variable. We specify a separate address argument for each conversion specifier in the format string.

Conversion Control

We may insert control characters between the `%` and the conversion character. The general form of a conversion specification is:

```
% * width size conversion_character
```

The three control characters are:

1. ***** - suppresses storage of the converted data (**discards** it without storing it)
2. ***width*** - specifies the maximum number of characters to be interpreted
3. ***size*** - specifies the size of the storage type

> **EXCEPTION**
>
> A conversion specifier that includes a `*` does not have a corresponding address in the argument list. This is an exception to the matching conversion-specifier/argument rule.

The size specifiers covered in this course are listed below. Others may be found on the web.

Specifier with Size	Input Text is a	Destination Type

Specifier with Size	Input Text is a	Destination Type
`%hhd` `%hhi`	very short decimal	`char`
`%hd` `%hi`	short decimal	`short`
`%ld` `%li`	long decimal	`long`
`%lld` `%lli`	very long decimal	`long long`
`%lf` `%le` `%lg` `%la`	floating-point	`double`
`%Lf` `%Le` `%Lg` `%La`	floating-point	`long double`
`%hhu` `%hho` `%hhx`	unsigned very short decimal	`unsigned char`
`%hu` `%ho` `%hx`	unsigned short decimal	`unsigned short`
`%lu` `%lo` `%lx`	unsigned long decimal	`unsigned long`
`%llu` `%llo` `%llx`	unsigned very long decimal	`unsigned long long`
`%hhn`	character string	`char`
`%hn`	character string	`short`
`%ln`	character string	`long`
`%lln`	character string	`long long`

Problems with `%c`

`scanf()` and `fscanf()` only extract the characters they need from the buffer, but problems arise with `%c` conversions. Consider the following program. On reading an integer value, `scanf()` leaves the newline character (`'\n'`)in the input buffer. Since the next call to `scanf()` starts with a `%c` specifier, `scanf()` treats the unprocessed `'\n'` as the input character. As a result, this program never collects the tax status input from the input buffer.

```c
/* scanf with %c Specification
* scanf_c.c
*/

#include <stdio.h>

int main(void)
{
    int items;
    char status; // tax status g or p

    printf("Number of items : ");
    scanf("%d", &items);

    printf("Status : ");
    scanf("%c", &status);   // ERROR: assigns \n to variable 'status'
                            //        and will not pause for user input

    printf("%d items (%c)\n", items, status);

    return 0;
}
```

The above program produces the following output:

```
Number of items : 25
Status : 25 items (
)
```

> **NOTE**
>
> Notice how the newline character (`'\n'`) (which was assigned to the tax **status** variable) places the closing parenthesis on a newline.

There are different ways to handle unprocessed `'\n'` characters. Some are listed in the ***code snippets*** below.

> **HELPFUL HINT**
>
> A **space character** *before* a conversion specifier forces the skipping of ***all leading whitespace*** before the next conversion.
> For example, `"%c"` directs `scanf()` to skip any whitespace characters before attempting to read the next non-whitespace character.

Method-1:

```
scanf("%d", &items);
scanf("%c%c", &junk, &status); // store one character in junk first
```

Method-2:

```
scanf("%d", &items);
scanf("%*c%c", &status);       // discard(ignore) one character first
```

Method-3:

```
scanf("%d", &items);
scanf(" %c", &status);         // discard(ignore) all whitespace first
```

Method-4:

```
scanf("%d%*c", &items);        // discard(ignore) newline ('\n')
scanf("%c", &status);
```

Method-5:

```
scanf("%d", &items);
clear();                       // call a custom function to clear the buffer
scanf("%c", &status);
```

`"%*c%c"` discards/ignores **one** character and accepts the next.
`"%c"` discards/ignores **all** whitespace characters before the next non-whitespace character.

One corrected version of the above program is:

```
// scanf with %c Specification
// scanf_cc.c

#include <stdio.h>

int main(void)
{
```

```
    int items;
    char status; // tax status g or p

    printf("Number of items : ");
    scanf("%d", &items);

    printf("Status : ");
    scanf(" %c", &status);   // note the space

    printf("%d items (%c)\n", items, status);

    return 0;
}
```

Unformatted Input

The library functions for processing unformatted input are:

- `getchar()` - character by character input from standard input (see Input Functions)
- `fgetc()` - character by character input from file (see Input Functions)
- `gets_s()` - line by line input from standard input (not universally implemented)
- `fgets()` - line by line input from file

gets_s (Optional)

The `gets_s()` function...

- accepts an empty string
- assumes no more than the specified number of characters
- reads the `'\n'` as the delimiter
- replaces the delimiter with the null terminator

`gets_s()` takes two arguments. Its prototype is:

```
char *gets_s(char *address, int n);
```

The first parameter receives the address of the string to be filled. The second parameter receives the maximum number of characters that can be stored including the null terminator byte. On success, this function returns the address of the filled string:

```
// Read and Display Lines
// gets_s.c

#include <stdio.h>

int main(void)
{
    char first_name[21];
    char last_name[21];

    printf("First Name : ");
    gets_s(first_name, 21);

    printf("Last Name  : ");
    gets_s(last_name, 21);

    puts(first_name);
    puts(last_name);

    return 0;
}
```

The above program produces the following output:

```
First Name : Arnold
Last Name  : Schwartzenegger
Arnold
Schwartzenegger
```

> **IMPORTANT**
>
> The behaviour of `gets_s()` is **undefined** if the user inputs a line **longer** than the allocated string. On a Windows platform, this function **crashes**.
> The standard recommends use of `fgets()` instead of `gets_s()`.

fgets

The `fgets()` function...

- reads a stream of bytes from the specified file
- accepts an empty string
- accepts no more than the specified number of characters
- reads until the `'\n'` delimiter
- includes the `'\n'` delimiter in the character string
- does not discard the `'\n'` delimiter
- adds the null terminator byte to the character string

The prototype for this function is:

```
char* fgets(char str[], int max, FILE *fp);
```

`str` receives the address of the string to be filled.
`max` receives the maximum number of bytes in `str` including space for the null terminator byte.
`fp` receives the address of the `FILE` object.
`fgets()` appends the null terminator byte to the stored string. `fgets()` *returns* the **address** of `str` if successful, otherwise, `NULL` in the event of an end of file or read error.

Output

The stdio library functions for processing output are:

- `printf()` - output to standard output under format control
- `fprintf()` - output to a file under format control
- `putchar()` - character by character output to standard output (see Output Functions)
- `fputc()` - character by character output to a file (see Output Functions)
- `puts()` - character string output to standard output (see Output Functions)
- `fputs()` - character string output to a file (see Output Functions)

Formatted Output

The `printf(...)` and `fprintf(...)` functions report the value of the variable(s) or expression(s) in the argument list to the standard output device or the specified file respectively. Their prototypes take the form:

```
int printf(const char *format, ...);
int fprintf(FILE *, const char *format, ...);
```

format

`format` is a string literal containing conversion specifiers and any characters to be output directly. Each conversion specifier begins with a `%` symbol and identifies the type of the source variable. The order of the specifiers matches the order of the values received.

Conversion Specifiers

The conversion specifiers include:

Specifier	Output Text is a	Use with Type
`%c`	character	`char`
`%d`	signed decimal	`int`, `short`, `long`, `long long`
`%i`	signed integer	`int`, `short`, `long`, `long long`
`%u`	unsigned decimal	`unsigned int`, `short`, `long`, `long long`
`%o`	unsigned octal	`unsigned int`, `short`, `long`, `long long`
`%x`	unsigned hexadecimal	`unsigned int`, `short`, `long`, `long long`
`%X`	unsigned hexadecimal (uppercase)	`unsigned int`, `short`, `long`, `long long`
`%n`	--	`int *`
`%f`	floating-point	`float`, `double`, `long double`
`%F`	floating-point (uppercase)	`float`, `double`, `long double`
`%e`	scientific floating-point	`float`, `double`, `long double`
`%E`	scientific floating-point (uppercase)	`float`, `double`, `long double`
`%g`	shortest floating-point	`float`, `double`, `long double`
`%G`	shortest floating-point (uppercase)	`float`, `double`, `long double`
`%a`	hexadecimal floating-point	`float`, `double`, `long double`
`%A`	hexadecimal floating-point (uppercase)	`float`, `double`, `long double`
`%s`	string of characters	`char *`
`%%`	the character %	`char *`
`%p`	address	--

REMARKS

- `%n` does not output any characters but instead returns the number of characters processed so far.
- ***Scientific*** (`%e` `%E`) refers to output in mantissa/exponent form `d.dddEdd` (for example, `0.123e3`, which stands for `0.123 x 103` or `123.0`).
- ***General*** (`%g` `%G`) refers to output in the shortest form possible; decimal or mantissa/exponent (for example, `0.123e-5` rather than `0.00000123` and `3.1` rather than `0.31e1`).

Conversion Control

We may insert control characters between the `%` and the conversion character. The general form of a conversion specification is:

```
% flags width . precision size conversion_character
```

The five control characters are:

1. ***flags***
   - prescribes left justification of the converted value in its field
   - `0` pads the field width with **leading zeros**
2. ***width*** sets the minimum field width within which to format the value (overriding with a wider field only if necessary). Pads the converted value on the left (or right, for left alignment). The padding character used is either a **space** or `0` if the padding flag is on
3. `.` separates the field's width from the field's precision
4. ***precision*** sets the number of digits to be printed after the decimal point for `%f` conversions and the minimum number of digits to be printed for an **integer** (adding leading zeros if necessary). A value of `0` suppresses the printing of the decimal point in a `%f` conversion. An `*` instead of a number applies the value from the next argument in the argument list
5. ***size*** identifies the *minimum* size of the type being output

The size specifiers covered in this course are listed below. Others may be found on the web

Specifier with Size	Output Text is	Use with Type
`%hhd` `%hhi`	very short decimal	`char`
`%hd` `%hi`	short decimal	`short`
`%ld` `%li`	long decimal	`long`
`%lld` `%lli`	very long decimal	`long long`
`%lf` `%lF` `%le` `%lE` `%lg` `%lG` `%la` `%lA`	floating-point	`double`
`%Lf` `%LF` `%Le` `%LE` `%Lg` `%LG` `%La` `%LA`	floating-point	`long double`
`%hhu` `%hho` `%hhx` `%hhX`	unsigned very short decimal	`unsigned char`
`%hu` `%ho` `%hx` `%hhX`	unsigned short decimal	`unsigned short`
`%lu` `%lo` `%lx` `%hhX`	unsigned long decimal	`unsigned long`
`%llu` `%llo` `%llx` `%hhX`	unsigned very long decimal	`unsigned long long`
`%hhn`	character string	`char`
`%hn`	character string	`short`
`%ln`	character string	`long`
`%lln`	character string	`long long`

Custom Input (Optional)

Mismatching Line Input

Managing line-oriented input helps in debugging. Consider a set of input lines some of which contain incorrect input. Ideally, a one-to-one correspondence should exist between the lines of input data and the lines read by the program. Even if the user inputs a line incorrectly, subsequent correct input may still be acceptable. In other words, incorrect input on one line should not cause incorrect reading of subsequent lines.

Ideally, line by line input should:

- store characters only to a specified maximum
- accept an empty string
- read the `'\n'` as the line delimiter
- discard the delimiting character along with any characters that overflow memory
- append the null terminator to the set of characters stored

The following code meets all of these conditions:

```c
// Custom Line-Oriented Input
// getline.c

#include <stdio.h>

// getline accepts a newline terminated
// string s of up to max - 1 characters,
// adds the null terminator and discards
// the remaining characters in the input
// buffer including terminating character
char *getline(char *s, int n)
{
    int i, c;

    for (i = 0; i < n - 1 && (c =getchar()) != EOF && c != (int)'\n'; i++)
    {
        s[i] = c;
    }

    s[i] = '\0';

    while (n > 1 && c != EOF && c != (int)'\n')
    {
        c = getchar();
    }

    return c != EOF ? s : NULL;
}

int main(void)
{
    char first_name[11];
    char last_name[11];

    printf("First Name : ");
    getline(first_name, 11);

    printf("Last Name  : ");
    getline(last_name, 11);

    puts(first_name);
    puts(last_name);

    return 0;
}
```

The above program produces the following output:

```
First Name : Arnold
Last Name  : Schwartzenegger
Arnold
Schwartzen
```

This function, unlike `gets_s()` has well-defined behavior if the number of characters entered exceeds the amount of memory available to store the string.

Insufficient Memory (Optional)

Consider the file named `spring.dat`, the contents of which are listed below. Each record in this file contains three fields: the first field holds the `quantity`, the second field holds a C string describing the item (`label`) and the third field holds the unit `price` of the item. The field delimiter is the semicolon (`;`) character:

```
2;Light Jacket;95.89
3;Long Pants;67.89
2;Large Duster;45.98
```

The following program reads each record from the file and displays the fields in a tabular format:

```c
// Tabular Data
// table.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    char label [14];
    int n;
    double price;

    fp = fopen("spring.txt","r");
    if (fp != NULL)
    {
        printf("    Spring Items\n"
            "    ============\n\n"
            "No Description  Price\n"
            "--------------------\n");

        while (fscanf(fp, "%d;%13[^;];%lf%*c", &n, label, &price) == 3)
        {
            printf("%2d %-13s%5.2lf\n", n, label, price);
        }

        fclose(fp);
    }

    return 0;
}
```

The above program produces the following output:

```
    Spring Items
    ============

No Description  Price
--------------------
 2 Light Jacket 95.89
 3 Long Pants   67.89
 2 Large Duster 45.98
```

> **NOTE**
>
> Notice how the field delimiters have been embedded within `fscanf()`'s format string.

Safe Coding

The above program executes successfully only if the descriptive strings in the file do not contain **more than 13 characters**. The data in a different file that contains longer labels will not fit into the space allocated by the program.

To process any file and safeguard against ***memory overflow***, we can modify the program to **skip the extra characters** in the description field that exceeds the memory allocated for the `label` C string character array variable. We do so by reading each record in two separate statements:

```c
// Insufficient Memory
// table_plus.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    char label [14];
    int n;
    double price;
    char c;

    fp = fopen("spring.txt","r");

    if (fp != NULL)
    {
        printf("    Spring Items\n"
            "    ============\n\n"
            "No Description  Price\n"
            "--------------------\n");

        while (fscanf(fp,"%d;%13[^;]%c", &n, label, &c) == 3)
        {
            if (c == ';')
            {
                fscanf(fp,"%lf\n", &price);
            }
            else
            {
                fscanf(fp, "%*[^;];%lf%*c", &price);
            }

            printf("%2d %-13s%5.2lf\n", n, label, price);
        }

        fclose(fp);
    }

    return 0;
}
```

The above program produces the following output:

```
     Spring Items
     ============

No Description  Price
--------------------
 2 Light Jacket 95.89
```

```
 3 Long Pants   67.89
 2 Large Duster 45.98
```

The first statement reads the first two fields stopping at the second delimiter or once memory is full, whichever comes first. If the statement has encountered the second delimiter, the second statement reads the `price`; if not, the alternate version of the second statement skips the remaining characters in the field and the second delimiter and only then reads the `price`.

The program stops reading altogether as soon as it encounters a record with other than 3 input values - the quantity, the descriptive string and the second delimiter.

On this page

Pointers, Arrays and Structs

Learning Outcomes

After reading this section, you will be able to:

- Connect procedures using pass-by-value and pass-by-address semantics to build a complete program
- Design data collections using arrays and structures to manage information efficiently

Introduction

A distinguishing feature of the C programming language is its direct access to primary memory. The core language defines a separate pointer type for each primitive type and C compilers accept pointer types for each derived type. This enables programmers to store the address of any type - primitive or derived - in a pointer variable. The pointer types corresponding to the primitive types are:

- `char*`
- `int*`
- `float*`
- `double*`

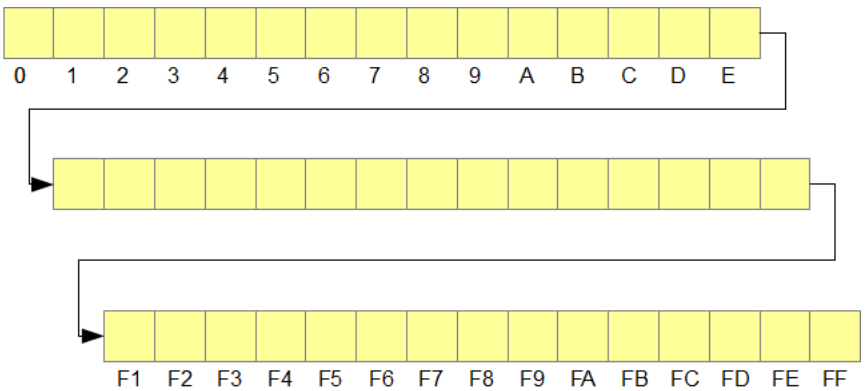
The type of a pointer variable determines the number of bytes to interpret when converting into a data type the bit string in primary memory that starts at the specified address.

Pointers and arrays are closely related in the C language. The name of an array holds the address of the start of the array; that is, the name of the array is a pointer. Since arrays by definition store element data contiguously in memory, we can access any array element using pointer syntax.

This chapter examines this relationship between pointers, arrays and structures in more detail. The chapter reviews pointer and array syntax and shows where syntax is interchangeable.

Review

We can model RAM as a linear map and use addresses on this map to identify bytes of information stored within memory. For instance, in 512Mb of RAM, the address 0 identifies the first byte, the address 1 identifies the second byte and address 512Mb-1 identifies the last byte.



Pointers

Syntax

A pointer is a variable that stores an address. To allocate memory for a pointer that holds the address of a variable of type `double`, we write:

```
double *p;
```

We say that `p` ***points*** to a `double`. We store the address of the `double` in `p` as follows:

```
double x = 456.7;
double *p = &x;
```

If the address is unknown, we initialize the pointer to `NULL`:

```
double *p = NULL;
// ...
double x = 456.7;
p = &x;
```

We can access data through a pointer once the pointer contains a valid address. To access the data at the address pointed to, we ***dereference*** the pointer using the `*` operator:

```
double *p = NULL;
double x = 456.7;

p = &x;

printf("Value stored in x %.1lf", *p);
```

The above code snippet would produce the following output:

```
Value stored in x 456.7
```

Arrays

The C language stores the elements of an array contiguously in memory. There is no empty space between adjacent elements and all elements share a common data type. The name of the array holds the address of the start of the array. We can determine the address of any element from the name of the array, the element's index and the array type.

The index in array subscript notation refers to the offset into the array in terms of the number of elements; that is, the number of elements beyond the start of the array:

```
double a[10]; // 10 elements of type double = 80 bytes
int i = 3;

a[i] = 5.4; // store 5.4 in memory location 3 x 8 bytes
            // beyond the address of a
```

For example, if the address of `a` is `0x4e55a00` the address of `a[i]` is `0x4e55a18`.

We can store the address of an array in a separate pointer:

```
double a[10];  // 10 elements of type double = 80 bytes
int i = 3;
double *p = a; // store address of a in p
p[i] = 5.4;    // store 5.4 in memory location 3 x 8 bytes
               // beyond the address of a
```

> **NOTE**
>
> `a` and `p` are interchangeable. `p` may also be used as an array!

Equivalence

Function Parameters

A function parameter that receives the *address of an array* is a **pointer**. In a function header, *array and pointer* notations are **equivalent**:

```
type identifier(type identifier[])
type identifier(type *identifier)
```

For example, the syntax in the first function prototype is **equivalent** to the second function prototype:

```
void foo(int a[])  // These two function prototypes are
void foo(int *b)   // equivalent and translate to the same thing!
```

> **NOTE**
>
> If parameter `b` from the second function prototype example above points to an array, it can be used to refer to specific array elements using standard array syntax: `b[i]`

Passing a Part of an Array

To pass a part of an array to a function, we simply pass the address of the first element of that part:

```
// Passing Part of an Array
// pass_part.c

#include <stdio.h>

void display(int *a, int n);

int main(void)
{
    int sku[] = { 2156, 4633, 3122, 5611};
    const int n = 4;

    display(&sku[1], n - 1);

    return 0;
}

void display(int *a, int n)
{
    int i;

    for (i = 0; i < n; i++)
    {
        printf("%5d\n", a[i]);
    }

    printf("\n");
}
```

The above program produces the following output:

```
4633
3122
5611
```

> ⚠ **CAUTION**
>
> The syntactic equivalence between pointer parameters and array names does not extend to the definition of an array. We cannot replace the definition of an array with a pointer definition. An array definition allocates the stated number of memory locations for all of the elements in the array. A pointer definition **allocates only one memory location to hold a single address**.

Pointer Arithmetic (Optional)

Learning pointer arithmetic clarifies this equivalence between pointers an arrays. We can obtain the address of an array element by multiplying the element's index by the number of bytes that each element occupies and add that product to the array's starting address. Then, we can access the data at the resulting address simply by dereferencing that address.

For example, the two examples below are equivalent:

Sample-1		Sample-2
`a[i]`	is equivalent to	`*(a + i)`
`&a[i]`	is equivalent to	`(a + i)`

`a + i` evaluates to the address of the `i+1-th` element of `a` (`&a[i]`). The rules for pointer arithmetic stipulate that we multiply the element's index by the size of an element before adding the array's starting address.

Examples

We may replace array subscript notation with equivalent pointer notation:

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

x = *a;      // stores 1 in x (the value of the first element)
x = *(a + 1) // stores 2 in x (the value of the second element)
x = *(a + 2) // stores 3 in x (the value of the third element)
```

We can also use pointer addition to move from one element to the neighbouring element:

```
int a[3];
int *p = a;

*p = 1;      // stores 1 in a[0]
p++;         // increment the pointer to point to the next element in memory
*p = 2;      // stores 2 in a[1]
p++;         // increment the pointer to point to the next element in memory
*p = 3;      // stores 3 in a[2]
```

> ⚠ **CAUTION**
>
> Although a pointer can be incremented, **an array name cannot**. This is because a pointer is a variable, while an array name is not a variable; that is, not a region of memory distinct from the array.

Array of Structures

We can define an array of objects of derived type just like we define an array of variables of primitive type. We suffix the object identifier with a brackets-enclosed integer specifying the number of elements in the array. This number is an integer constant or integer-constant expression.

For example, to define an array of 40 `Student`s, we write:

```
struct Student s[40];
```

The subscripting rules for arrays apply equally to primitive types and structure types:

- ***element indexing*** is 0-based - the first element is `s[0]`
- `s[i]` refers to the `i+1-th` element of the array
- the ***name of the array alone*** refers to the **address** of its first element - the address `s` is the **same** as the address `&s[0]`

Tabular Data

A structure type provides a convenient way of storing tabular data. Each instance of the structure holds the information for one row in the table. Each member of each instance holds one field of data in the row or record. The array of objects holds the entire table.

Consider the following structure:

```
struct Student
{
    int id;         // student ID
    float grade[4]; // grades
    char name[31];  // student name
};

int main(void)
{
    // ...

    struct Student s[40]; // table of 40 Student objects

    // ...
    return 0;
}
```

The student ID and name of the ***third student*** would be accessed by `s[2].id` and `s[2].name` respectively.

To initialize the members of the first three elements of the table, we write:

```
struct Student s[40] = { {10001, 78.9f, 56.7f, 0f, 0f, "Harry"},
                         {10002, 67.8f, 92.1f, 74.3f, 81.2f, "Jack"},
                         {10003, 55.4f, 66.5f, 88.3f, 34.6f, "Chris"}   };
```

We arrange the initial values member by member for each object in turn. The interior braces distinguish the values for one object from those for another object. The interior braces are optional: we may list the values in the order in which they are stored in memory:

```
struct Student s[40] = { 10001, 78.9f, 56.7f, 0f, 0f, "Harry",
                         10002, 67.8f, 92.1f, 74.3f, 81.2f, "Jack",
                         10003, 55.4f, 66.5f, 88.3f, 34.6f, "Chris" };
```

Composition

A structure type may contain a member that is of another structure type. The relationship between the types is called a ***composition*** relationship.

Consider a section of a course that contains a list of enrolled students:

```
struct Student
{
    int id;         // student ID
    float grade[4]; // grades
    char name[31];  // student name
};

struct Section
{
    int studentCount;
    struct Student students[40];
};
```

> **IMPORTANT**
>
> A structure *type* may **NOT** contain an object of its **own type** (see below example).

```
// A derrived type is not allowed to have its own type as a member!
struct Section
{
    int studentCount;
    struct Section section;  // <==== !!!! ERROR !!!!
};
```

Member Access

Dot, arrow, and subscript syntax extends to the members of structure types that are themselves structure types.

For example, let us define a `Section` object:

```
struct Section abc123a;
```

To set the number of students in `abc123a` to `23`, we write:

```
abc123a.studentCount = 23;
```

To set the student ID of the sixth student to `123456789`, we write:

```
abc123a.students[5].id = 123456789;
```

To set the third grade of the sixth student to a `67.8`, we write:

```
abc123a.students[5].grade[2] = 67.8f;
```

We say that *Section* `abc123a` has a *Student* with student ID `123456789`, whose third grade has been set to `67.8`.

Variable-Length Arrays

The size of an array can be defined at run-time; that is, the memory for the array can be allocated once its size is known. Its size is a program variable. We define the array only after its size is available. The memory for the array is deallocated once we leave the block within which it has been allocated.

For example:

```c
// Variable-Length Array
// var_len_array.c

#include <stdio.h>

void display(int g[], int n);

int main(void)
{
    int i, n;

    printf("Enter no of grades : ");
    scanf("%d", &n);
    int grade[n];    // Set array to size: n

    for (i = 0; i < n; i++)
    {
        printf("Grade %d : ", i + 1);
        scanf("%d", &grade[i]);
    }

    display(grade, n);

    return 0;
}

void display(int g[], int n)
{
    int i;

    for (i = 0; i < n; i++)
    {
        printf("%d ", g[i]);
    }

    printf("\n");
}
```

The above program produces the following output:

```
Enter no of grades : 3
Grade 1 : 9
Grade 2 : 10
Grade 3 : 7
9 10 7
```

The array is stored in the ***stack*** segment of RAM **alongside the local variables**.

> **IMPORTANT**
>
> **!!! PORTABILITY ALERT !!!**
> The Linux `gcc` compiler accepts variable length arrays, while the Windows `cl` compiler ***does not*** (the Windows compiler implements remenants of the older 1990 standard, which does not include variable-length arrays. For more on the language standard, see the chapter entitled Portability).

On this page

Two-Dimensional Arrays

Learning Outcomes

After reading this section, you will be able to:

- Connect procedures using `pass-by-value` and `pass-by-address` semantics to build a complete program
- Design data collections using arrays and structures to manage information efficiently

Introduction

A simple data structure for organizing tabular data is the two-dimensional array. The C language supports multi-dimensional arrays. The C compiler treats a two-dimensional array as an array of arrays. An obvious application of this data structure is an array of character strings.

This chapter introduces two-dimensional arrays, describing their `syntax` and their `organization` in memory. This chapter includes sample code snippets related to arrays of character strings.

Two-Dimensional Syntax

A table is a useful analogy for describing a two-dimensional array. An entry in a table is identified by its row and column positions. Consider the row and column indices in the figure below. `The first index` refers to the `row` and `the second index` refers to the `column`.

To **identify an element** of a two-dimensional array we use `two pairs of brackets`. The index within the `left pair` identifies the `row`, while the index within the `right pair` identifies the `column`:

```
array[ row ][ column ]
```

Indexing is **0-based** for both `rows` and `columns`.

Definition

The definition of a two-dimensional array takes the form:

```
type identifier[ r ][ c ] = init;
```

- `r` is the number of **rows** in the array
- `c` is the number of **columns** in the array
  - `r` and `c` are integer constants or constant integer expressions
- `init` is a braces-enclosed, comma-separated list of initial values.

The `total number of elements` in the array is determined by multiplying the row size by the column size: `r * c`. The assigment operator (`=`) together with `init` are optional.

If we add an initialization list, we may optionally omit the value of `r`. If `r * c` exceeds the number of initial values, the compiler initializes the remaining elements to `0`. If we omit the initialization list, we must specify `r`. We must always specify `c`.

For example:

```
int a[4][5] = {11, 12, 13, 14, 15,
               21, 22, 23, 24, 25,
               31, 32, 33, 34, 35,
               41, 42, 43, 44, 45};
```

To improve clarity, we may enclose each subset of initial values for each row in additional braces:

```
int a[4][5] = {{11, 12, 13, 14, 15},
               {21, 22, 23, 24, 25},
               {31, 32, 33, 34, 35},
               {41, 42, 43, 44, 45}};
```

Order

The C language stores the elements of a two-dimensional array in `row-major` order: the first row, **column-element by column-element**, then the second row, column-element by column-element, then the third row, etc..

For example, the elements of the array:

```
int a[4][5];
```

are stored as follows:

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

> ⚠ **WARNING:** Some programming languages store two-dimensional arrays in column-major order!

Passing to a Function

We pass a two-dimensional array to a function in the same way that we pass a one-dimensional array. We specify the name of the array as an argument in the function call. The corresponding function parameter receives the value of this argument as the address of the array. The parameter declaration identifies the array as two-dimensional by two pairs of brackets. The parameter declaration includes the array's column dimension.(the column dimension must be included)

For Example:

```
// Two-Dimensional Arrays
// pass2DArray.c

#include <stdio.h>

#define NCOLS 3

void display(int data[][NCOLS], int rows, int cols);

int main(void)
{
    int a[2][NCOLS] = {{11, 12, 13}, {21, 22, 23}};

    display(a, 2, 3);
}

void display(int data[][NCOLS], int rows, int cols)
{
    int i, j;
```

```
    for (i = 0; i < rows; i++)
    {
        for (j = 0; j < cols; j++)
        {
            printf("%d ", data[i][j]);
        }
        printf("\n");
    }
}
```

The above program produces the following output:

```
11 12 13
21 22 23
```

The compiler needs the column dimension (`NCOLS`) to determine the start of each `row` within the array. As with one-dimensional arrays the first dimension does not need to be included.

Passing a Specific Row of an Array

A two-dimensional C array is an array of one-dimensional arrays:

```
a[0][0] a[0][1] a[0][2] a[0][3] a[0][4] first row
a[1][0] a[1][1] a[1][2] a[1][3] a[1][4] second row
a[2][0] a[2][1] a[2][2] a[2][3] a[2][4] third row
a[3][0] a[3][1] a[3][2] a[3][3] a[3][4] fourth row
```

A reference to an entire row of a two-dimensional array takes the form of the name of the array followed by the row number within brackets:

```
array[ row ]
```

To pass a specific row to a function, we identify the row as part of the argument in the function call.

For example:

```
// Two-Dimensional Arrays
// passRow.c

#include <stdio.h>

#define NCOLS 3

void displayRow(int data[], int cols);

int main(void)
{
    int a[2][NCOLS] = {{11, 12, 13}, {21, 22, 23}};

    displayRow (a[0], NCOLS); // pass first row
    displayRow (a[1], NCOLS); // pass second row
}

void displayRow(int data[], int cols)
{
    int i;

    for (i = 0; i < cols; i++)
    {
        printf("%d ", data[i]);
```

```
    }
    printf("\n");
}
```

The above program produces the following output:

```
11 12 13
21 22 23
```

`a[0]` points to the **first row** of `array a` and holds the address of the **first element** of that row.

`a[1]` points to the **second row** of `array a` and holds the address of the **first element** of that row.

Arrays of Character Strings

An array of C-strings is a two-dimensional array. The `row` index refers to `a particular character string`, while the `column` index refers to `a particular character within a character string`.

Definition

The definition of an array of character strings takes the form:

```
char identifier[NO_OF_STRINGS][MAX_NO_OF_BYTES_PER_STRING];
```

To declare an array of `5` **names** with each name holding up to `30` **characters**, we write:

```
char name[5][31];
```

> **Note:** The number of names (`[5]`) precedes the maximum number of characters in a name (`[31]`).

Initialization

Initialization of an array of character strings takes the form:

```
char identifier[NO_OF_STRINGS][MAX_NO_OF_BYTES_PER_STRING] = {
                    initializer_1, initializer_2, ... };
```

For Example:

```
char name[5][31] = {"Harry", "Jean", "Jessica", "Irene", "Jim"};
```

A String within an Array of Strings

To refer to a string within an array of strings, we follow the array identifier with `a single pair of brackets`. The index within the pair of brackets identifies the string within the array.

The address of a string in an array of strings takes the form:

```
identifer[index]
```

For Example:

```
name[1]
```

The above example references the second string in `name`.

Address of a Character

To refer to a character within a string, we follow the array identifier with `two pairs of brackets`, the first containing the index that identifies the string and the second containing the index that identifies the character within that string.

For Example:

```
name[1][2]
```

The above example references the third character within the second string of `name`.

```
&name[1][2]
```

The above example references the address of the third character within the second string of `name`.

Input and Output

- **Input**

To accept input for a list of 5 names, we write:

```
int i;
char name[5][31];

for (i = 0; i < 5; i++){
    scanf(" %[^\n]", name[i]);
}
```

> **Reminder:** The space in the format string skips leading whitespace before accepting the string.

- **Output**

To display the third string in name, we write:

```
char name[5][31] = {"Harry", "Jean", "Jessica", "Irene", "Jim"};

printf("%s", name[2]);
```

Functions

- **Arguments**

To pass a list of names to a function, we write:

```
char names[5][31] = {"name1", "name2", "name3", "name4", "name5"};

display(names, 5);
```

- **Parameters**

To receive the address of this array of strings in a function parameter, we write:

```
```

```c
void display(char names[][31], int count)
{
    int i;

    for(i = 0; i < count; i++)
    {
        printf("%s\n", names[i]);
    }
}
```

On this page

Algorithms

Learning Outcomes

After reading this section, you will be able to:

- Design procedures using selection and iteration constructs to solve a programming task
- Design data collections using arrays to manage information efficiently

Introduction

The central aspect of solving a programming task is the design of an appropriate algorithm. An algorithm is the set of rules that define the sequence of operations required to complete the task. Examples of tasks that require algorithms include finding the elements in a list satisfying a specified condition, sorting the elements of a list in a specified order and mixing the elements of a list. The design of an algorithm typically involves selections and iterations; in some cases, nested selections and nested iterations. Often, there is more than one algorithm that solves the programming task. Different algorithms exhibit different efficiencies.

This chapter introduces the implementations of a few more common algorithms. We implement them in function form for use as black boxes in other applications.

Searching

Search algorithms find the index of one or more array elements that satisfy a specified condition or set of conditions. These algorithms work with key-value pairs. Each key is unique while the values are not necessarily unique.

Two Algorithms

Unsorted Key Array

Given an unsorted key array, we start our search at the first element and progress through the array element by element until we find a match. This algorithm involves an iteration and a selection:

```c
// Find Unit Price
// find.c

#include <stdio.h>

// find returns the index of the first element in skuData[skuCount]
// that contains the value 'findSKU'; -1 if not found
//
int find(int skuData[], int skuCount, int findSKU)
{
    int i, skuIndex = -1;

    for (i = 0; skuIndex < 0 && i < skuCount; i++)
    {
        if (findSKU == skuData[i])
        {
            skuIndex = i; // save the index
        }
    }

    return skuIndex;
}
```

```c
int main(void)
{
    int i, searchSKU;
    int sku[]      = { 2156, 4633, 3122, 5611};
    double price[] = {12.34, 7.89, 6.56, 9.32};
    const int itemCount    = 4;

    printf("SKU   : ");
    scanf("%d", &searchSKU);

    i = find(sku, itemCount, searchSKU);

    if (i >= 0 && i < itemCount)
    {
        printf("Price : $%0.2lf\n", price[i]);
    }
    else
    {
        printf("%d not in system\n", searchSKU);
    }

    return 0;
}
```

The above program produces the following output:

```
SKU   : 4633
Price : $7.89
```

> **NOTE**
>
> The value returned by `find()` (variable `i`) is validated to ensure that it is within the bounds of the key array (that is, we check that it is not -1 and not more than the number of items in the array).

Sorted Key Array (Optional)

Given a sorted key array, we start our search in the middle of the array and at each step discard the half that doesn't contain the search key. Although this algorithm is slightly more complicated than the unsorted one, it is significantly faster, which is important with a large number of elements.

```c
// Find Unit Price - Sorted Keys Ascending Order
// find_ascend.c

#include <stdio.h>

// find_ascend returns the index of the first element
// in ascending order skuData[n] that contains the value 'findSKU'
// returns -1 if not found
//
int find_ascend(int skuData[], int skuCount, int findSKU)
{
    int i, low = 0, high = skuCount - 1, skuIndex = -1;

    do {
        i = (low + high) / 2;

        if (skuData[i] < findSKU)
        {
            low = i + 1;
        }
        else if (skuData[i] > findSKU)
        {
            high = i - 1;
```

```
        }
        else
        {
            skuIndex = i;
        }
    } while (low <= high && skuIndex == -1);

    return skuIndex;
}

int main(void)
{
    int i, searchSKU;
    int sku[] =      { 2156,  3122, 4633, 5611 };
    double price[] = { 12.34, 6.56, 7.89, 9.32 };
    const int itemCount = 4;

    printf("SKU   : ");
    scanf("%d", &searchSKU);

    i = find_ascend(sku, itemCount, searchSKU);

    if (i >= 0 && i < itemCount)
    {
        printf("Price : $%0.2lf\n", price[i]);
    }
    else
    {
        printf("%d not in system\n", searchSKU);
    }

    return 0;
}
```

```
SKU   : 4633
Price : $7.89
```

> **NOTE**
>
> The above example using a divide and conquer approach is referred to as a ***binary search***.

Masking

Masking algorithms distinguish certain array elements from all other elements. The masking array is a parallel array with respect to the other arrays in the set. The elements of the masking array are flags that identify inclusion or exclusion.

Consider the following program, which calculates the total purchase price for a set of products. The user enters the SKU (Stock-Keeping Unit) for each product purchased and the quantity. Some products attract HST (Harmonized Sales Tax), while others do not. We store the SKUs, unit prices and tax status in three parallel arrays. The tax status array is the masking array. The user enters 0 to terminate input.

```
// Total Purchase Price
// masking.c

#include <stdio.h>

#define HST 0.13

int find(int skuData[], int skuCount, int findSKU)
{
    int i, skuIndex = -1;
    for (i = 0; skuIndex < 0 && i < skuCount; i++)
```

```
    {
        if (findSKU == skuData[i])
        {
            skuIndex = i;
        }
    }

    return skuIndex;
}

int main(void)
{
    int i, searchSKU, quantity;
    int sku[] =      { 2156,  3122, 4633, 5611 };
    double price[] = { 12.34, 6.56, 7.89, 9.32 };
    int tax[] =      {   1,    0,    0,    1 };
    const int itemCount = 4;
    double sum = 0.0;

    do {
        printf("SKU     : ");
        scanf("%d", &searchSKU);

        if (searchSKU != 0)
        {
            i = find(sku, itemCount, searchSKU);

            if (i >= 0 && i < itemCount)
            {
                printf("Quantity: ");
                scanf("%d", &quantity);

                if (tax[i] == 1)
                {
                    sum += quantity * price[i] * (1.0 + HST);
                }
                else
                {
                    sum += quantity * price[i];
                }
            }
            else
            {
                printf("%d not in system\n", searchSKU);
            }
        }
    } while (searchSKU != 0);

    printf("Total is $%0.2lf\n", sum);

    return 0;
}
```

The below output is a sample execution of the above program:

```
SKU     : 2156
Quantity: 3
SKU     : 3121
3121 not in system
SKU     : 3122
Quantity: 2
SKU     : 5611
Quantity: 1
SKU     : 0
Total is $65.48
```

Sorting

Sorting algorithms rearrange the elements of an array according to a pre-defined rule. Typically, this rule is ascending or descending order. The sorting criterion may be numeric or based upon a collating sequence such as ASCII or EBCDIC.

The two simplest algorithms are:

- selection sort
- bubble sort

Selection Sort

A selection sort selects a reference element and steps through the rest of the elements looking for any one with a value that does not meet the test condition. If found, the algorithm swaps that element with the reference element.

The following program sorts the array in ascending order. Starting with the first element in the array, it picks the first unsorted element as the reference element, swaps it with the smallest element in the unsorted part of the array, and iterates until it reaches the last element in the array.

```c
// Selection Sort
// selectionSort.c

#include <stdio.h>

// selectSort sorts the elements of data[itemCount] in ascending order
//
void selectSort(int data[], int itemCount)
{
    int i, j, minIdx;
    int temp;

    for (i = 0; i < itemCount; i++)
    {
        minIdx = i;

        for (j = i + 1; j < itemCount; j++)
        {
            if (data[j] < data[minIdx])
            {
                minIdx = j;
            }
        }

        if (minIdx != i)
        {
            temp = data[i];
            data[i] = data[minIdx];
            data[minIdx] = temp;
        }
    }
}

int main(void)
{
    int i;
    int sku[] = { 2156, 4633, 3122, 5611 };
    const int skuCount = 4;

    selectSort(sku, skuCount);

    for (i = 0; i < skuCount; i++)
    {
        printf("%6d\n", sku[i]);
    }
```

```
    return 0;
}
```

The above program produces the following output:

```
  2156
  3122
  4633
  5611
```

Bubble Sort

A bubble sort moves through the array element by element swapping elements if the next one does not satisfy the sort condition. The algorithm repeats this process for each unsorted subset of the array starting with the first element. The algorithm moves elements to their terminal positions just like bubbles rising through a liquid - hence the name bubble sort.

```
// Bubble Sort
// bubbleSort.c

#include <stdio.h>

// bubbleSort sorts the elements of a[skuCount] in ascending order
//
void bubbleSort(int data[], int itemCount)
{
    int i, j;
    int temp;

    for (i = itemCount - 1; i > 0; i--)
    {
        for (j = 0; j < i; j++)
        {
            if (data[j] > data[j + 1])
            {
                temp = data[j];
                data[j] = data[j + 1];
                data[j + 1] = temp;
            }
        }
    }
}

int main(void)
{
    int i;
    int sku[] = { 2156, 4633, 3122, 5611 };
    const int skuCount = 4;

    bubbleSort(sku, skuCount);

    for (i = 0; i < skuCount; i++)
    {
        printf("%6d\n", sku[i]);
    }

    return 0;
}
```

The above program produces the following output:

```
  2156
  3122
```

```
4633
5611
```

Sorting Strings (Optional)

The following program accepts a list of names, sorts them in alphabetic order and displays the sorted list:

```c
// Sort a List of Names
// sortNames.c

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

#define MAX_NAMES 10      // maximum number of names in the list
#define MAX_NAME_LEN 30   // maximum number of characters in a name
#define FMT_NAME_LEN "30" // used in the format string of scanf()

void bubble(char names[][MAX_NAME_LEN + 1], int size);

int main(void)
{
    char name[MAX_NAMES][MAX_NAME_LEN + 1];
    int i, nameCount, keepgoing;

    // Input the list of names
    printf("Enter names (^ to stop)\n");
    i = 0;

    do {
        printf("Name-%d: ", i + 1);
        scanf(" %"FMT_NAME_LEN"[^\n]", name[i]);

        keepgoing = strcmp(name[i], "^") != 0;
        i++;
    } while (keepgoing == 1 && i < MAX_NAMES);

    if (keepgoing == 1)
    {
        nameCount = MAX_NAMES;
    }
    else
    {
        nameCount = i - 1;
    }

    // sort the names
    bubble(name, nameCount);

    // display thesorted list
    for (i = 0; i < nameCount; i++)
    {
        printf("%s\n", name[i]);
    }

    return 0;
}

// bubbleSort sorts the elements of names[size] in ascending order
//
void bubble(char names[][MAX_NAME_LEN + 1], int size)
{
    int i, j;
    char temp[MAX_NAME_LEN + 1];

    for (i = size - 1; i > 0; i--)
```

```
    {
        for (j = 0; j < i; j++)
        {
            if (strcmp(names[j], names[j + 1]) > 0)
            {
                strcpy(temp, names[j]);
                strcpy(names[j], names[j + 1]);
                strcpy(names[j + 1], temp);
            }
        }
    }
}
```

The below output is a sample execution of the above program:

```
Enter names (^ to stop)
Name-1: Timmothy
Name-2: Helen
Name-3: Demetri
Name-4: Zamphire
Name-5: Ariana
Name-6: ^
Ariana
Demetri
Helen
Timmothy
Zamphire
```

> **NOTE**
>
> Notice the concatenation of string literals in the format string of the call to `scanf()`. This lets us set the maximum number of input characters alongside the maximum number of characters in the array of strings at the head of the program code. The C compiler converts a concatenation of string literals into a single literal removing the intermediate pairs of double quotes; that is, to the compiler `"a""b""c"` is the same as `"abc"`.

Mixing (Optional)

Mixing algorithms have applications in games of chance. Examples include shuffling the cards in a deck or tumbling numbered balls into a lottery chute. The algorithm depends on the extent to which we seek to generate a truly fair result.

Consider the following program, which tumbles 10 balls into a lottery chute. To simulate mixing, the algorithm picks the index of a reference element, randomly picks the index of another element further along in the array and swaps the values stored in the two elements. This algorithm is attributed to Donald Knuth, a pioneer of computer science.

```
// Mix Lottery Balls
// mix.c

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define BALL_COUNT 10

// mix mixes the elements in data[arraySize] randomly
//
void shuffle(int data[], int arraySize)
{
    int i, remainingItems, j, temp;

    remainingItems = arraySize;
```

```
    for (i = 0; i < arraySize; i++)
    {
        j = i + rand() % remainingItems;

        temp = data[i];
        data[i] = data[j];
        data[j] = temp;

        remainingItems--;
    }
}

int main(void)
{
    int i;
    int ball[BALL_COUNT];

    for (i = 0; i < BALL_COUNT; i++)
    {
        ball[i] = i + 1;
    }

    srand(time(NULL));
    shuffle(ball, BALL_COUNT);

    for (i = 0; i < BALL_COUNT; i++)
    {
        printf("%2d\n", ball[i]);
    }

    return 0;
}
```

The below output is a sample execution of the above program:

```
 2
 7
 3
 8
 6
 9
 5
10
 1
 4
```

On this page

Portability

Learning Outcomes

After reading this section, you will be able to:

- Explain procedural programming using nontechnical terminology to inform a business person

Introduction

Mature languages attract international standards. These standards promote uniformity across host platforms. Committees of compiler writers, academics, application developers and representatives from various national organizations review, discuss, negotiate and reach consensus on which elements should be included in the standard definition. Elements that are excluded are called extensions. Compiler writers include their *extensions* in their own implementations of the agreed standard.

A standard definition enables developers to write programs with the expectation that they will compile and run on a wide variety of platforms that support the standard. By avoiding the use of extensions, we increase the portability of our software. We call standard-compliant software *highly* portable.

This chapter reviews the C standards that have been defined throughout the language's history, introduces some coding guidelines, and describes utilities available for analyzing the degree of portability.

The C Standards

Dennis Ritchie created the C language between 1969 and 1973, while working at AT&T Bell Labs. Brian Kernighan and Dennis Ritchie published their original description of the language in the classic book entitled "The C Programming Language" in 1978. They published a second edition in 1988.

Since then, the C language has undergone three 'standard' definitions: one in 1989, one in 1999 and the latest in 2011.

C89

In 1982, the American National Standards Institute formed committee X3J11 to propose the first standard for C. An international group converted this standard with minor modifications into the ISO/IEC 9899:1990 standard, which ANSI subsequently adopted. This was called Standard C to distinguish it from Kernighan and Ritchie C (K & R C, in short).

Standard C introduced:

- Function prototypes
- A standard library
- Whitespace before `#`
- New keywords

In 1995, ANSI introduced some amendments to Standard C, called C95, which included new conversion specifiers for `printf()` and `scanf()`, and more library functions.

C99

ISO/IEC approved the second international standard in 1999: ISO/IEC 9899:1999 (C99).

C99 introduced:

- `long long` data type
- `_Bool` data type
- `_Complex` data type
- Complex arithmetic
- Variable-length arrays
- `//` C++ style comments
- Better support for floating-point types, including math functions for all types

C11

ISO/IEC approved the third international standard in 2011: ISO/IEC 9899:2011 (C11).

C11 introduced:

- Multi-threading support
- Improved Unicode support
- Bounds-checking interfaces
- A create and open mode **("x")** for `fopen()`
- Removal of `gets()`
- Optional support for complex arithmetic
- Optional support for variable-length arrays

The optional support applies to features that had been mandatory under the C99 standard.

Structured Programs

Software engineers have advocated certain paradigms for developing software. The *Structured Programming* paradigm achieves clarity by distinguishing high-level coding techniques from low-level assembly language coding techniques. The principles of this paradigm apply to coding in many languages, including C. They may be summarized as follows:

- A structured program consists solely of three constructs: sequences, selections, and iterations
- Each structured construct exhibits a single entry point and a single exit point
- Statements that jump outside a construct are not allowed with certain exceptions (`switch case` constructs in the C language)

The following style points highlight structured properties of program code:

- Indent and align code to reveal the structure of the logic
- Indent the code consistently within a structured construct
- Separate constructs by blank lines to improve clarity

Static Analysis (Optional)

Static analysis of a program can reveal code that might not be portable. Tools for static analysis flag suspicious code. Some popular tools for C language programs include:

- `lint` - the original analyzer written by Stephen Johnson in 1979
- Parasoft `C/C++test` - plugins for Visual Studio and Eclipse IDEs
- `Splint` - a modern open source version of `Lint`

Lint Checking

Consider the following program:

```c
// Lint Checking
// lint.c

#include <stdio.h>

int main(void)
{
    char c;
    c = getchar();

    if (c == EOF)
    {
        printf("End of data\n");
    }
    else
    {
        printf("You entered %c\n", c);
    }

    return 0;
}
```

Under normal settings, the `gcc` and `cl` compilers will not issue any warning or error messages. If this program compiles under either of these compilers and the user enters the "*end of data*" key-combination, the output will be:

```
End of data
```

If this program compiles under an **AIX** compiler and the user enters "*end of data*" key-combination, the output will be:

```
You entered ÿ
```

AIX compilers store `char` with a range of 0 to 255, while `gcc` and `cl` store `char` with a range -127 to 127. To resolve the ambiguity, we declare `c` as `int`.

```c
// Lint Checking
// lint.c

#include <stdio.h>

int main(void)
{
    int c;  // store character as an int
    c = getchar();

    if (c == EOF)
    {
        printf("End of data\n");
    }
    else
    {
        printf("You entered %c\n", c);
    }

    return 0;
}
```

To identify this portability issue on an **AIX** platform, we can pass the source code through the Lint code checker:

```
lint lint.c
```

Example message from `Lint`:

```
"lint.c", line 13: warning: comparison of unsigned with negative constant
```

To identify this portability issue on a `cl` platform, we change the compiler warning level options to `/W4` (warning level 4) and `/WX` (treat all warnings as errors).

> **NOTE**
>
> You can set these attributes in the Visual Studio project properties

```
cl lint.c /W4 /WX
```

Example message from `Lint`:

```
lint.c(11) : error C2220: warning treated as error - no object file generated
lint.c(11) : warning C4244: '=' : conversion from 'int' to 'char', possible loss of data
```

Documentation

You can find documentation on lint code checkers at

- `Lint`
- `LCLint/Splint`

Guidelines (Optional)

Many organizations establish their own guidelines for maintaining consistency and continuity throughout the life cycle of the applications written by several developers. Guidelines differ from standards in the scope of their applicability: they are less prescriptive than standards.

Examples of institutional guidelines include:

- GCC Coding Conventions - guidelines for C code in the GNU project
- Debian - guidelines for inclusion in the Debian project
- Carnegie Mellon University - Electrical and Computer Engineering
- MISRA C 2012 - guideline for use of the C language in critical systems

Examples of common features in these guidelines include:

- `#include`s should follow the header comments
- `#define`s should follow the `#include`s
- `#define` constants should be in all **CAPS**
- Avoid names that differ in case only
- Avoid names that look like each other
- Constants should be of the same data type as the variables that the constants initialize
- Avoid declarations that shadow declarations at a higher level
- Avoid assuming the ASCII collating sequence
- Split compound conditions into one simple condition per line
- Avoid defaulting a test condition to non-zero
- Compare to **FALSE** rather than **TRUE**
- Comment each fall through in a `switch case:`
- Attach unary operators to their operand without spaces
- Compare a pointer to `NULL` instead of to **0**

- Avoid assumptions about parameter passing and the order of their evaluation
- Avoid possible side effects such as `a[i] = b[i++];`

ASCII Collating Sequence

The character encoding sequence compatible with Unicode

Binary	Hex.	Dec.	Symbol		Binary	Hex.	Dec.	Symbol		Binary	Hex.	Dec.	Symbol		Binary	Hex.	I
00000000	00	0	NUL		00100000	20	32	SP		01000000	40	64	@		01100000	60	
00000001	01	1	SOH		00100001	21	33	!		01000001	41	65	A		01100001	61	
00000010	02	2	STX		00100010	22	34	"		01000010	42	66	B		01100010	62	
00000011	03	3	ETX		00100011	23	35	#		01000011	43	67	C		01100011	63	
00000100	04	4	EOT		00100100	24	36	$		01000100	44	68	D		01100100	64	
00000101	05	5	ENQ		00100101	25	37	%		01000101	45	69	E		01100101	65	
00000110	06	6	ACK		00100110	26	38	&		01000110	46	70	F		01100110	66	
00000111	07	7	BEL		00100111	27	39	'		01000111	47	71	G		01100111	67	
00001000	08	8	BS		00101000	28	40	(		01001000	48	72	H		01101000	68	
00001001	09	9	HT		00101001	29	41	)		01001001	49	73	I		01101001	69	
00001010	0A	10	LF		00101010	2A	42	*		01001010	4A	74	J		01101010	6A	
00001011	0B	11	VT		00101011	2B	43	+		01001011	4B	75	K		01101011	6B	
00001100	0C	12	FF		00101100	2C	44	,		01001100	4C	76	L		01101100	6C	
00001101	0D	13	CR		00101101	2D	45	-		01001101	4D	77	M		01101101	6D	
00001110	0E	14	SO		00101110	2E	46	.		01001110	4E	78	N		01101110	6E	
00001111	0F	15	SI		00101111	2F	47	/		01001111	4F	79	O		01101111	6F	
00010000	10	16	DLE		00110000	30	48	0		01010000	50	80	P		01110000	70	
00010001	11	17	DC1		00110001	31	49	1		01010001	51	81	Q		01110001	71	
00010010	12	18	DC2		00110010	32	50	2		01010010	52	82	R		01110010	72	
00010011	13	19	DC3		00110011	33	51	3		01010011	53	83	S		01110011	73	
00010100	14	20	DC4		00110100	34	52	4		01010100	54	84	T		01110100	74	
00010101	15	21	NAK		00110101	35	53	5		01010101	55	85	U		01110101	75	
00010110	16	22	SYN		00110110	36	54	6		01010110	56	86	V		01110110	76	
00010111	17	23	ETB		00110111	37	55	7		01010111	57	87	W		01110111	77	
00011000	18	24	CAN		00111000	38	56	8		01011000	58	88	X		01111000	78	

Binary	Hex.	Dec.	Symbol		Binary	Hex.	Dec.	Symbol		Binary	Hex.	Dec.	Symbol		Binary	Hex.	I
00011001	19	25	EM		00111001	39	57	9		01011001	59	89	Y		01111001	79	
00011010	1A	26	SUB		00111010	3A	58	:		01011010	5A	90	Z		01111010	7A	
00011011	1B	27	ESC		00111011	3B	59	;		01011011	5B	91	[		01111011	7B	
00011100	1C	28	FS		00111100	3C	60	<		01011100	5C	92	\		01111100	7C	
00011101	1D	29	GS		00111101	3D	61	=		01011101	5D	93	]		01111101	7D	
00011110	1E	30	RS		00111110	3E	62	>		01011110	5E	94	^		01111110	7E	
00011111	1F	31	US		00111111	3F	63	?		01011111	5F	95	_		01111111	7F	

EBCDIC Collating Sequence

Dec	Hex	Character		Dec	Hex	Character		Dec	Hex	Character
0	00	NUL		**86**	56			**171**	AB	
1	01	SOH		**87**	57			**172**	AC	
2	02	STX		**88**	58			**173**	AD	
3	03	ETX		**89**	59			**174**	AE	
4	04	PF		**90**	5A	!		**175**	AF	
5	05	HT(tab)		**91**	5B	$		**176**	B0	
6	06	LC		**92**	5C	*		**177**	B1	
7	07	DEL		**93**	5D	)		**178**	B2	
8	08			**94**	5E	;		**179**	B3	
9	09			**95**	5E	| OR NOT		**180**	B4	
10	0A	SMM		**96**	60	-		**181**	B5	
11	0B	VT		**97**	61	/		**182**	B6	
12	0C	FF(formfeed)		**98**	62			**183**	B7	
13	0D	CR (car ret)		**99**	63			**184**	B8	
14	0E	S0		**100**	64			**185**	B9	' or
15	0F	SI		**101**	65			**186**	BA	
16	10	DLE		**102**	66			**187**	BB	
17	11	DC1		**103**	67			**188**	BC	
18	12	DC2		**104**	68			**189**	BD	
19	13	TM		**105**	69			**190**	BE	
20	14	RES		**106**	6A	| or		**191**	BF	
21	15	NL		**107**	6B	,		**192**	C0	
22	16	BS(backspace)		**108**	6C	%		**193**	C1	A
23	17	IL		**109**	6D	_		**194**	C2	B
24	18	CAN		**110**	6E	>		**195**	C3	C
25	19	EM		**111**	6F	?		**196**	C4	D

Dec	Hex	Character		Dec	Hex	Character		Dec	Hex	Character
26	1A	CC		**112**	70			**197**	C5	E
27	1B	CUI		**113**	71			**198**	C6	F
28	1C	IFS		**114**	72			**199**	C7	G
29	1D	IGS		**115**	73			**200**	C8	H
30	1E	IRS		**116**	74			**201**	C9	I
31	1F	IUS		**117**	75			**202**	CA	
32	20	DS		**118**	76			**203**	CB	
33	21	SOS		**119**	77			**204**	CC	
34	22	FS		**120**	78			**205**	CD	
35	23			**121**	79			**206**	CE	
36	24	BYP		**122**	7A	:		**207**	CF	
37	25	LF		**123**	7B	#		**208**	D0	
38	26	ETB		**124**	7C	@		**209**	D1	J
39	27	ESC(escape)		**125**	7D	'		**210**	D2	K
40	28			**126**	7E	=		**211**	D3	L
41	29			**127**	7F	"		**212**	D4	M
42	2A	SM		**128**	80			**213**	D5	N
43	2B	CU2		**129**	81	a		**214**	D6	O
44	2C			**130**	82	b		**215**	D7	P
45	2D	ENQ		**131**	83	c		**216**	D8	Q
46	2E	ACK		**132**	84	d		**217**	D9	R
47	2F	BEL		**133**	85	e		**218**	DA	
48	30			**134**	86	f		**219**	DB	
49	31			**135**	87	g		**220**	DC	
50	32	SYN		**136**	88	h		**221**	DD	
51	33			**137**	89	i		**222**	DE	
52	34	PN		**138**	8A			**223**	DF	
53	35	RS		**139**	8B			**224**	EO	

Dec	Hex	Character		Dec	Hex	Character		Dec	Hex	Character
54	36	UC		**140**	8C			**225**	E1	
55	37	EOT		**141**	8D			**226**	E2	S
56	38			**142**	8E			**227**	E3	T
57	39			**143**	8F			**228**	E4	U
58	3A			**144**	90			**229**	E5	V
59	3B	CU3		**145**	91	j		**230**	E6	W
60	3C	DC4		**146**	92	k		**231**	E7	X
61	3D	NAK		**147**	93	l		**232**	E8	Y
62	3E			**148**	94	m		**233**	E9	Z
63	3F	SUB		**149**	95	n		**234**	EA	
64	40	SP		**150**	96	o		**235**	EB	
65	41			**151**	97	p		**236**	EC	
66	42			**152**	98	q		**237**	ED	
67	43			**153**	99	r		**238**	EE	
68	44			**154**	9A			**239**	EF	
69	45			**155**	9B			**240**	F0	0
70	46			**156**	9C			**241**	F1	1
71	47			**157**	9D			**242**	F2	2
72	48			**158**	9E			**243**	F3	3
73	49			**159**	9F			**244**	F4	4
74	4A	c(cent)		**160**	A0			**245**	F5	5
75	4B	.		**161**	A1			**246**	F6	6
76	4C	<		**162**	A2			**247**	F7	7
77	4D	(		**163**	A3	t		**248**	F8	8
78	4E	+		**164**	A4	u		**249**	F9	9
79	4F	|, ! OR		**165**	A5	v		**250**	FA	
80	50	&		**166**	A6	w		**251**	FB	
81	51			**167**	A7	x		**252**	FC	

Dec	Hex	Character		Dec	Hex	Character		Dec	Hex	Character
82	52			**168**	A8	y		**253**	FD	
83	53			**169**	A9	z		**254**	FE	
84	54			**170**	AA			**255**	FF	

On this page

Data Conversions

Learning Outcomes

After reading this section, you will be able to:

- Convert between binary and hexadecimal notation
- Convert between binary and decimal notation

Introduction

A C program at machine-level is an assembly language program. Assembly language uses hexadecimal representation for data. The hardware itself processes information in bits. When a program outputs data in hexadecimal or binary form, we may prefer to convert it into decimal form.

This chapter describes how to convert across binary, hexadecimal and decimal representations and shows what a trivially simple program looks like in binary and hexadecimal representations.

Binary - Hexadecimal

The most convenient base for storing byte-wise information is hexadecimal (base 16). Two hexadecimal (base 16) digits can represent one byte of information. Each hexadecimal digit represents 4 bits of binary information.

For example, the hexadecimal value 0x5C is equivalent to the binary 010111002. The 0x prefix identifies the number as hexadecimal notation. The digits in the hexadecimal number system are {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. The characters A through F denote decimal values 10 through 15 respectively.

Binary to Hexadecimal

To convert a binary number to its hexadecimal equivalent, we:

1. group the bits into nibbles,
2. assign powers of 2 to the different bits in each nibble,
3. multiply each bit value by the corresponding power of 2,
4. add the products together for each nibble, and
5. concatenate the nibble results

Consider the 8-bit number $\mathbf{01011100_2}$:

Nibble #	1				0			
Bit #	7	6	5	4	3	2	1	0
Multiplier	8	4	2	1	8	4	2	1
Contents	0	1	0	1	1	1	0	0
Nibble Values	0*8 + 1*4 + 0*2 + 1*1 = 0x5				1*8 + 1*4 + 0*2 + 1*0 = 0xC			
Byte Value	0x5C							

Hexadecimal to Binary

To convert a hexadecimal number into its binary equivalent, we work from the lowest order bit to the highest. We identify the lowest order bit as the first target bit, then:

- divide by 2,
- put the remainder into the target bit,
- change the target to the next higher order bit

... and repeat the above until there are no more bits.

Consider the hexadecimal number **0x5C**:

- Identify the first target bit as bit 0
- Divide the number (0x5C) into left and right hexadecimal digits
- Take the right digit (0xC), divide it by 2 and put the remainder (0) in bit 0
- Take the result (0x6), divide it by 2 and put the remainder (0) in bit 1
- Take the result (0x3), divide it by 2 and put the remainder (1) in bit 2
- Take the result (0x1), divide it by 2 and put the remainder (1) in bit 3
- Take the left hexadecimal digit (0x5), divide it by 2 and put the remainder (1) in bit 4
- Take the result (0x2), divide it by 2 and put the remainder (0) in bit 5
- Take the result (0x1), divide it by 2 and put the remainder (1) in bit 6
- Take the result (0x0), divide it by 2 and put the remainder (0) in bit 7

Bit#	7	6	5	4	3	2	1	0
Byte Value	0x5C							
Nibble Values	0x5				0xC			
Divide by 2	0	0	1	2	0	1	3	6
Bit Values	0	1	0	1	1	1	0	0

Decimal - Binary

To convert a non-negative integer into its binary equivalent, we start with the value and an empty container that consists of target bits. We take the integer value, identify the lowest order bit in the container as our target bit, and then:

- divide the value by 2,
- store the remainder in the target bit,
- take the result as the new integer value,
- identify the next higher-order bit our new target bit, and
- repeat this set of instructions until no value is left

Consider the value **92**:

- Identify the target bit as bit numbered 0
- Take **92**, divide it by 2 and put the remainder (**0**) in bit 0
- Take the result (**46**), divide it by 2 and store the remainder (**0**) in bit 1
- Take the result (**23**), divide it by 2 and store the remainder (**1**) in bit 2
- Take the result (**11**), divide it by 2 and store the remainder (**1**) in bit 3
- Take the result (**5**), divide it by 2 and store the remainder (**1**) in bit 4
- Take the result (**2**), divide it by 2 and store the remainder (**0**) in bit 5
- Take the result (**1**), divide it by 2 and store the remainder (**1**) in bit 6

- Take the result (**0**), divide it by 2 and store the remainder (**0**) in bit 7

Bit#	7	6	5	4	3	2	1	0
Value	0	1	2	5	11	23	46	92
Bit Values	0	1	0	1	1	1	0	0

(Eight bits and right to left bit numbering are for brevity and illustrative purposes only.)

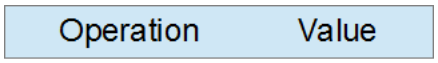
To convert a binary number into its decimal equivalent, we multiply the value in each bit by its corresponding power of 2 and add the products together.

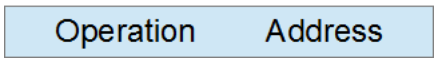
Consider the 8-bit binary number $\mathbf{01011100_2}$:

Bit #	7	6	5	4	3	2	1	0
Power of 2	7	6	5	4	3	2	1	0
Bit Values	0	1	0	1	1	1	0	0
Multiplier	128	64	32	16	8	4	2	1
Byte Value	0*128 + 1*64 + 0*32 + 1*16 + 1*8 + 1*4 + 0*2 + 0*1 = 92							

Program Instructions (optional)

A program instruction consists of an operation and possibly some operands. Each instruction performs an operation on its operands or on values stored in operand addresses. The addresses are either register names or addresses in primary memory.





The set of instructions in binary on a Windows 7 machine for a program that displays the phrase "This is C" looks like:

```
10110100 00001001
10111010 00001001 00000001
11001101 00100001
11001101 00100000
01010100
01101000
01101001
01110011
00100000
01101001
01110011
00100000
01000011
00100100
```

The equivalent hexadecimal representation is:

```
B409
BA0901
CD21
CD20
54
```

```
68
69
73
20
69
73
20
43
24
```

The first instruction moves the value 09 into register AH. 09 identifies the instruction that displays characters starting at the offset stored in register DX. The second instruction moves the offset value 0109 into register DX. The third instruction executes the instructions stored in register AH: displays the characters starting at offset 0109. The fourth instruction stops execution. The fifth through thirteenth lines hold the characters to be displayed. The fourteenth line holds the terminator that identifies the end of the set of characters.

The assembly language version of these instructions provides a more readable program. Assembly language consists of symbols and values that are more readable than simple hexadecimal digits. The assembly language version looks like:

```
MOV AH,09
MOV DX,0109
INT 21
INT 20
DB 'T'
DB 'h'
DB 'i'
DB 's'
DB ' '
DB 'i'
DB 's'
DB ' '
DB 'C'
DB '$'
```

A Windows command line accepts assembly language instructions through the `a` input option on the `debug` program. Open a command prompt window and type the following:

```
debug
-a100
1456:0100 MOV AH,09   ;move code for displaying text into register AH
1456:0102 MOV DX,0109 ;move text address offset into register DX
1456:0105 INT 21      ;call the interrupt stored in register AH
1456:0107 INT 20      ;stop execution
1456:0109 DB 'T'      ;text
1456:010A DB 'h'      ;...
1456:010B DB 'i'      ;to
1456:010C DB 's'      ;...
1456:010D DB ' '      ;be
1456:010E DB 'i'      ;...
1456:010F DB 's'      ;displayed
1456:0110 DB ' '      ;...
1456:0111 DB 'C'
1456:0112 DB '$'      ;terminator character
1456:0113
-
```

`a` refers to the input option to the `debug` program. `100` identifies the offset in memory where the instructions start.

The first entry on each line is the memory address in segment:offset form. In `debug` applications, the segments share the same address (`0x14560`). The semi-colon refers to the end of a statement and the start of programmer comments.

To execute this program, we enter:

```
-g
This is C
Program terminated normally
-
```

To quit the debug program, we enter:

```
-q
```

The `debug` program uses an operating system program called an ***assembler*** to convert our assembly language instructions into binary information as shown in the figure below:

We call the binary result ***machine language***.

Operator Precedence

Operators	Associativity
`()` `[]` `++`(postfix) `--`(postfix)	left to right **HIGH** right to left
`++`(prefix) `--`(prefix) `!` `+`(unary) `-`(unary) `*` `&`	left to right right to left
`*` `/` `%`	left to right **MEDIUM**
`+`(binary) `-`(binary)	left to right
`<` `<=` `>` `>=`	left to right
`==` `!=`	left to right
`&&`	left to right
`||`	left to right
`=` `+=` `-=` `*=` `/=` `%=`	right to left
`?:`	right to left **LOW**
`,`	left to right

Unary `+`, `-` and `*` have higher precedence than the binary forms. The operator `()` refers to function call.

Precedence determines the order in which operands are bound to operators. Operators on the same line have the same precedence; rows are in order of decreasing precedence. C does *not* specify the order in which the operands of an operator are evaluated. Similarly, the order in which function arguments are evaluated is not specified. Examples:

```
// Example-1:
x = f() + g();

// Example-2:
a[i] = i++;

// Example-3:
printf("%d %d\n",++n,power(2,n));

// Example-4:
z = x / ++x;
```

Programs should not depend upon the order of evaluation of expressions, except as guaranteed by ANSI C for the following operators:

```
1.  a, b          comma operator (not the comma between arguments)
2.  a && b        logical and
3.  a || b        logical or
4.  a ? b : c     conditional
```

All of these guarantee that expression `a` will be computed before expression `b` (or `c`).

In addition, when a function-call takes place all arguments are evaluated before control transfers to the function.

```
5.  a(b) function call
```

ANSI C++ guarantees that each full expression will be evaluated before going on.

```
6. each full expression
```

Suggested Weekly Reading Schedule

Week	Topic
1	• Computers • Information • Compilers
2	• Types • A Simple Calculation • Expressions
3	• Logic • Style Guidelines • Testing and Debugging
4	• Arrays
5	• Structure Types
6	• Functions • Pointers
7	• Functions, Arrays and Structs
8	• Character Strings • Input Functions • Output Functions • Library Functions
9	• String Library
10	• Text Files • Records and Fields
11	• More Input and Output • Pointers Arrays Structures
12	• Two-Dimensional Arrays • Algorithms
13	• Portability