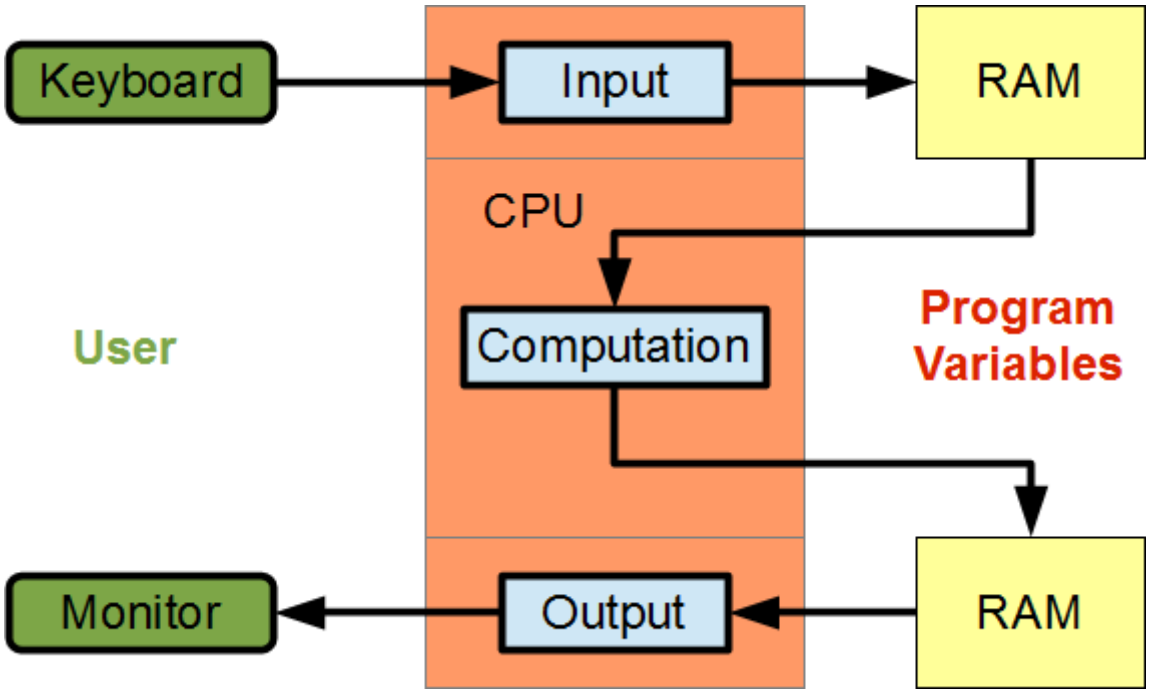
A Simple Calculation

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	*
single quote	\'	39	125
double quote	\"	34	127
question mark	\?	63	111

> **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

```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

    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

```c
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:

```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

    // ... 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:

```c
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.

Specifier	Output as a	Use With Type	Most Common
%c	character	char	*
%d	decimal	char, int	*
%f	floating-point	float	*
%lf	floating-point	double	*

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.

Arrays

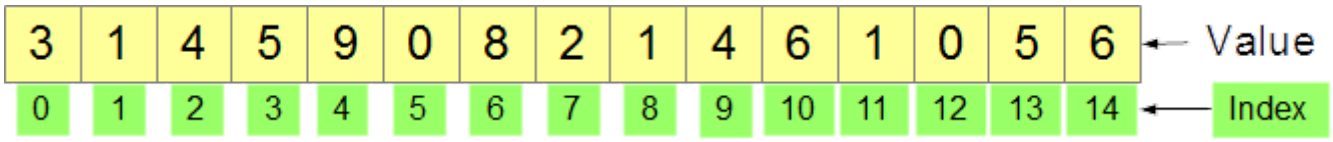
Learning Outcomes

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

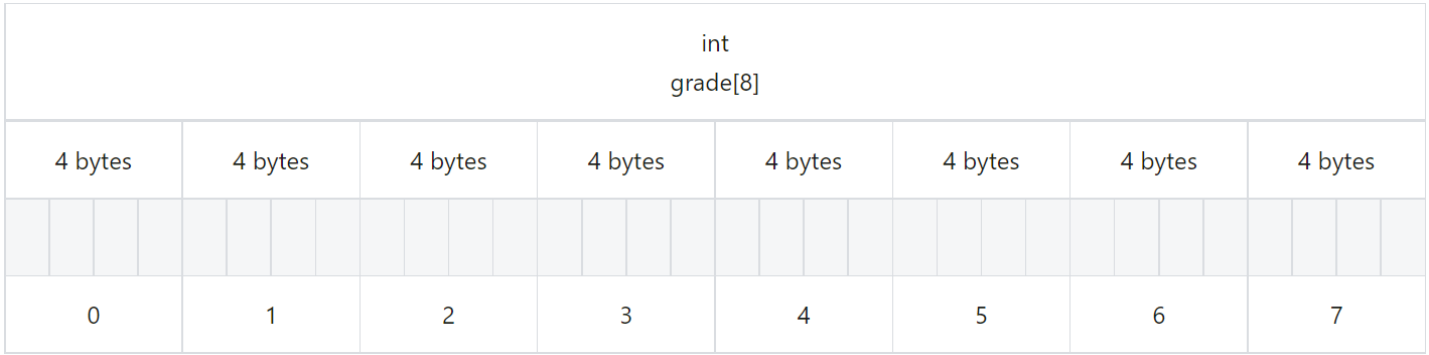
```
type identifier [ size ];
```

`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};
```

Array Identifier	int grade							
Value	10	9	10	8	7	9	8	10
Element Index	0	1	2	3	4	5	6	7

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'`.

char																
																\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.

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***

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.

Compilers

Learning Outcomes

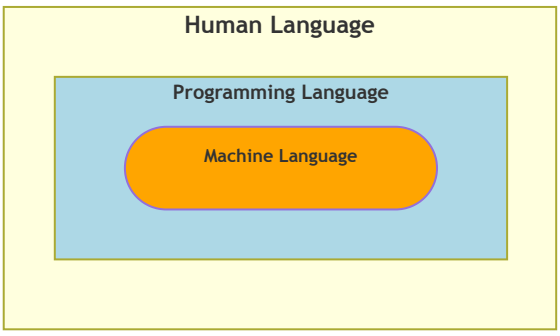
After reading this section, you will be able to:

- 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.