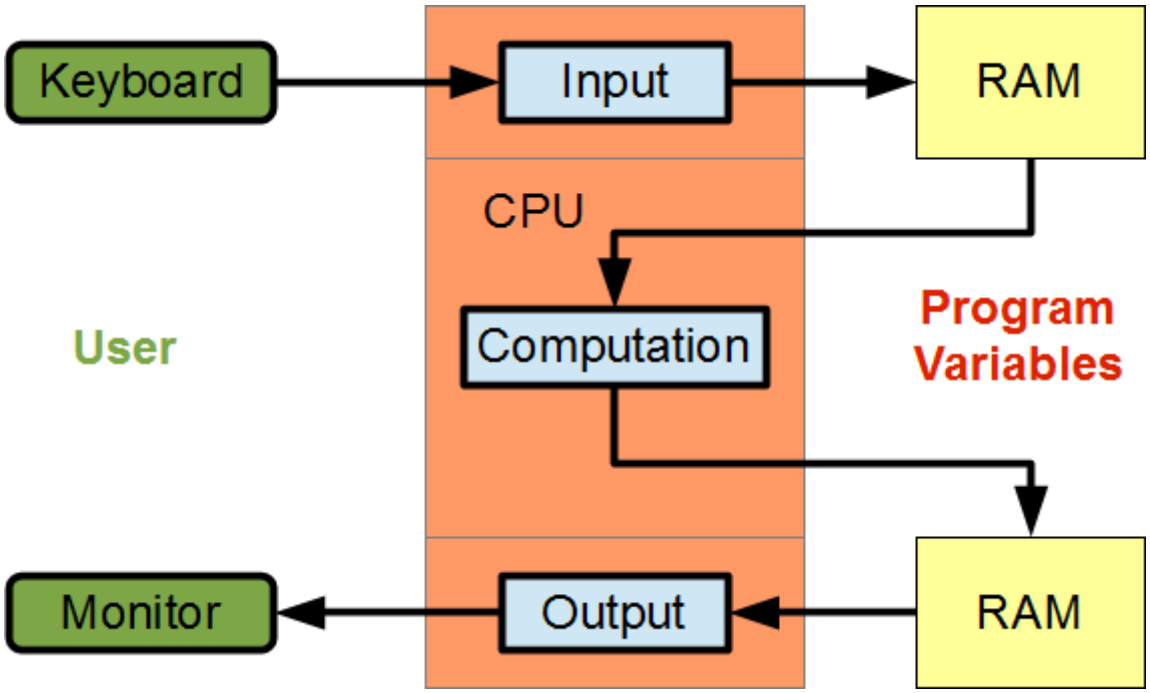
A Simple Calculation

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.