C++ PROGRAMMING:

FROM PROBLEM ANALYSIS TO PROGRAM DESIGN

BY: D. S. MALIK

CHAPTER 1: AN OVERVIEW OF COMPUTERS AND PROGRAMMING LANGUAGES

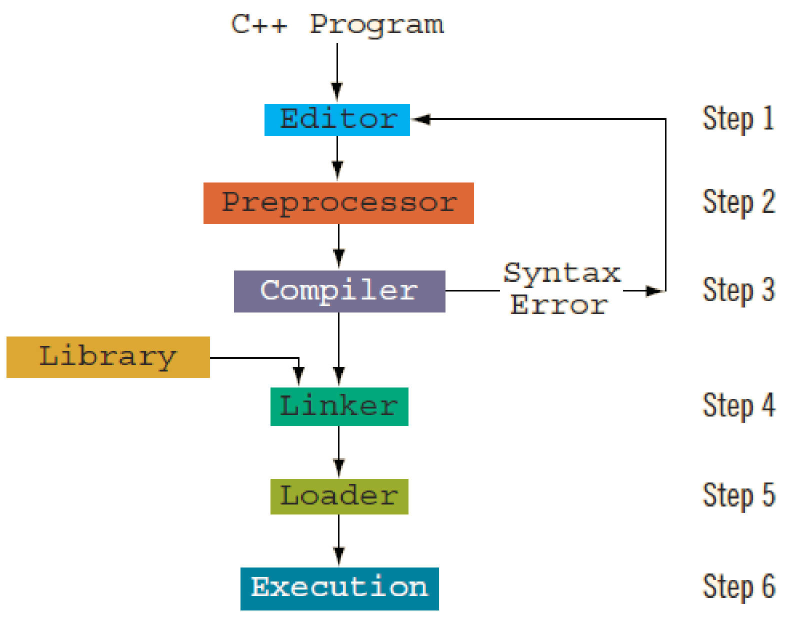
SUMMARY & EXAMPLES

PREPARED BY:

E. MANAR JARADAT

CHAPTER 1: AN OVERVIEW OF COMPUTERS AND PROGRAMMING LANGUAGES

- A **computer** is an electronic device capable of performing the commands.
- A computer composed of two main components:
    - Hardware components:
        - Input/Output devices to perform input data and output result.
        - Memory to store data.
        - CPU to perform arithmetic and logical operations.
    - Software Components:
        - They are programs written using a programming language to perform specific tasks.
        - Programming languages examples: C, C++, C#, Java, Python, R ...
- How do computers process C++ programs?
    1. A text **editor** is used to create a C++ program following it's rules, or syntax.
        - The created program is called the **source code**, or **source program**.
        - The program must be saved in a text file that has the extension `.cpp`.
    2. In a C++ program, statements that begin with the symbol `#` are called preprocessor directives. These statements are processed by a program called **preprocessor**.
    3. A program called the **compiler** checks the source program for syntax errors.
        - **Recall**, computers understand binary language only which we can’t speak.
        - If the source program has no syntax error, the compiler also acts like a translator to translate source program to machine language (object program).
        - **Object program**: The machine language version of the high-level language program.
    4. A program called a **linker** combines the object program with the programs from libraries (prewritten programs). When the object code is linked with the system resources, the executable code is produced and saved in a file with the file extension `.exe`.
    5. The translated program stored in memory. A program called a **loader** loads the executable program into main memory for execution.
    6. The final step is to execute the program.
- The figure below shows how a typical C++ program is processed.



Processing a C++ program

- **An Integrated Development Environment (IDE)** is a software that contain programs to helps you to do all the previous steps using Build and Rebuild command
    - examples: Microsoft Visual Studio, Eclipse, Code::Blocks.

Programming with the Problem Analysis–Coding–Execution Cycle

- **Algorithm:** is a step-by-step problem-solving process (transform input to output) in a finite amount of time.
- **Programming:** is a process of problem-solving and converting algorithms to executable programs.
- The Problem Analysis–Coding–Execution Cycle
    - Step 1: Analyze the problem and outline it's requirements.
    - Step 2: Implement the algorithm in a programming language.
    - Step 3: Maintenance the algorithm until get the expected results.
    - Step 4: Execution.
- **EXAMPLE 1-1:** Design an algorithm to find the perimeter and area of a rectangle.
    - **Output:** Rectangle area and perimeter.
    - **Input:** Rectangle length and width.
    - **Algorithm:**
        1. Get the length of the rectangle.
        2. Get the width of the rectangle.
        3. Find the perimeter using the following equation: $perimeter = 2 * (length + width)$
        4. Find the area using the following equation:
           $area = length * width$

- **EXAMPLE 1-2:** Design an algorithm that calculates the sales tax and the final price of an item sold in a particular state. Knowing that, the sales tax is calculated as follows: The state's portion of the sales tax is 4%, and the city's portion of the sales tax is 1.5%. If the item is a luxury item, such as a car more than $50,000, then there is a 10% luxury tax.
  Knowing that the final price of the item is the selling price in addition to the sales tax.
- **Output:** The sales tax, the final price of an item.
- **Input:** The selling price of the item and whether the item is a luxury or not.
- **Algorithm:**
  1. Get the selling price of the item.
  2. Find the state's portion of the sales tax using the formula:
     $stateSalesTax = salePrice * 0.04$
  3. Find the city's portion of the sales tax using the formula:
     $citySalesTax = salePrice * 0.015$
  4. Determine whether the item is a luxury item.
  5. Find the luxury tax using the following formula:

```
if (item is a luxury item)
    luxuryTax = salePrice * 0.1
otherwise
    luxuryTax = 0
```

  6. Find the sales tax using the formula:
     $salesTax = stateSalesTax + citySalesTax + luxuryTax$
  7. Find amount due using the formula:
     $amountDue = salePrice + salesTax$

- **EXAMPLE 1-3:** Design an algorithm that calculates the monthly paycheck of a salesperson at a local department store. Knowing that every salesperson has a base salary. The salesperson also receives a bonus at the end of each month, based on the following criteria: If the salesperson has been with the store for five years or less, the bonus is $10 for each year that he or she has worked there. If the salesperson has been with the store for more than five years, the bonus is $20 for each year that he or she has worked there. The salesperson can earn an additional bonus as follows: If the total sales made by the salesperson for the month are at least $5,000 but less than $10,000, he or she receives a 3% commission on the sale. If the total sales made by the salesperson for the month are at least $10,000, he or she receives a 6% commission on the sale.
  - **Output:** Monthly paycheck of a salesperson
    $payCheck = baseSalary + bonus + additionalBonus$
  - **Input:** Salesperson base salary, number of his service years, his total sales.
  - **Algorithm:**
    1. Get salesperson base salary,
    2. Get number of salesperson service years.
    3. Calculate **bonus** using the following formula:

```
if (noOfServiceYears is less than or equal to five)
    bonus = 10 * noOfServiceYears
otherwise
    bonus = 20 * noOfServiceYears
```

4. Get amount of salesperson total sales.
5. Calculate additional bonus using the following formula:

```
if (totalSales is less than 5000)
    additionalBonus = 0
otherwise
    if (totalSales is greater than or equal to 5000 and
        totalSales is less than 10000)
        additionalBonus = totalSales * (0.03)
    otherwise
        additionalBonus = totalSales * (0.06)
```

6. Calculate pay check using the equation:

```
payCheck = baseSalary + bonus + additionalBonus
```

- **EXAMPLE 1-4**: Design an algorithm to play a number-guessing game. Knowing that the number-guessing game is played as follows: An integer greater than or equal to 0 and less than 100 is randomly generated. Then prompt the player (user) to guess the number. If the player guesses the number correctly, output an appropriate message. Otherwise, check whether the guessed number is less than the random number. If the guessed number is less than the random number generated, output the message, “Your guess is lower than the number. Guess again!”; otherwise, output the message, “Your guess is higher than the number. Guess again!”. Then prompt the player to enter another number. The player is prompted to guess the random number until the player enters the correct number.
    - **Output:** A message that congratulates the player for correctly guessing the secret number.
    - **Input:** a randomly generated secret number, and user guessing's
    - **Algorithm:**
        1. Generate a random number and call it `num`.
        2. Repeat the following steps until the player has guessed the correct number:

```
Prompt the player to enter guess.
if (guess is equal to num)
    Print "You guessed the correct number."
otherwise
    if guess is less than num
        Print "Your guess is lower than the number. Guess
again!"
    otherwise
        Print "Your guess is higher than the number. Guess
again!"
```

- **EXAMPLE 1-5:** There are 10 students in a class. Each student has taken five tests, and each test is worth 100 points. Design an algorithm to calculate the grade for each student, as well as the class average. The grade is assigned as follows: If the average test score is greater than or equal to 90, the grade is A; if the average test score is greater than or equal to 80 and less than 90, the grade is B; if the average test score is greater than or equal to 70 and less than 80, the grade is C; if the average test score is greater than or equal to 60 and less than 70, the grade is D; otherwise, the grade is F.
  - **Output**: The grade for each student, and the class average grade.
  - **Input**: 5 scores for each of the ten student.
- **Algorithm to calculate a student grade:**
  1. Get the five test scores.
  2. Add the five test scores. Suppose sum stands for the sum of the test scores.
  3. Suppose average stands for the average test score. Then average = sum / 5;
  4. Calculate the grade according to the following algorithm:

```
if average is greater than or equal to 90
    grade = A
otherwise  if average is greater than or equal to 80 and less than 90
    grade = B
otherwise  if average is greater than or equal to 70 and less than 80
    grade = C
otherwise  if average is greater than or equal to 60 and less than 70
    grade = D
otherwise
    grade = F
```

  - Algorithm to calculate class average grade:
    1. total Average = 0;
    2. Repeat the following steps for each student in the class:
    3. Use the algorithm as discussed above to find the average test score.
    4. Use the algorithm as discussed above to find the grade.
    5. Update total Average by adding the current student’s average test score.
  3. Determine the class average as follows:
  $classAverage = totalAverage/10$

C++ PROGRAMMING:

FROM PROBLEM ANALYSIS TO PROGRAM DESIGN

BY: D. S. MALIK

CHAPTER 2: BASIC ELEMENTS OF C++

SUMMARY & EXAMPLES

PREPARED BY:

E. MANAR JARADAT

CHAPTER 2: BASIC ELEMENTS OF C++

- A computer **program** is a sequence of statements whose objective is to accomplish a task.
- **Programming** is a process of planning and creating a program.
- **Programming language**: is a set of syntax and semantic rules, symbols, and special words.
    - The **syntax rules** tell which statements (instructions) are legal or valid and accepted by the programming language and which are not.
    - The **semantic rules** determine the meaning of the instructions.

A Quick Look at a C++ Program

- Study the following C++ code snippet which prints a welcoming statement to computer programming course.

```cpp
// First C++ program
#include <iostream>
using namespace std;

int main()
{
    cout << "Welcome to Computer programming course";
    return 0;
}
```

OUTPUT:

```
Welcome to Computer programming course
```

The `main` Function

- A C++ program is a collection of functions, one of which is the function `main`.
- A function is a set of instructions designed to accomplish a specific task.
- If a C++ program consists of only one function, then it must be the function `main`.
- The syntax of the function main used throughout this course has the following form:

```cpp
int main()
{
    statement_1
    .
    .
    .
    statement_n
    return 0;
}
```

- The basic parts of the function `main` are the heading and the body.
    - The first line of the function `main`, is called the heading of the function `main`.

      ```
      int main()
      ```

    - The statements enclosed between the curly braces `{` and `}` form the body of the function `main`, and it contains two types of statements:
        - **Declaration statements:** used to declare things, such as variables.
        - **Executable statements:** perform calculations, manipulate data, create output, accept input, and so on.
- All C++ statements must end with a semicolon `;`. The semicolon is also called a statement terminator.
- As soon as the statement `return 0;` is executed, the execution of the program stops immediately.
- The statement `return 0;` returns the exit code 0 to the operating system, and this indicates that the program was executed successfully.
- If the statement `return 0;` is misplaced in the body of the function main, the results generated by the program may not be to your liking.
- **EXAMPLE 2-1:** What is the output of the following program?

  ```
  #include <iostream>
  using namespace std;
  int main()
  {
      return 0;
      cout << "Welcome to Computer programming course";
  }
  ```

  OUTPUT:

  ```

  ```

- Using `return 0;` at the end of function `main` is optional, that is; if you forget it the program will return 0 to the operating system once it reaches the end of the function.
- In C++, `return` is a reserved word.

Comments

- Comments are notes written by the programmer for the code reader, and they are completely ignored by the compiler.
- Why would programmers use comments?
    - To identify the authors of the program.
    - To give the date when the program is written or modified.
    - To give a brief explanation of the program, and explain the meaning of key statements in it.
- There are two common types of comments in a C++ program

- **Single-line comments:** begin with `//`
- **Multiple-line comments:** enclosed between `/*` and `*/`.

- Single-line comments and can be placed anywhere in the line. Everything encountered in that line after `//` is ignored by the compiler.
- The following C++ code snippet demonstrates the use of single-line comments.

```
// First C++ program
int main()  // function main header
{
    // function main body
}
```

- Multiple-line comments start with `/*`, and the compiler will ignore anything after it until it reaches `*/`.
- The following C++ code snippet demonstrates the use of multiple-line comments.

```
/* First C++ Program*/
int main()  // function main header
{
    /*
    In C++ comments
    can span
    multiple lines
    cout<<"Go Ahead..."<<endl;
    */
    cout<<"Go Ahead..."<<endl;
}
```

Preprocessor Directives

- Only a small number of operations, such as arithmetic and assignment operations, are explicitly defined in C++.
- Many of the functions and symbols needed to run a C++ program are provided as a collection of libraries.
- Every library has a name and is referred to by a header file.
- For example, the descriptions of the functions needed to perform input/output (I/O) are contained in the header file `iostream`.
- If you want to use functions from a library, you use preprocessor directives and the names of header files to tell the computer the locations of the code provided in libraries.
- **Preprocessor directives** are commands supplied to the preprocessor that cause the preprocessor to modify the text of a C++ program before it is compiled.
- All preprocessor commands begin with `#`.
- There are no semicolons at the end of preprocessor commands because they are not C++ statements.
- To use a header file in a C++ program, use the preprocessor directive `include`.
- The general syntax to include a header file (provided by the IDE) in a C++ program is:

```
#include <headerFileName>
```

- **EXAMPLE 2-4:** The following statement includes the header file `iostream` in a C++ program:

```
#include <iostream>
```

- Preprocessor directives to include header files are placed as the first line of a program so that the identifiers declared in those header files can be used throughout the program.
- Both `cin` and `cout` are predefined identifiers declared in the header file `iostream`, within a namespace `std`.
- There are several ways you can use an identifier declared in the namespace `std`.
  - Refer to them as `std::cin` , `std::cout` and `std::endl` throughout the program.

```
#include <iostream>

int main()
{
    std::cout<<"Hello everybody!"<<std::endl;
}
```

  - Another option is to add the following statement in your program `using namespace std;` after the statement `#include` . You can then refer to `cin` and `cout` without using the prefix `std::`.

```
#include <iostream>
using namespace std;

int main()
{
    cout<<"Hello everybody!"<<endl;
}
```

- In C++, `namespace` and `using` are reserved words.

Output

- The standard output device is usually the screen.
- In C++, output on the standard output device is accomplished via the use of `cout` and the stream insertion operator `<<`.
- The general syntax of an output statement (`cout` together with `<<`) is:

```
cout << expression or manipulator << expression or manipulator...;
```

- C++ output statement causes the computer to evaluate the expression after the stream insertion operator `<<` and display the result on the screen starting from the insertion point (where the cursor is).

    - If the expression was a string (anything in double quotes `" "`) it evaluates to itself.
    - Arithmetic expressions are evaluated according to rules of arithmetic operations, which you typically learn in an algebra course.
    - Manipulators change the format of the output as we will study later.
- **EXAMPLE 2-2:** What is the output of the following statements?

```
cout << "Welcome ";
cout<<2<<"C++ course ";
cout<<5*20<<" Times :)";
```

OUTPUT:

```
Welcome 2C++ course 100 Times :)
```

- In C++, you can use manipulators in **output statements** to format the output.
- The simplest manipulator is `endl`, which causes the insertion point to move to the beginning of the next line.
- **EXAMPLE 2-3:** What is the output of the following statements?

```
cout << "Welcome "<<endl;
cout<<2<<"C++ course "<<endl;
cout<<5*20<<endl<<" Times :)";
```

OUTPUT:

```
Welcome
2C++ course
100
 Times :)
```

- **EXAMPLE 2-4:** What is the output of the following code snippets.

```
cout << 29 / 4 << endl;
cout << "Hello there." << endl;
cout << 12 << endl;
cout << "4 + 7" << endl;
cout << 4 + 7 << endl;
cout << 'A' << endl;
cout << "4 + 7 = "<<endl << 4 + 7 << endl;
cout << "4 + 7 = " << 4 + 7 << endl;
cout << 2 + 3 * 5 << endl;
cout << "Hello \nthere." << endl;
```

OUTPUT:

```
7
Hello there.
12
4 + 7
11
A
4 + 7 =
11
4 + 7 = 11
17
Hello
there.
```

- **EXAMPLE 2-5:** Write a C++ program to print the area and perimeter of a rectangle which have a length of 5 and a width of 10.

```cpp
cout << "Area = "<< 10 * 5 << endl;
cout << "Perimeter = "<< 2 * (10 + 5) << endl;
```

OUTPUT:

```
Area = 50
Perimeter = 30
```

- **NOTE:** When an output statement outputs the value of a string, it outputs only the string without the double quotes `" "` (unless you include double quotes as part of the output).

Escape Sequences

- In C++, the backslash `\` is called the escape character.
- `\n` is called the **newline escape sequence**.
- `\n` may appear anywhere in the **string**, and when it is encountered, the insertion point is positioned at the beginning of the next line.
- **EXAMPLE 2-6:** What is the output of the following C++ statements

```cpp
cout << "Welcome \n";
cout<<2<<"C++ course \n";
cout<<5*20<<"\n Times :)"<<endl;
cout << "Hello \nthere. \nMy name is Manar.";
```

OUTPUT:

```
Welcome
2C++ course
100
Times :)
Hello
there.
My name is Manar.
```

- Note that the output of the following three statements is the same:
    - `cout << '\n';`
    - `cout << "\n";`
    - `cout << endl;`
- **EXAMPLE 2-7:** Consider the following C++ statements:

```
cout<<"1"<<'\n'<<"22"<<endl<<"333"<<"\n"<<"444\n4";
```

OUTPUT

```
1
22
333
444
4
```

- The return (or Enter) key on your keyboard cannot be part of the string. That is, in programming code, a string cannot be broken into more than one line by using the return (Enter) key on your keyboard.
- Study the following 2 statements

```
cout << "It is sunny day. "
<< "We can go golfing." << endl; // legal statement

cout << "It is sunny day.
We can go golfing." << endl;    // illegal statement
```

- The following table lists some of the commonly used escape sequences.

Escape	Sequence	Description
\n	Newline	Cursor moves to the beginning of the next line
\t	Tab	Cursor moves to the next tab stop
\b	Backspace	Cursor moves one space to the left
\r	Return	Cursor moves to the beginning of the current line.
\\	Backslash	Backslash is printed

Escape	Sequence	Description
\'	Single quotation	Single quotation mark is printed
\"	Double quotation	Double quotation mark is printed

- **EXAMPLE 2-8:** What is the output of the following statements?

	Sequence	Description
1.	`cout<<"Hello\nEverybody"<<endl;`	Hello Everybody
2.	`cout<<"Hello\tEverybody"<<endl;`	Hello Everybody
3.	`cout<<"Hello\bEverybody"<<endl;`	HellEverybody
4.	`cout<<"Hello\rEverybody"<<endl;`	Everybody
5.	`cout<<"Everybody\rhello"<<endl;`	Hellobody
6.	`cout<<"Hello\\Everybody"<<endl;`	Hello\Everybody
7.	`cout<<"Hello\"Every\'Body"<<endl;`	Hello"Every'Body
8.	`cout<<"Every\"Body\rHello"<<endl;`	Hello"Body
9.	`cout<<"HelloEverybody\b"<<endl;`	HelloEverybody

10.	`cout<<"HelloEverybody\b";`
	`cout<<"Hello\\Everybody"<<endl;`
		HelloEverybodHello\Everybody

11.	`cout<<"HelloEverybody\b";`
	`cout<<"Everybody\rGood"<<endl;`
		GoodoEverybodEverybody

The Basics of a C++ Program

- The **token** is the smallest individual unit of a program written in any language.
- C++'s tokens are divided into special symbols, word symbols, and identifiers.

Special Symbols

- Special symbols in C++ include:
  - Punctuators: `[` `]` `(` `)` `{` `}` `,` `;` `:` `*` `#`
  - Operators (arithmetical operators, Relational operators, Logical operators, Unary operators, Assignment operators, Conditional operators, Comma operator). `<=` `>=` `==` `!=` `+` `-` `*` `/` `+=`
  - The blank,that you create a blank symbol by pressing the space bar (only once) on the keyboard.
- Special symbols cannot be used for anything other than their intended use.
- No character can come between the two characters in the token, not even a blank.

Whitespaces

- Whitespaces include blanks, tabs, and newline characters.
- In a C++ program, whitespaces are used to separate special symbols, reserved words, identifiers, and numbers when data is input.
- Blanks must never appear within a reserved word or identifier.
- Utilization of whitespaces is important.

Reserved Words (Keywords)

- Some of the reserved word symbols include the following:

  `int`, `float`, `double`, `char`, `const`, `void`, `return`

- The letters that make up a reserved word are always **lowercase**.
- Word symbols cannot be redefined within any program;that is, they cannot be used for anything other than their intended use.

Identifiers

- Identifiers are names of things that appear in programs, such as variables, constants, and functions.
- All identifiers must obey C++'s rules for identifiers.
  - It consists of letters, digits, and the underscore character `_` .
  - It must begin with a letter or underscore.
  - It must not have any special symbols like space or arithmetic operators.
  - C++ is case sensitive. The identifier `Num` is not the same as the identifier `num`.
  - Reserved words cannot be used as identifiers.
- It is preferred to use self-documenting identifiers, they make comments less necessary.
- Two predefined identifiers that you will encounter frequently are `cout` and `cin`.
- Predefined identifiers can be redefined, but it would not be wise to do so.
- **EXAMPLE 2-9:** Which of the following is a legal identifier in C++

Identifier	Is legal?	Description
first	yes	
payRate	yes	
counter1	yes	
_first	yes	
employee Salary	false	It has space
Hello!	false	It has special symbol `!`
one+two	false	It has special symbol `+`
2nd	false	It starts with a number

VARIABLES

- A **variable** is a memory location whose content may change during program execution.
- Declaring variable means instructing the computer to allocate memory, and this can be accomplished by.
  1. Determine the names to use for each memory location.
  2. Determine the type of data to store in those memory locations e.g. `int`, `double`.
- The syntax rule to declare a variable is:

```
dataType identifier;
```

- Study the following C++ statements:

```cpp
double amountDue;
int counter;
int x;
```

- The syntax rule to declare one variable or multiple variables of the same data type is:

```
dataType identifier, identifier, . . .;
```

- Study the following C++ statements:

```cpp
int day, month, year;
double length, width;
```

- Using semicolon to separate the definition of multiple variables will result in syntax error.

```cpp
int x1; x2; x3;      // syntax error
```

- In C++, you must declare all identifiers before you can use them. otherwise the compiler will generate a syntax error.

```cpp
cout << "num = " << num << endl;    // error: 'num' was not
                                    // declared in this scope
```

- How do you put data into variables? In C++, you can place data into a variable in two ways:
  - Use C++'s assignment statement.
  - Use input (read) statements.

Initializing Variables using Assignment Statement

- A variable is said to be initialized the first time a value is placed in it.
- When a variable is declared, C++ may not automatically initialize variables.
- If you only declare a variable and do not instruct the computer to put data into the variable,the value of that variable is garbage, and the computer performs calculations using those values in memory.
- Using a variable without being initialized might give unexpected results and/or the complier might generate a warning message indicating that the variable has not been initialized.

```
int num1;
cout << "num1 = " << num1 << endl;  //num1 = ??
```

- Assignment operator `=` can be used to assign data into variables.
- The assignment statement takes the following form:

```
variable = expression;
```

- The expression on the right side is evaluated, and its value is assigned to the variable on the left side.

```
int first;          // declaration statement
first = 13 - 4;     // inialization statement
                    // first = 9
```

- You can declare and initialize variables at the same time

```
int second = 12;    // declaration and inialization statement
```

- You can define multiple variables and initialize them in a single C++ statement

```
int x = 5, y = 7, z = 12;
```

- **EXAMPLE 2-10:** What are the values stored in the variables `num1`, `num2` and `num3` after executing each of the following statements.

```
int num1, num2, num3;   // num1 = ??, num2 = ??, num3 = ??
num1 = 18;              // num1 = 18, num2 = ??, num3 = ??
num1 = num1 + 27;       // num1 = 45, num2 = ??, num3 = ??
num2 = num1;            // num1 = 45, num2 = 45, num3 = ??
num3 = num2 / 5;        // num1 = 45, num2 = 45, num3 = 9
num3 = num3 / 4;        // num1 = 45, num2 = 45, num3 = 2
cout<<num1<<"\t"<<num2<<"\t"<<num3<<"\n"; //45  45  2
```

- The assignment operator, `=`, is evaluated from right to left, the associativity of the assignment operator is said to be from right to left.

```cpp
int num1, num2 = 2, num3 = 4;   // num1 = ?, num2 = 2, num3 = 4
num1 = num2 = num3;             // num1 = 4, num2 = 4, num3 = 4
num1 = num2 = num3 = 9;         // num1 = 9, num2 = 9, num3 = 9
```

- To save the value of an expression and use it in a later expression, do the following:
  - Declare a variable of the appropriate data type.
  - Assign the value of the expression to the variable that was declared, using the assignment statement.
  - Wherever the value of the expression is needed, use the variable holding the value.

NAMED CONSTANTS

- **Named constant** is a memory location whose content is not allowed to change during program execution.
- The syntax to declare a named constant is:

```cpp
const dataType identifier = value;
```

- In C++, `const` is a reserved word.
- C++ programmers typically prefer to use **uppercase letters** to name a named constant.
- If the name of a named constant is a combination of more than one word, then the words are separated using an underscore.
- Study the following C++ statements:

```cpp
const double PI = 22.0 / 7.0;
const int NO_OF_STUDENTS = 20;
```

- If you declare a named constant you must initialize it on the same statement, otherwise the compiler will generate a syntax error.

```cpp
const double PI;         // error: unintialized const variable
PI = 22.0 / 7.0;         // error: assignment of read only variable
```

- Any attempt to modify the value of a named constant will cause the compiler to generate a syntax error.

```cpp
const double PI = 22.0 / 7.0;
PI = 3.14;                      // error: assignment of read only variable
```

- **EXAMPLE 2-11:** Write a program to calculate and print the volume of a cylinder whose $radius = r = 5$ and $height = h = 10$, knowing that $volume = base\_area * height = \pi r^2 h$

```
double r, h, area, volume;
r = 5;
h = 10;
const double PI = 22.0 / 7.0;
area = PI * r * r;
volume = area * h;
cout<<"base area    = "<<area<<endl;
cout<<"cylinder volume = "<<volume<<endl;
```

OUTPUT:

```
cylinder area   = 78.5714
cylinder volume = 785.714
```

Data Types

- **Data type** is a set of values together with a set of operations.
- C++ is a strongly typed programming language since we need to specify the data type of each created variable.
- Specifying the data type of a variable is useful to:
    - Determine the size and layout of the variable's memory
    - Determine the range of values that can be stored within that memory
    - Determine the set of operations that can be applied to the variable.
- C++ data types fall into the following three categories:
    - Simple data type
    - Structured data type
    - Pointers

Simple Data Types

- The simple data type is the fundamental data type in C++ because it becomes a building block for the structured data type.
- There are three categories of simple data:
    - **Integral data types**: deal with integers, or numbers without a decimal part. e.g. `char`, `short`, `int`, `long long`, `bool`, `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`.
    - **Floating-point data types**: deals with decimal numbers. e.g. `float`, `double`.
    - **Enumeration data types:** deals with user-defined data type.
- The following table gives the range of possible values associated with some integral data types and the size of memory allocated to manipulate these values.

Data Type	Values	Storage (in bytes)
bool	true and false	1
char	-128 to 127	1
short	–32768 to 32767	2

Data Type	Values	Storage (in bytes)
int	-2147483648 to 2147483647	4
long long	-2^{63} to $2^{63}-1$	64

- Which data type you use depends on how big a number your program needs to deal with.
    - Which data type would you use to store whether a student pass an exam or not?
    - An international company has 5000 employees, which data type would you use to store number of employees in this company?
    - Assume the distance between two cities in Jordan that is 79.7 KM, which data type would you use to store the distance ?
- Different compilers may allow different ranges of values.

`int` Data Type

- Integers in C++, as in mathematics, are numbers without decimal parts.
- For example: `-6728`, `-67`, `0`, `78`, `36782`, `+763`
- Positive integers do not need a `+` sign in front of them.
- Study the following C++ statements.

```
int num1 = -12;
num1 = +3;
num1 = 6;
num1 = 12000;
```

- In C++, commas are used to separate items in a list, and you cannot use them within an integer.

```
int num1 = 12,000;   // is illegal
```

- **Recall:** In C++, you can store any number from -2147483648 to 2147483647 in an integer data type.
- **EXAMPLE 2-12:** Execute the following code segment on your preferred C++ compiler and note the result of the second output statement

```
int x = 1000;
x = x*x;
cout << x << endl;
x = x * x;
cout << x << endl;
```

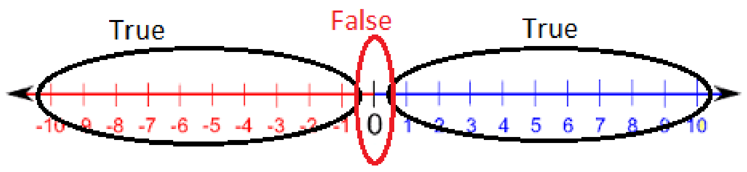
OUTPUT

```
1000000
-727379968
```

- Explain the result and think in a solution for this problem.

`bool` Data Type

- The data type `bool` has only two values: `true` and `false`.
- `true` and `false` are called the logical (Boolean) values.
- The central purpose of this data type is to manipulate logical (Boolean) expressions.
- In C++, `bool`, `true`, and `false` are reserved words.
- Any none zero value is considered as true.



- Study the following C++ statements.

```
bool b1;
b1 = true;       //b1 = 1
b1 = false;      //b1 = 0
b1 = 0;          //b1 = 0
b1 = 100;        //b1 = 1
b1 = -150;       //b1 = 1
```

`char` Data Type

- The data type char is the smallest integral data type.
- Can store numbers in the range -128 to 127.
- It is mainly used to represent single characters—that is, letters, digits, and special symbols.
- All the individual symbols located on the keyboard that are printable may be considered as possible values of the char data type.
- When using the char data type, you enclose each character represented within single quotation marks.
- `'A'`, `'a'`, `'0'`, `'*'`, `'+'`, `'$'`, `'&'`, `' '`, `'_'`
- When an output statement outputs char values, it outputs only the character without the single quotes.
- **EXAMPLE 2-13:** What is Consider the output of following code snippet?

```
char score = 'A';
char plus = '+';
char num = '5';
const char BLANK = ' ';
const char DOLLAR = '$';
cout<<score<<plus<<num<<BLANK<<DOLLAR;
```

OUTPUT:

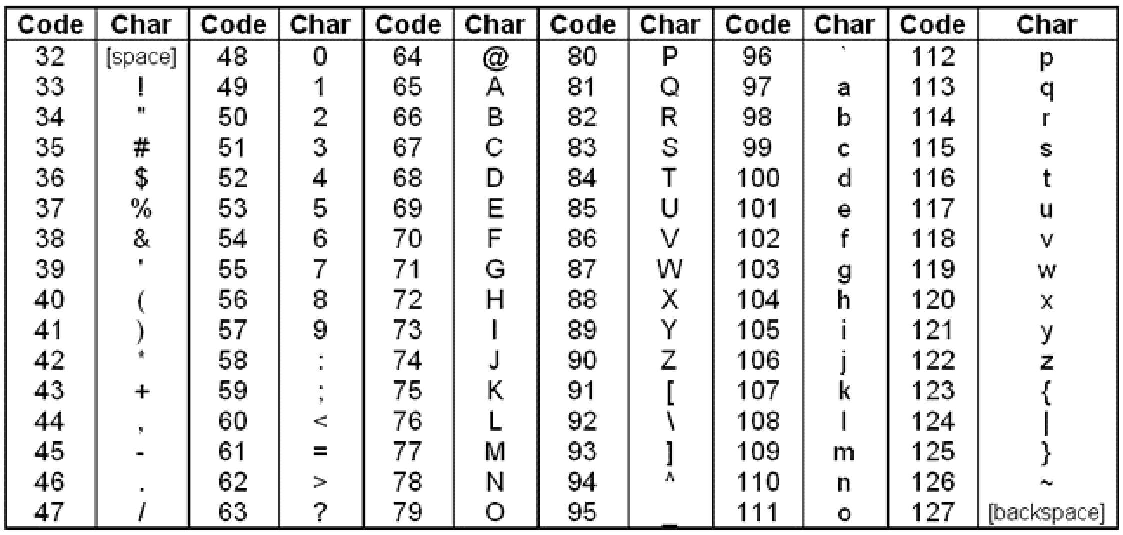
```
A+5 $
```

- Note that a blank space is a character and is written as `' '`, with a space between the single quotation marks.
- The data type char allows only one symbol to be placed between the single quotation marks.
    - `'abc'` and `'!='` is not of the type `char`.
    - If you enclose multiple symbols in single quotes the compiler will save the `last character` in our variable

    ```
    char ch5 = 'abd';        // The compiler will give you a warning
    cout<<ch5; //d
    ```

ASCII Code Set

- Recall, computers can only work with binary data, so we have different character encoding systems that give each character a unique numeric code.
- Several different character data sets are currently in use. The most common are the American Standard Code for Information Interchange (ASCII)
- The ASCII character set has 128 values.
    - Each of the 128 values of the ASCII character set represents a different character.
    - Each character has a predefined ordering represented by the numeric value associated with the character. This ordering is called a **collating sequence**, in the set.
- The order of characters in the ASCII code set (collating sequence) is used when you compare characters.
- The table below includes the ASCII table for most characters in your keyboard.



Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

- Variables of `char` data type can be initialized in the following two ways
  - single symbol enclosed in single quotes.
  - The symbol equivalent ASCII code.
- **EXAMPLE 2-14:** What is the output of the following code snippet?

```
char ch1 = 65;       // ch = A
char ch2 = 50;       // ch = 2
char ch3 = 99;       // ch = c
char ch4 = 63;       // ch = ?
cout<<ch1<<ch2<<ch3<<ch4;
```

OUTPUT:

```
A2c?
```

Floating-Point Data Types

- C++ provides the floating-point data type to deal with decimal numbers.
- Decimal numbers usually written using the scientific notation. For example:
  - $43872918 = 4.3872918 * 10^7$
  - $.0000265 = 2.65 * 10^{-}5$
  - $47.9832 = 4.79832 * 10^1$
- C++ uses a form of scientific notation called **floating-point notation** to represent decimal numbers.
- In the C++ floating-point notation, the letter **E** stands for the **exponent**.
- **EXAMPLE 2-18:** The following table shows some decimal numbers and their representation using scientific notation and C++ floating-point notation.

	Decimal Number	Scientific Notation	C++ Floating-Point Notation
1.	75.924	$7.5924 * 10^1$	7.592400E1
2.	0.18	$1.8 * 10^{-1}$	1.800000E-1
3.	0.0000453	$4.53 * 10^{-5}$	4.530000E-5

	Decimal Number	Scientific Notation	C++ Floating-Point Notation
4.	-1.482	$-1.482 * 10^0$	-1.482000E0
5.	7800.0	$7.8 * 10^3$	7.800000E3

- **EXAMPLE 2-15:** What is the equivalent decimal numbers in scientific notation for the following numbers represented using C++ floating-point notation?

	C++ Floating-Point Notation	Scientific Notation
1.	3.450000E20	$3.450 * 10^{20}$
2.	7.940000E-13	$7.940 * 10^{-13}$
3.	2.623000E0	$2.623 * 10^0$

- **NOTE:** The decimal number $34.567 * 5^{12}$ is not equivalent to $3.4567E13$.
- C++ provides two data types to manipulate decimal numbers: `float` and `double`.
- The following table show the range of values and required storage space for `float`, and `double` data types.

Data Type	Values	Storage (in bytes)
float	$-3.4 * 10^{38}$ to $3.4 * 10^{38}$	4
double	$-1.7 * 10^{308}$ to $1.7 * 10^{308}$	8

- **EXAMPLE 2-20:** What is the output of following code snippet?

```
float x = 3.440000E10;
float y = 2.00000E7;
float z = x / y;
cout << z << endl;
double w = 3.154E40;
cout << w << endl;
```

OUTPUT:

```
1720
31540
```

`string` Data Type

- The data type `string` is a programmer-defined data type.
- The data type `string` is not directly available for use in a program like the simple data types discussed earlier, so you need to access its definition from the header file `string`.

```
#include <string>
```

- A string is a sequence of zero or more characters.
- Strings in C++ are enclosed in double quotation marks `" "`.
- A string containing no characters is called a null or empty string.
- **EXAMPLE 2-16:** What is the output of the following code snippet?

```
string str1, str2;
str1 = "William Jacob";
str2 = "Mickey";
const string TAB = "     ";
cout<<str1<<TAB<<str2;
```

OUTPUT:

```
William Jacob    Mickey
```

- Every character in a string has a relative position in the string.
  - The position of the first character is 0, and the position of the second character is 1, and so on.
- The length of a string is the number of characters in it including the spaces.
- **EXAMPLE 2-17:** For the string `"HELLO World!"` answer the following questions.
  - What is the length of the string? `12`
  - What is the position of character 'H'? `0`
  - What is the position of character 'L'? `2`
  - What is the position of character 'O'? `4`
  - What is the position of character 'o'? `7`
  - What is the position of character ' ' the space? `5`

- **EXAMPLE 2-18:** For the string `"It is a beautiful day."` answer the following questions.
  - What is the length of the string? `22`
  - What is the position of character 'a'? `6`

- C++ provides many operations and functions to manipulate strings.
  - To find a string length use the function `length()` as follows

```
stringId.length();
```

- To print the character located at position `p` use the square bracket operator `[ ]` as follows

  ```
  stringId[p]
  ```

- **EXAMPLE 2-19:** What is the output of the following code snippet?

  ```cpp
  string name = "Ahmad Ali";
  int len = name.length();
  cout<<name<<" has "<<len<<" characters\n";
  cout<<name[0]<<name[2]<<name[7];
  ```

  OUTPUT

  ```
  Ahmad Ali has 9 characters
  Aml
  ```

Arithmetic Operators, Operator Precedence, and Expressions

- **Arithmetic expressions:** contain values and arithmetic operators
  - **Operands:** are the values on which the operators will work
  - **Operators:** can be unary (one operand) or binary (two operands)
- C++ arithmetic operators divided in to two types:
  - **Binary operators:** which are operators with 2 operands
    - `+` addition, `-` subtraction, `*` multiplication, `/` division, `%` modulus (or remainder)
  - **Unary operators:** which are operators with one operand
    - `-` negation, `++` increment, `--` decrement

Order of Precedence

- When more than one arithmetic operator is used in an expression, C++ uses the operator precedence rules to evaluate the expression.
- All operations inside of parenthesis `()` are evaluated first
- Unary operators
- Multiplication `*`, Division `/`, and modulus `%` are at the same level of precedence and are evaluated next
- Addition `+` and Subtraction `-` have the same level of precedence and are evaluated last.
- When operators are on the same level performed from left to right (associativity).
- Study the following arithmetic expression.

```
3 * 7 - 6 + 2 * 5 / 4 + 6
// = (((3 * 7) - 6) + ((2 * 5) / 4)) + 6
// = 21 - 6 + 2 + 6 = 23
```

- There are three types of arithmetic expressions in C++:
  - Integral expressions—all operands in the expression are integers. An integral expression yields an integral result.

    ```
    3 + 5 / 2 - 1
    = 3 + 2 - 1
    = 4
    ```

  - Floating-point (decimal) expressions—all operands in the expression are decimal numbers. A floating-point expression yields a floating-point result.

    ```
    10.0 * 3.75 - 34.2
    = 37.5 - 34.2
    = 3.3
    ```

  - Mixed expressions—the expression contains both integers and decimal numbers.
- Evaluation rules for arithmetic expressions
  - If operator has same types of operands, then it is evaluated according to the type of the operands

    ```
    5 / 2 = 2
    10 / 3 = 3
    5.0 / 2.0 = 2.5
    ```

  - If operator has both types of operands then integer is changed to floating-point, Operator is evaluated, and the result is floating-point.

    ```
    5.0 / 2 = 2.5
    5 / 2.0 = 2.5
    ```

  - Entire expression is evaluated according to precedence rules.
- **EXAMPLE 2-20:** What is the output of the following code snippet?

```
int n1  = 2, n2 = 4, n3 = 6;
cout<< 3 + n1 - n2 / 7<<endl;
cout<< n1 + 2 * (n2 - n3) + 18<<endl;

double d1 = 10.0, d2 = 20.4;
cout<< d1 * 10.5 + d2 - 16.2 <<endl;

cout<< 5.4 * 2 - 12.6 / 3 + 18 / 2 <<endl;
cout<< 5.4 * 2 - 13.6 + 18 % 5<<endl;
```

OUTPUT:

```
5
16
109.2
15.6
0.2
```

- **PRACTICE:** What is the output of each of the following code snippet?

```cpp
int x1 = 10, x2 = 4;
cout<< x1 / x2 <<endl;
cout<< x1 % x2 <<endl;
cout<<x1<<x2<<endl;
```

  - **NOTE:** The value of a variable will not changed unless you use assignment operator `=` or input statement `cin >>`.

- **EXAMPLE 2-21:** Write a program to calculate number of weeks and the remaining days in 100 days

```cpp
int days = 100, weeks;
const int DAYS_WEEK = 7;

weeks = days / DAYS_WEEK;
days = days % DAYS_WEEK;
cout<<"# of weeks = "<<weeks<<endl;
cout<<"# of remaining days = "<<days<<endl;
```

- **PRACTICE:** Knowing that 1 foot has 12 inches, write a program to calculate number of feet and the remaining inches in 500 inches.
- **EXAMPLE 2-22:** What is the output of the following code snippet?

```cpp
bool b1, b2, b3;
b1 = 10;
b2 = true;
b3 = false;
cout<<(b1 + b2 + b3) * 3 % 5;
```

OUTPUT

```
1
```

- Because the char data type is also an integral data type, C++ allows you to perform arithmetic operations on char data.
- When using a `char` variable in an arithmetic expression, the equivalent ASCII code of the value stored in it will be used to evaluate the expression.

- There is a difference between the character '8' and the integer 8. The integer value of 8 is 8. The integer value of '8' is 56, which is the ASCII collating sequence of the character '8'.
- **Example 2-23:** What is the output of the following code snippet?

```
char x1 = 'a';  // ASCII code of 'a' = 97
cout<<x1 + '3'<<endl; // ASCII code of '3' = 51
cout<<x1 / 2 <<endl;
cout<<x1 % 5<<endl;
```

OUTPUT:

```
148
48
2
```

- When storing the result of arithmetic expression in a `char` variable, then the corresponding character of the result will be stored in this variable.
- **Example 2-24:** What is the output of the following code snippet?

```
int num = 50;
bool b = true;
char x1 = num * 2 - 3 + b;
cout<<x1<<" "<<num<<endl;
cout<<x1 + 5<<endl;
```

OUTPUT:

```
b 50
103
```

- All arithmetic operators can be used with integral and floating-point data types EXCEPT modulus operator which can be used ONLY with integral types.

```
double x1 = 7, x2 = 3;
cout<< x1 % x2 <<endl;  //syntax error
```

Compound Assignment Statements

- The assignment statements you have seen so far are called simple assignment statements.
- In certain cases, you can use special assignment statements called compound assignment statements to write simple assignment statements in a more concise notation.
- Corresponding to the five arithmetic operators `+`, `-`, `*`, `/`, and `%`, C++ provides five compound operators: `+=`, `-=`, `*=`, `/=`, and `%=`, respectively.
- Using the compound operator `*=`, you can rewrite the simple assignment statement:

`variable = variable * (expression);`

as:

`variable *= expression;`

- **EXAMPLE 2-25:** What is the equivalent compound assignment statements for the following assignment statements?

	simple assignment statement	Compound Assignment Statement
1.	`i = i + 5`	`i += 5`
2.	`count = count - 3;`	`count -= 3;`
3.	`amount = amount * (interest +1);`	`amount *= (interest +1);`
4.	`x = x / ( y + 5);`	`x /= y + 5;`

- **EXAMPLE 2-26:** What is the output of the following code snippet.

```
int x = 5, y = 8, z = 9;
x *= y - z % 5;
cout<<x<<" "<<y<<" "<<z;
```

OUTPUT:

```
20 8 9
```

Increment and Decrement Operators

- Increment operator `++`: increase the variable value by 1
  - **Pre-increment:**
    - The increment operator is written before the variable name e.g. `++variable`
    - The variable value is increased then used

```
int x = 5;
int y = ++x;          // x = 6, y=x=6
cout << x << y;       // 66
```

  - **Post-increment:**
    - The increment operator is written after the variable name e.g. `variable++`
    - The variable value is used then increased

```
int x = 5;
int y = x++;          // y = x = 5, x = 6
cout << x << y;       // 65
```

- Decrement operator `--`: decrease the variable value by 1

- **Pre-decrement:**
  - The decrement operator is written before the variable name e.g. `--variable`
  - The variable value is decreased then used

```
int x = 5;
int y = --x;         // x = 4, y=x=4
cout << x << y;      // 44
```

- **Post-decrement:**
  - The decrement operator is written after the variable name e.g. `variable--`
  - The variable value is used then decreased

```
int x = 5;
int y = x--;         // y = x = 5, x = 4
cout << x << y;      // 45
```

- **Example 2-27:** What is the output of the following code snippet?

```
int a = 5;
int b = 2 + (++a);   // a = 6, b=2+6=8
cout<<a<<b;
```

OUTPUT:

```
68
```

- **Example 2-28:** What is the output of the following code snippet?

```
int a = 5;
int b = 2 + (a--);   // b= 2+a= 7, a = 4
cout<<a<<b;
```

OUTPUT:

```
47
```

- **Example 2-29:** What is the output of the following code snippet?

```
int x = 10;
cout<<++x<<endl;  // x = 11, print x
cout<<x++<<endl;  // print x, x = 12
cout<<x;
```

OUTPUT:

```
12
```

Type Conversion (Casting)

- Changing the data type of the result of an expression to another data type.
- Type conversion could be:
  - **Implicit:** When the expression results data type is **automatically** changed to another type **temporarily** by the compiler.
  - **Explicit:** When the expression results data type is **manually** changed to another type **temporarily** using **cast** operator.

Implicit Type Conversion

- If you are not careful about data types, implicit type coercion can generate unexpected results.
- **RECALL**, the following is the syntax to define a variable and store the result of expression in it.

```
type variable = expression;
```

- To execute the previous statement:
  1. Evaluate the expression according to the evaluation rules we studied earlier.
  2. Modify the value of the result of expression to suit the variable's data type.
- **NOTE:** The compiler can implicitly convert values from one integral and decimal data type to another as shown in the following examples.
- **EXAMPLE 2-30:** What is the output of the following code snippet?

```
bool x1 = 1;
int x2 = 13;
double x3 = 4.0;
char x4 = 'B';

// Print the result of arithmetic expression
cout<< x2 - 13 <<endl;
cout<< x4 / x3 <<endl;
cout<< x4 + x3-x2 <<endl;
cout<< x4 % 5 <<endl;

// Store the result of arithmatic expression in a variable
x1 = x2 - 13;
x3 = x4 / x3;
x4 += (x3 - x2);
x2 = x4 % 5;

cout<<x1<<x2<<x3<<x4<<endl;
```

OUTPUT:

```
0
16.5
57
1
0416.5E
```

- **Example 2-31:** What is the output of the following code snippet?

```cpp
char v1 = 'c';              // v1 = 99
int v2 = 5 * ++v1 + 70;     // v1 = 100, v2=570
int v3 = 600 - 3 * v1--;    // v3 = 300, v1=99
cout<<v1<<v2<<v3;
```

OUTPUT:

```
c570300
```

- **PRACTICE:** What is the output of the following code snippet?

```cpp
// Remember: a boolean variable can only store values 0 or 1
bool b1 = 10 * 6 - 60;      // b1 = 0
bool b2 = 3.14;             // b2 = 1
bool b3 = 0.04;             // b3 = 1
bool b4 = 'a';              // b4 = 1
cout<<b1<<"\t"<<b2<<"\t"<<b3<<"\t"<<b4;
```

- **PRACTICE:** What is the output of the following code snippet?

```cpp
bool b = 10;        // b = 1
int v1 = 13.7;      // v1 = 13
int v2 = 0.999;     // v2 = 0
int v3 = 'a';       // v3 = 97
int v4 = b;         // v4 = 1
int v5 = false;     // v5 = 0
cout<<v1<<"\t"<<v2<<"\t"<<v3<<"\t"<<v4;
```

- **PRACTICE:** What is the output of the following code snippet?

```cpp
bool b = 0;
char c1 = 'A';      // c1 = A
char c2 = 66;       // 66 is the ASCII code of 'B' => c1 = B
char c3 = 97.65;    // 97 is the ASCII code of 'a' => c2 = a
char c4 = b;        // 0 is the ASCII code of null => c3 store null
cout<<c1<<c2<<c4<<c3;
```

- **PRACTICE:** What is the output of the following code snippet?

```cpp
bool b = 10;            // b = 1
double d1 = 4;          // d1 = 4.0
double d2 = b / 2.0;    // d2 = 0.5
double d3 = 'B';        // d3 = 66.0
double d4 = 'a';        // d4 = 97.0
cout<<d1<<d2<<d3<<d4;
```

- **NOTE:** You cannot store a string in an integral or decimal variables.

```cpp
bool b = "str";     // error: cannot convert string to bool
int v = "str";      // error: cannot convert string to int
char c = "str";     // error: cannot convert string to char
double d = "str";   // error: cannot convert string to double
string text = 4;    // error: cannot convert int to string
```

Explicit Type Conversion

- To avoid implicit type conversion, C++ provides for explicit type conversion through the use of a `cast` operator.
- The cast operator, also called type conversion or type casting, takes the following form:

```cpp
static_cast<dataTypeName>(expression)
```

  - First, the expression is evaluated.
  - Then, its value is converted to a value of the type specified by dataTypeName.
- In C++, `static_cast` is a reserved word.
- **EXAMPLE 2-32:** What is the output of each of the following code snippet?

```cpp
cout<<static_cast<int>(7.9)<<endl;
cout<<static_cast<int>(3.3)<<endl;
cout<<static_cast<double>(25)<<endl;
cout<<static_cast<double>(5 + 3)<<endl;
cout<<static_cast<double>(15) / 2<<endl;
cout<<static_cast<double>(15 / 2)<<endl;
cout<<static_cast<int>(7.8 + static_cast<double>(15) / 2)<<endl;
cout<<static_cast<int>(7.8 + static_cast<double>(15 / 2))<<endl;

char v = static_cast<int>(66.9);
int w = 'a' + static_cast<bool>(66.9);
double x = static_cast<int>(100.0) / 3;
cout<<v<<w<<x<<endl;
```

OUTPUT:

```
7
3
25
8
7.5
7
15
14
B9833
```

C++ PROGRAMMING:

FROM PROBLEM ANALYSIS TO PROGRAM DESIGN

BY: D. S. MALIK

CHAPTER 3: INPUT / OUTPUT

SUMMARY & EXAMPLES

PREPARED BY:

E. MANAR JARADAT

CHAPTER 3: INPUT/OUTPUT

- A program performs three basic operations: it gets data, it manipulates the data, and it outputs the results.
- The standard input device is usually the keyboard, and the standard output device is usually the screen.
- In C++, data is moved between computers and input/ output devices as a sequence of characters from the source to the destination called a **stream**.
- There are two types of streams:
  - **Input stream:** A sequence of characters moved from an input device to the computer.
  - **Output stream:** A sequence of characters moved from the computer to an output device.
- To receive data from the keyboard and send output to the screen, every C++ program must use the header file `iostream`.
- The header file `iostream` contains, among other things, the following:
  - The definitions of two data types, `istream` (input stream) and `ostream` (output stream).
  - The declaration of the input stream variable `cin`, which stands for common input. `istream cin`
  - The declaration of the output stream variable `cout`, which stands for common output. `ostream cout`
- To use `cin` and `cout` in your C++ program must use the preprocessor directive:

  ```
  #include <iostream>
  ```

- Because `cin` and `cout` are already defined and have specific meanings, to avoid confusion, you should never redefine them in programs.
- **EXAMPLE 3-1:** Study the following code snippet, and notice the effect of redefinition a predefined variable.

  ```
  int x = 5, cin = 10;
  cin += 15;
  cout << cin << endl;
  cin >> x;       //warning: statement has no effect
  cout << x;
  ```

  OUTPUT:

  ```
  25
  5
  ```

Input (Read) Statement

- In Chapter 2 we learned how to use assignment statement to put data into variables.

- In this chapter we will learn how to develop **interactive programs** that use input (or read) statements to put data into variables from the standard input device.
- Putting data into variables from the standard input device is accomplished via the use of `cin` and the stream extraction operator `>>`.
- The syntax of the input statement (`cin >>`):

```
cin >> variable;
```

- The extraction operator `>>` is a binary operator since it takes two operands.
    - The left-side operand must be an input stream variable, such as `cin`.
    - The right-side operand is a variable of **simple data types** (`int`, `double`, `char` ...).
- A single input statement can read more than one data item by using the extraction operator `>>` several times.

```
cin >> variable >> variable ...;
```

- Every occurrence of `>>` extracts the next data item from the input stream.
- Study the following code snippets, and notice how you can read the length and width of a rectangle using either a single input statement or multiple input statements.

```
cin >> length >> width;
```

```
cin >> length;
cin >> width;
```

- During programming execution, if more than one value is entered in a line, these values must be separated by at least one blank or tab. Alternately, one value per line can be entered.
- How does the extraction operator `>>` work?
    - When scanning for the next input, `>>` skips all whitespace characters (lines, blanks, tabs).
    - The extraction operator `>>` simply finds the next input data in the input stream.
- How does the extraction operator `>>` distinguish between the character 2 and the number 2?
    - The right-side operand of the extraction operator `>>` makes this distinction.
        - If the right-side operand is a variable of the data type `char`, the input 2 is treated as the character 2 and, in this case, the ASCII value of 2 is stored.
        - If the right-side operand is a variable of the data type `int` or `double`, the input 2 is treated as the number 2.
- During program execution, when entering character data such as letters, you do not enter the single quotes around the character.
- **Example 3-2:** Consider the following variable declarations and answer the question below.

```
int a;
double z;
char ch;
```

What is the value stored in each variable after executing the following input statements.

	Statement	Input	Value stored in Memory
1	`cin >> a >> ch >> z;`	57 A 26.9	a = 57, ch = 'A', z = 26.9
2	`cin >> a >> ch >> z;`	57 A 26.9	a = 57, ch = 'A', z = 26.9
3	`cin >> a >> ch >> z;`	57 A 26.9	a = 57, ch = 'A', z = 26.9
4	`cin >> a >> ch >> z;`	57A26.9	a = 57, ch = 'A', z = 26.9

- **NOTE:** The `char` data type takes one printable character except the blank
- **NOTE:** The integral data types takes an integer, possibly preceded by a `+` or `-` sign.
- **NOTE:** The decimal data types takes a decimal number, possibly preceded by a `+` or `-` sign. If the actual data input is an integer, then the input is converted to a decimal number with the zero decimal part.
- **Example 3-3:** Consider the following variable declarations and answer the question below.

```
int a;
double z;
char ch;
```

What is the value stored in each variable after executing the following input statements.

	Statement	Input	Value stored in Memory
1	`cin >> ch;`	A	ch = 'A'
2	`cin >> ch;`	AB	ch = 'A' 'B' is held for later input
3	`cin >> a;`	+48	a = 48
4	`cin >> a;`	-46.35	a = -46 .35 is held for later input
5	`cin >> z;`	74.35	z = 74.35
6	`cin >> z;`	39	z = 39.0
7	`cin >> z >> a;`	65.78-38	z = 65.78, a = -38

- **Example 3-4:** Consider the following variable declarations and answer the question below.

```
int a, b;
double z;
char ch, ch1, ch2;
```

What is the value stored in each variable after executing the following input statements.

	Statement	Input	Value stored in Memory
1	`cin >> z >> ch >> a;`	36.78B34	z = 36.78, ch = 'B', a = 34
2	`cin >> z >> ch >> a;`	36.78 B34	z = 36.78, ch = 'B', a = 34
3	`cin >> a >> b >> z;`	11 34	a = 11, b = 34, Computer waits for the next input
4	`cin >> a >> z;`	78.49	a = 78, z = 0.49
5	`cin >> ch >> a;`	256	ch = '2', a = 56
6	`cin >> a >> ch;`	256	a = 256, Computer waits for the next input
7	`cin >> ch1 >> ch2;`	A B	ch1 = 'A', ch2 = 'B'

- **EXAMPLE 3-5:** What is the output of the following code snippet?

```
int x, y;
double z;
cout << "Insert 2 integers and 1 decimal number"<<endl;
cin >> x >> y >> z;
cout << 4 + x / y + z;
```

SAMPLE RUN: Assume user input is 12 5 7.25

```
Insert 2 integers and 1 decimal number
12 5 7.25
13.25
```

- **EXAMPLE 3-6:** What is the output of the following code snippet?

```
int x;
double y;
char z;
cout << "Enter integer, character, and decimal number" << endl;
cin >> x >> z >> y;
z += y / x;
cout<<z;
```

SAMPLE RUN: Assume user input is 12B27.5

```
Enter integer, character, and decimal number
12B27.5
D
```

- **EXAMPLE 3-7:** Modify example 2-21 from chapter 2 to read the value of days from user.

```cpp
int days, weeks;
const int DAYS_WEEK = 7;

cout<<"Enter number of days: "<<endl;// prompt statement
cin >> days;

weeks = days / DAYS_WEEK;
days = days % DAYS_WEEK;

cout<<"# of weeks = "<<weeks<<endl;
cout<<"# of remaining days = "<<days<<endl;
```

SAMPLE RUN: Assume user input is 500

```
Enter number of days:
500
# of weeks = 71
# of remaining days = 3
```

- When you enter data for processing, the data values should correspond to the data types of the variables in the input statement. because computers does not tolerate any other kind of mismatch.
- For example, entering a char value into an `int` or `double` variable causes serious errors, called input failure.
- What happens if the input stream has more data items than required by the program?
  - After the program terminates, any values left in the input stream are discarded.

Input Failure

- What would happen if you tried to input a letter into an `int` variable? If the input data did not match the corresponding variables, the program would run into problems.
- For example, trying to read a letter into an `int` or `double` variable would result in an input failure.
- **EXAMPLE 3-8:** Consider the following variable definitions, and answer the question below.

```cpp
int a, b, c;
double x;
```

What will be the value stored in each variable after executing the following input statements given the corresponding input.

Statement	Input	result
`cin >> a >> b;`	35 67.93 48	a = 35, b = 67

Statement	Input	result
`cin >> a >> b;`	W 54	Input failure, input the character 'W' into the `int` variable a.
`cin >> a >> b >> c;`	35 67.93 48	a = 35, b = 67, Input failure, input the character '.' into the `int` variable c.

- Input failure causes the input stream to enter a fail state.
- What happens happens when the input stream enters the fail state?
  - All further I/O statements using that stream are ignored.
  - The program continues to execute with whatever values are stored in variables and produces incorrect results.
- **EXAMPLE 3-9:** What is the output of the following code segment for each of the given user inputs

```cpp
int main()
{
    string name;
    int age = 0;
    int weight = 0;
    double height = 0.0;

    cout << "Enter name, age, weight, and height: ";
    cin >> name >> age >> weight >> height;//Sam 35.0 156 6.2

    cout << "Name: " << name << endl;
    cout << "Age: " << age << endl;
    cout << "Weight: " << weight << endl;
    cout << "Height: " << height << endl;
}
```

SAMPLE RUN1: Assume the user input is Sam 35 q56 6.2

```
Enter name, age, weight, and height: Sam 35 q56 6.2
Name: Sam
Age: 35
Weight: 0
Height: 0
```

Sample Run2: Assume the user input is Sam 35.0 156 6.2

```
Enter name, age, weight, and height: Sam 35.0 156 6.2
Name: Sam
Age: 35
Weight: 0
Height: 0
```

Input the string Type

- You can use the input statement `cin >>`, to read a string into a variable of the data type `string`.
- **NOTE:** You cannot use the extraction operator to read strings that contain blanks.
- **EXAMPLE 3-10:** Consider the following variable declarations and answer the question below.

```
int a;
double d;
char ch;
string str;
```

What is the value stored in each variable after executing the following input statements?

	Statement	Input	Value stored in Memory

1	`cin >> a >> d >> ch >> str;`	5 4 % txt	a = 5 , d = 4.0
			ch = '%', str = txt2

2	`cin >> a >> d >> ch >> str;`	5.43Programming	a = 5 , d = 0.43
			ch = 'P', str = rogramming

3	`cin >> a >> d >> ch >> str;`	5.4 3Programming	a = 5 , d = 0.4
			ch = '3', str = Programming

- **EXAMPLE 3-11:** What is the output of the following code snippet?

```
string name;
cout <<"What is your name? "<<endl;
cin >> name;
cout <<"Welcome "<< name;
```

SAMPLE RUN: Assume user input is Manar Jaradat

```
What is your name?
Manar Jaradat
Welcome Manar
```

- To read a string containing blanks, you can use the function `getline`.
- The syntax to use the function `getline` is:

```
getline(istreamVar, strVar);
```

  - `istreamVar` is an input stream variable, and `strVar` is a string variable.
- The function `getline` reads until it reaches the end of the current line `'\n'`. The newline character is also read but not stored in the string variable.
- **EXAMPLE 3-12:** What is the output of the following code snippet?

```
string name;
cout <<"What is your name? "<<endl;
getline(cin, name);
cout <<"Welcome "<< name;
```

SAMPLE RUN: Assume user input is Manar Jaradat

```
What is your name?
Manar Jaradat
Welcome Manar Jaradat
```

Using Predefined Functions in a Program

- C++ comes with a wealth of functions, called predefined functions, that are already written.
- **RECALL:** In Chapter 2 we learned that predefined functions are organized as a collection of libraries, called header files.
- To use a predefined function in a program, you need to know the following:
  - You must be aware of what the function is going to do.
  - The name of the header file containing the specification of the function and include that header file in the program.
  - The name of the function.
  - The number of parameters the function takes, and the type of each parameter.
- Consider the use of the power function `pow`
  - The specification of the function `pow` is contained in the header file `cmath` .
  - The function pow has two parameters, which are decimal numbers.
  - It is used to calculate the first parameter to the power of the second parameter. pow(x, y) = x^y.

- Consider the use of the power function `sqrt`
  - The specification of the function `sqrt` is contained in the header file `cmath` .
  - The function sqrt has a single parameter, which is a decimal number.
  - It is used to calculate the square root of the parameter. sqrt(x) = $\sqrt{x}$.

- **EXAMPLE 3-13:** Write the equivalent C++ statement for the following mathematical equations.

	Mathematical Equation	Equivalent C++ Statement
1.	$x = 2.0^{3.0}$	`double x = pow(2.0, 3.0)`
2.	$y = \sqrt{4}$	`double y = pow(4, 0.5)` **OR** `double y = sqrt(4)`
3.	$z = x^y$	`double z = pow(x, y)`

	Mathematical Equation	Equivalent C++ Statement
4.	$w = 34.15 * 10^9$	`double w = 34.15 * pow(10, 9)` **OR** `double w = 34.15000E9`
5.	$v = 10.9 * 5^{-12}$	`double v = 10.9 * pow(5, -12)`

- **EXAMPLE 3-14:** Write the required C++ program to calculate and print the volume of a cylinder, read the values of radius and height from the user.

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main ()
{
    double radius;
    double height;
    const double PI = 3.14;
    cout << "Insert the radius and height of the cylinder" << endl;
    cin >> radius >> height;

    double volume = PI * pow (radius, 2) * height;
    cout << "A cylinder of radius = " << radius;
    cout << " and height = " << height;
    cout << " has volume = " << volume << endl;
}
```

SAMPLE RUN: Assume user input is: 2 10

```
Insert the radius and height of the cylinder
2 10
A cylinder of radius = 2 and height = 10 has volume = 125.6
```

- **EXAMPLE 3-15:** Knowing that a point `p1` has the coordinates (x1,y1), and another point `p2` has the coordinates (x2,y2). Write the required code to read the coordinates from user and calculate the distance between them.

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main ()
{
    double x1, y1;
    double x2, y2;
    cout << "Enter the coordinates of the first point:" << endl;
    cin >> x1 >> y1;
    cout << "Enter the coordinates of the second point:" << endl;
    cin >> x2 >> y2;

    double tmp = pow (x2 - x1, 2) + pow (y2 - y1, 2);
    double distance;

    distance = sqrt (tmp);
    cout << "distance between points = " << distance;
}
```

SAMPLE RUN: Assume user input is: 4 7 9 -5

```
Enter the coordinates of the first point:
4 7
Enter the coordinates of the second point:
9 -5
distance between points = 13
```

C++ PROGRAMMING:

FROM PROBLEM ANALYSIS TO PROGRAM DESIGN

BY: D. S. MALIK

CHAPTER 4: CONTROL STRUCTURES I (SELECTION)

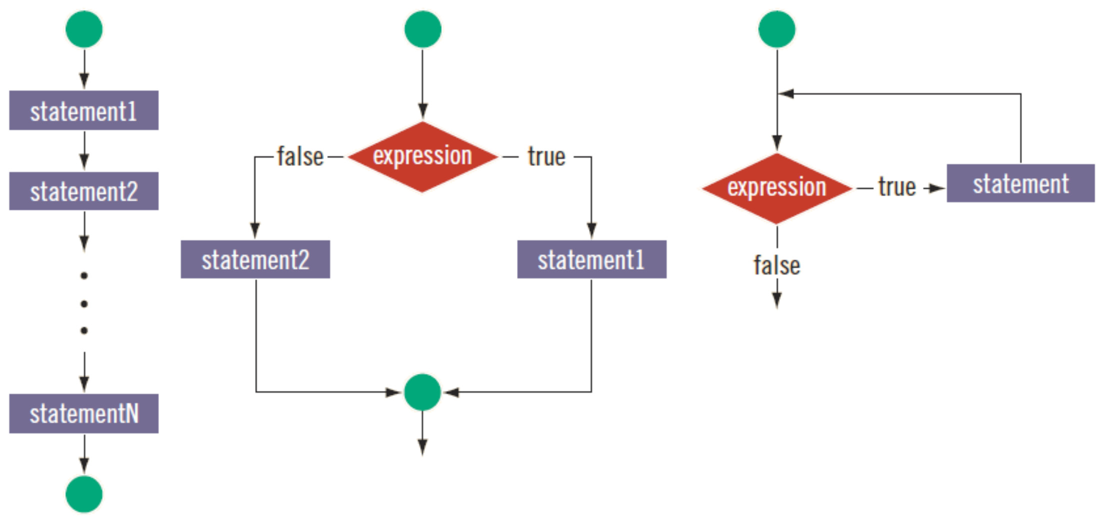
SUMMARY & EXAMPLES

PREPARED BY:

E. MANAR JARADAT

CHAPTER 4: CONTROL STRUCTURES I (SELECTION)

- The programming examples we studied so far included simple **sequential programs**. In sequential programs, the computer starts at the beginning and follows the statements in order.
- **Control structures** provide alternatives to sequential program execution and used to alter the sequential flow of execution.
- The two most common control structures are :
    - **Selection:** The program executes particular statements depending on some condition (branches).
    - **Repetition:** The program repeats particular statements a certain number of times based on some condition(s).
- Another popular control structure is **function call**, and we will study it later in chapter 6.
- The figure below illustrates the first three types of program flow.



a. Sequence b. Selection c. Repetition

- Both selection statements and repetition statements use conditional statements to decide whether to execute a set of statements or not.
- Consider the following three conditional statements:

1.
```
if (score is greater than or equal to 90) then grade is A
```

2.
```
if (hours worked are less than or equal to 40)
    wages = rate * hours
otherwise
    wages = (rate * 40) + 1.5 *(rate *(hours - 40))
```

3.
```
if (temperature is greater than 70 degrees and it is not raining)
    Go golfing!
```

Relational Operators

- To make decisions in C++, you must be able to express conditions and make comparisons.
- In C++, a condition is represented by a logical (Boolean) expression that has a value of either `true` or `false`.
- In C++, comparisons are done using **relational operators**.
- **Relational operators** are binary operators; that is, they requires two operands.
- Expressions that use the relational operators evaluate to either `true` or `false`.
- The following table lists the relational operators that allow you to state conditions and make comparisons.

Operator	Description
==	equal to
!=	not equal to
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

- In C++, the symbol `==`, is called the **equality operator**, and it checks whether two expressions are equal or not.
- In C++, the symbol `=` is called the **assignment operator**, and it assigns the value of an expression to a variable.

Relational Operators and Simple Data Types

- You can use the relational operators to compare values from all three simple data types (integral, decimal)
- **EXAMPLE 4-1:** Consider the following relational expressions.

Expression	Meaning	Value
$8 < 15$	is 8 less than 15?	true
$6! = 6$	is 6 not equal to 6?	false
$2.5 > 5.8$	is 2.5 greater than 5.8?	false
$5.9 <= 7.5$	is 5.9 less than or equal to 7.5?	true

- For `char` values, the relational operators evaluates to true or false depends on their ASCII collating sequence
- **EXAMPLE 4-2:** Consider the following relational expressions.

Expression	Meaning	Value
`' ' < 'a'`	is ASCII value of `' '` (32) less than the ASCII value of `'a'` (97)?	true
`'R' > 'T'`	is ASCII value of `'R'` (82) less than the ASCII value of `'T'` (84)?	false
`'+' < '*'`	is ASCII value of `'+'` (43) less than the ASCII value of `'*'` (42)?	false
`'A' <= 'a'`	is ASCII value of `'A'` (65) less than or equal the ASCII value of `'a'` (97)?	true

- **RECALL:** Operands of the relational operators could be of any simple data type.
- **EXAMPLE 4-3:** What is the output of the following code snippet?

```cpp
// ASCII value of '2' = 50
bool b;
int x = 50;
double y = 7.75;
char three = '3', five = '5';

cout << (three < 50) << endl;
cout << (five >= 53) << endl;
cout << (x <= y) << endl;
b = x > y;
cout << (1 > b) << endl;
cout << (x == '2') << endl;
```

OUTPUT:

```
0
1
0
0
1
```

- Relational operators have left to right associativity. that is, when two operators of same precedence are adjacent, the left most operator is evaluated first.
- **EXAMPLE 4-4:** Assume `num` is an `int` variable. Study the following statement and answer the questions below?

```cpp
cout<< (0 <= num <= 10);
```

  - What is the output if `num = 5`?

```
// ((0 <= 5) <= 10)
// (1 <= 10)
1
```

- What is the output if `num = -3`?

```
// ((0 <= -3) <= 10)
// (0 <= 10)
1
```

- **PRACTICE:** What is the output if `num = 13`?

Relational Operators and the string Type

- How to apply the relational operators on variables of type string.
  - Variables of type string are compared character by character, starting with the first character and using the ASCII collating sequence.
  - The character-by-character comparison continues until:
    - A mismatch is found
    - The last characters have been compared and are equal.

- **Example 4-5:** Consider the following variable declarations and evaluate the expressions in the table below.

```
string str1 = "Hello";
string str2 = "Hi";
string str3 = "Air";
string str4 = "Bill";
string str5 = "Big";
```

	Expression	Value /Explanation
1	`str1 < str2`	**true** The first characters of str1 and str2 are the same. The second character 'e' of str1 is less than the second character 'i' of str2. Therefore, str1 < str2 is true.
2	`str1 > "Hen"`	**false** The first two characters of str1 and "Hen" are the same. The third character 'l' of str1 is less than the third character 'n' of "Hen". Therefore, str1 > "Hen" is false.

	Expression	Value /Explanation
3	`str3 < "An"`	**true** The first characters of str3 and "An" are the same. The second character 'i' of "Air" is less than the second character 'n' of "An". Therefore, str3 < "An" is true.
4	`str1 == "hello"`	**false** The first character 'H' of str1 is less than the first character 'h' of "hello", because the ASCII value of 'H' is 72, and the ASCII value of 'h' is 104. Therefore, str1 == "hello" is false.
5	`str3 <= str4`	**true** The first character 'A' of str3 is less than the first character 'B' of str4. Therefore, str3 <= str4 is true.
6	`str2 > str4`	**true** The first character 'H' of str2 is greater than the first character 'B' of str4. Therefore, str2 > str4 is true.
7	`str4 >= "Billy"`	**false** All four characters of str4 are the same as the corresponding first four characters of "Billy". "Billy" is the larger string. Therefore, str4 >= "Billy" is false.
8	`str5 <= "Bigger"`	**true** All three characters of str5 are the same as the corresponding first three characters of "Bigger", "Bigger" is the larger string. Therefore, str5 <= "Bigger" is true.

- You cannot use the relational operator to compare a string type with any of the simple data types.

```
string str = "hi";
int x = 5;
bool b = str >= x;   //error: no match for ‘operator>=’
                     //(operand types are ‘std::string’ and ‘int’)
```

Logical (Boolean) Operators and Logical Expressions

Logical (Boolean) Operators

- Logical operators take only logical values as operands and yield only logical values as results.
- Logical (Boolean) operators enable you to combine logical expressions.
- C++ has three logical (Boolean) operators listed in the following table.

Operator	Description
!	not
&&	and
||	or

- The operators `&&`, and `||` are binary operators, while the operator `!` is a unary operator.

Not (`!`) Operator

- Putting `!` in front of a logical expression reverses the value of that logical expression.
  - `!true` is false.
  - `!false` is true.
- The following table defines the operator `!` (not).

Expression	!(Expression)
`true` (nonzero)	`false` (0)
`false` (0)	`true` (1)

- **EXAMPLE 4-6:** What is the output of the following code snippet?

```
cout<<!('A' > 'B');
cout<<!(6 < 7);
```

OUTPUT:

```
10
```

- **THINK 🤔: ** Is it correct to write not operator after the expression `(Expression)!` ?

And (`&&`) Operator

- `Expression1 && Expression2` is true if and only if both Expression1 and Expression2 are true; otherwise, Expression1 && Expression2 evaluates to false.
- The following table defines the operator `&&` (and).

Expression1	Expression2	Expression1 && Expression2
`true` (nonzero)	`true` (nonzero)	`true` (1)
`true` (nonzero)	`false` (0)	`false` (0)
`false` (0)	`true` (nonzero)	`false` (0)
`false` (0)	`false` (0)	`false` (0)

- **EXAMPLE 4-7:** What is the output of the following code snippet?

```
cout<<(14 >= 5) && ('A' < 'B');
cout<<(24 >= 35) && ('A' < 'B');
```

OUTPUT:

```
10
```

- **EXAMPLE 4-8:** Write the required expression to check if an integer number `num` is between 20 and 100 (inclusive).

```
(20 <= num) && (num <= 100)
```

- **EXAMPLE 4-9:** Write the required expression to check if a character `ch` is a capital letter.

```
(ch >= 'A') && (ch <= 'Z')
```

- **EXAMPLE 4-10:** Write the required expression to check if a character `ch` is a NOT a capital letter.

```
!((ch >= 'A') && (ch <= 'Z'))
```

Or (`||`) Operator

- `Expression1 || Expression2` is true if and only if at least one of the expressions is true; otherwise, `Expression1 || Expression2` evaluates to false.
- The following table defines the operator `||` (or).

Expression1	Expression2	Expression1 || Expression2
`true` (nonzero)	`true` (nonzero)	`true` (1)
`true` (nonzero)	`false` (0)	`true` (1)
`false` (0)	`true` (nonzero)	`true` (1)
`false` (0)	`false` (0)	`false` (0)

- **EXAMPLE 4-11:** What is the output of the following code segments.

```
cout << (14 >= 5) || ('A' > 'B');
cout << (24 >= 35) || ('A' > 'B');
cout << ('A' <= 'a') || (7 != 7);
```

OUTPUT:

```
101
```

- **EXAMPLE 4-12:** Write the required expression to check if a character `ch` is a NOT a capital letter.

```
(ch < 'A') || (ch > 'Z')
```

- **PRACTICE:** Solve examples 4-8 and 4-9 using the or (`||`) and not (`!`) operators.

Order of Precedence

- To work with complex logical expressions, there must be some priority scheme for evaluating operators.
- The following table shows the order of precedence of some C++ operators, including the arithmetic, relational, and logical operators.

Operators	Precedence
`!`, `+` (plus), `-` (minus), `++`, `--` (unary operators)	first
`*`, `/`, `%`	second
`+`, `-`	third

Operators	Precedence
`<`, `<=`, `>=`, `>`	fourth
`==`, `!=`	fifth
`&&`	sixth
`||`	seventh
`=` (assignment operator)	last

- Using the precedence rules in an expression, relational and logical operators are evaluated from left to right (left to right associativity).
- At any time you can also use parentheses to override the precedence of operators.
- **NOTE:** If a logical expression evaluates to true, the corresponding output is 1; if the logical expression evaluates to false, the corresponding output is 0.
- **EXAMPLE 4-13:** What is the output of the following code snippet?

```
cout << (5 + 3 <= 9 && 2 > 3);
cout << (11 > 5 || 6 < 15 && 7 >= 8);
cout << (2 + 6 * 2 <= 12 == 30 > 4 * 10);
```

OUTPUT:

```
011
```

- **EXAMPLE 4-14:** What is the output of the following code snippet?

```
bool found = true;
int age = 20;
double hours = 45.30;
double overTime = 15.00;
int count = 20;

cout<<!age<<endl;
cout<<(!found && (age >= 18))<<endl;
cout<<(!(found && (age >= 18)))<<endl;
cout<<(hours + overTime <= 75.00)<<endl;
cout<<((age >= 0) && (++count > 20))<<endl;
```

OUTPUT:

```
0
0
0
1
1
```

Short-Circuit Evaluation

- In C++, **short-circuit evaluation** is an algorithm used by compilers to evaluate logical expressions efficiently by avoiding unnecessary calculations.
- **RECALL:** Logical expressions are evaluated from left to right.
- **RECALL:** `Expression1 && Expression2` is false if at least of of the expressions is false.
- **RECALL:** `Expression1 || Expression2` is true if at least of of the expressions is true.
- In **short-circuit evaluation**, as soon as the value of the entire logical expression is known, the evaluation stops.
- **EXAMPLE 4-16:** What is the output of the following code snippet?

  ```
  char grade = 'B';
  cout << (grade == 'A') && (grade >= 7);
  ```

  OUTPUT:

  ```
  0
  ```

- **EXAMPLE 4-17:** What is the output of the following code snippet?

  ```
  int x = 4, y = 2;
  cout << ((x > y) || (x == 5)) << endl;
  cout << ((x > y) || (++x == 5)) << endl;
  cout << x << endl;
  cout << ((++x == 5) || (x > y) ) << endl;
  cout << x << endl;
  ```

  OUTPUT:

  ```
  1
  1
  4
  1
  5
  ```

Selection: `if` and `if...else`

- Selection structures incorporate decision making to alter the processing flow of a program.
- In C++, there are two selections, or branch control structures: `if` statements and the `switch` structure.
- The selection statements `if` and `if. . .else` can be used to create one of the following:
  - One-way selection.
  - Two-way selection.
  - Multiple selections.

One-Way Selection

- In C++, one-way selections are incorporated using the `if` statement.
- The syntax of one-way selection is:

```
if (expression)
    statement
```

- The elements of this syntax.
    - It begins with the reserved word `if`,
    - The expression is usually a logical expression, and it is sometimes called a **decision maker** because it decides whether to execute the statement that follows it or not.
    - The expression contained within parentheses `( )`, and they are part of the syntax.
    - The statement following the expression is sometimes called the **action statement**. It is executed only if the value of the expression is true, otherwise, the statement does not execute and the computer goes on to the next statement in the program.
- The figure below shows the flow of execution of the if statement (one-way selection).

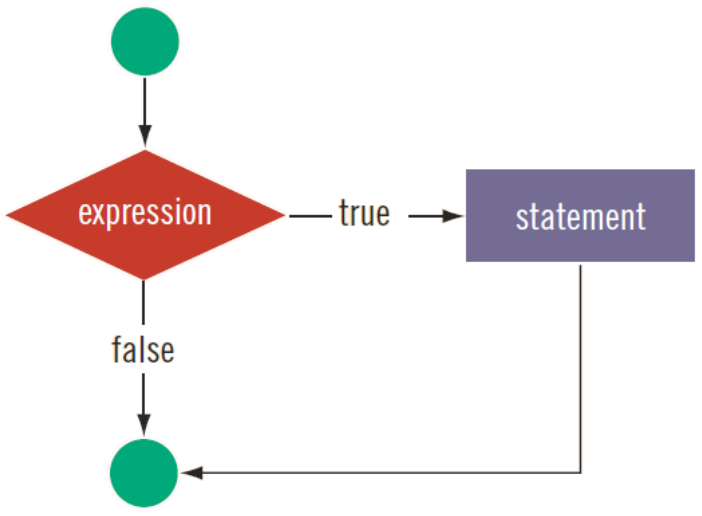


FIGURE 4-2 One-way selection

- **EXAMPLE 4-18:** Study the following code snippet and answer the questions below.

```
int x;
cout << "Enter an integer value..."<<endl;
cin >> x;
if (x > 0)
    cout << x <<" is a positive number";
```

    - Identify the elements of the used one-way selection structure
    - What is the output if user input is **6**?

```
Enter an integer value...
6
6 is a positive number
```

- What is the output if user input is **-5**?

```
Enter an integer value...
-5
```

- **RECALL:** Parentheses `( )` are part of the syntax of `if` statement, and forgetting them will cause syntax error.

```
if expression       //error
    statement
```

- **EXAMPLE 4-19:** Study the following code snippet, and find out the line that will cause a syntax error.

```
int x;
cout << "Enter an integer value..."<<endl;
cin >> x;
if x > 0
    cout << x <<" is a positive number";
```

  - Line 4: error: expected ‘(’ before 'x'

- Putting a semicolon after the parentheses following the expression in an if statement (that is, before the action statement) is a semantic error.

```
if (expression);
    statement
```

  - The semicolon will terminate the `if` statement.
  - The action of this if statement is null (empty), and the statement in line 2 is not part of the `if` statement any more.
  - The statement in line 2 executes regardless of how the if statement evaluates.
- **EXAMPLE 4-20:** Study the following code snippet and answer the questions below:

```
int x;
cout << "Enter an integer value..."<<endl;
cin >> x;
if (x > 0);
    cout << x <<" is a positive number";
```

  - What is the output if user input is **6**?

```
Enter an integer value...
6
6 is a positive number
```

- What is the output if user input is **-5**?

  ```
  Enter an integer value...
  -5
  -5 is a positive number
  ```

- **EXAMPLE 4-21:** Study the following code snippet and answer the questions below.

```cpp
int x;
cout << "Enter an integer value..."<<endl;
cin >> x;
if(++x >= 5)
    x += 2;
cout << "x = " << x;
```

  - What is the output if user input is **4**?

  ```
  Enter an integer value...
  4
  x = 7
  ```

  - What is the output if user input is **-6**?

  ```
  Enter an integer value...
  -6
  x = -5
  ```

- **EXAMPLE 4-22:** Study the following code snippet and answer the questions below.

```cpp
int x;
cout << "Enter an integer value..."<<endl;
cin >> x;
if(++x >= 5);
    x += 2;
cout << "x = " << x;
```

  - What is the output if user input is **4**?

  ```
  Enter an integer value...
  4
  x = 7
  ```

  - What is the output if user input is **-6**?

  ```
  Enter an integer value...
  -6
  x = -3
  ```

- **EXAMPLE 4-23:** Write a program to read a student name, gender and score in computer programming course, if his score is greater than or equals 50, print "Well done NAME you've scored SCORE".

```
string name;
int score;
char gender;
cout << "Enter your name, gender (F, M) and score..."<<endl;
cin >> name >> gender >> score;
if(score >= 50)
    cout << "Well done "<<name<<" you\'ve scored "<<score;
```

SAMPLE RUN:

```
Enter your name, gender (F, M) and score...
Rana F 51
Well done Rana you've scored 51
```

- **EXAMPLE 4-24:** Modify the previous program so that it adds 5 marks as bonus for female students before it check if the student pass the course.

```
string name;
int score;
char gender;
cout << "Enter your name, gender (F, M) and score..."<<endl;
cin >> name >> gender >> score;
if (gender == 'F')
score += 5;
if(score >= 50)
    cout << "Well done "<<name<<" you\'ve scored "<<score;
```

SAMPLE RUN:

```
Enter your name, gender (F, M) and score...
Rana F 48
Well done Rana you've scored 53
```

- **EXAMPLE 4-25:** Modify the previous program to accept the characters (`f` and `F`) to indicate that the student gender is female.

```
string name;
int score;
char gender;
cout << "Enter your name, gender (F, M) and score..."<<endl;
cin >> name >> gender >> score;
if (gender == 'F' || gender == 'f')
    score += 5;
if(score >= 50)
    cout << "Well done "<<name<<" you\'ve scored "<<score;
```

SAMPLE RUN:

```
Enter your name, gender (F, M) and score...
Rana f 48
Well done Rana you've scored 53
```

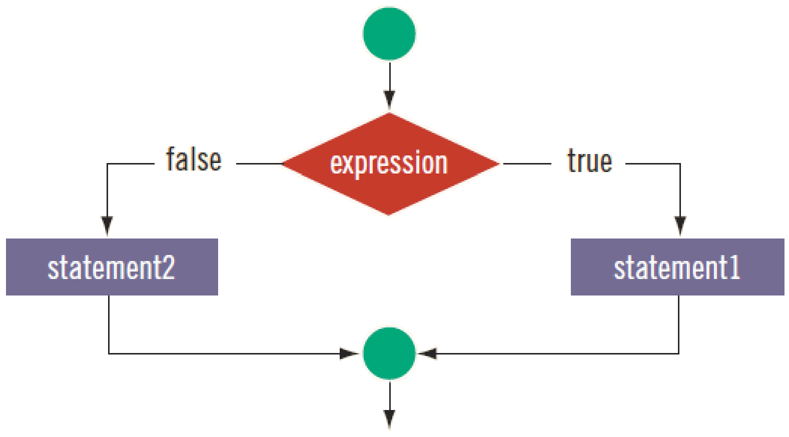
- **PRACTICE:** Write a program to compute and output the sales tax and the final price of an item sold in a particular state. Knowing that, the sales tax is calculated as follows: The state's portion of the sales tax is 4%, and the city's portion of the sales tax is 1.5%. If the item is a luxury item (items with price over $10,000), then there is a 10% luxury tax. The final price of the item is the selling price in addition to the sales tax.

Two-Way Selection

- Two-way selections are used in situations in which you must choose between two alternatives.
    - For example, if a student passed the exam or not.
- To implement two-way selections—C++ provides the `if. . .else` statement.
- Two-way selection uses the following syntax:

```
if (expression)
    statement1
else
    statement2
```

- The elements of this syntax:
    - In C++, `if` and `else` are reserved words.
    - Statements 1 and 2 are any valid C++ statements.
    - In a two-way selection, if the value of the expression is true, statement1 executes, otherwise, statement2 executes.
- The figure below shows the flow of execution of the `if. . .else` statement (two-way selection).



Two-way selection

- **EXAMPLE 4-26:** Study the following code snippet and answer the questions below:

```cpp
int x;
cout << "Enter an integer value..."<<endl;
cin >> x;
if (x >= 0)
    cout << x <<" is a positive number";
else
    cout << x <<" is a negative number";
```

- What is the output if user input is **6**?

  ```
  Enter an integer value...
  6
  6 is a positive number
  ```

- What is the output if user input is **-5**?

  ```
  Enter an integer value...
  -5
  -5 is a negative number
  ```

- **EXAMPLE 4-27:** Study the following code snippet and answer the questions below:

```cpp
int x;
cout << "Enter a positive integer value..."<<endl;
cin >> x;
if (0<= x <= 10)
    cout << x <<" is less than 10";
else
    cout << x <<" is greater than 10";
```

- What is the output if user input is **6**?

  ```
  Enter a positive integer value...
  6
  6 is less than 10
  ```

- What is the output if user input is **15**?

  ```
  Enter a positive integer value...
  15
  15 is less than 10
  ```

- **PRACTICE:** What is the output if user input is **-5**?

- In `if. . .else` statement, if the `if` statement ends with a semicolon, statement1 is no longer part of the `if` statement, and the `else` part stands all by itself.
- In C++, There is no stand-alone `else` statement. That is, it cannot be separated from the if statement.

- In a two-way selection statement, putting a semicolon after the expression separates the `else` statement from the `if` statement and this will cause a syntax error at the `else` statement.
- **EXAMPLE 4-28:** Study the following code snippet, and find out the line that will cause a syntax error.

```
1  int x;
2  cout << "Enter an integer value..."<<endl;
3  cin >> x;
4  if (x >= 0);
5      cout << x <<" is a positive number";
6  else
7      cout << x <<" is a negative number";
```

  - Line 6: error: 'else' without a previous 'if'

- **NOTE:** In a one-way selection, the semicolon at the end of an if statement is a logical error, whereas, in a two-way selection, it will cause a syntax error.
- **EXAMPLE 4-29:** Modify example 4-25 , so that it prints "Sorry NAME you've scored SCORE" if the student score is less than 50.

```
string name;
int score;
char gender;
cout << "Enter your name, gender and score..."<<endl;
cin >> name >> gender >> score;
if (gender == 'F' || gender == 'f')
    score += 5;

if(score >= 50)
    cout << "Well done "<<name;
else
    cout << "Sorry "<<name;
cout<<" you\'ve scored "<<score;
```

SAMPLE RUN:

```
Enter your name, gender and score...
Rana f 35
Sorry Rana you've scored 40
```

Compound (Block of) Statements

- **EXAMPLE 4-30:** Study the following code snippet, and find out the line that will cause a syntax error.

```
1  int x;
2  cout << "Enter an integer value..."<<endl;
3  cin >> x;
4  if (x >= 0)
5      x *= 10;
6      cout << x <<" is a positive number";
7  else
8      cout << x <<" is a negative number";
```

- Line 6: error: 'else' without a previous 'if'

- **RECALL:** The if and if. . .else structures control only one statement at a time.
- To permit more complex statements, C++ provides a structure called a compound statement or a block of statements.
- A **compound statement** are a sequence of statements enclosed in curly braces `{` and `}`, and it takes the following form:

```
{
    statement_1
    statement_2
    .
    .
    .
    statement_n
}
```

- **EXAMPLE 4-30:** Study the following code segment and answer the questions below?

```
int age;
cout<<"Enter your age... ";
cin >> age;

if (age >= 18)
{
    cout << "Eligible to vote." << endl;
    cout << "No longer a minor." << endl;
}
else
{
    cout << "Not eligible to vote." << endl;
    cout << "Still a minor." << endl;
}
```

- What is the output if user input is **22**

```
Enter your age... 22
Eligible to vote.
No longer a minor.
```

- What is the output if user input is **16**

  ```
  Enter your age... 16
  Not eligible to vote.
  Still a minor.
  ```

- The compound statement is very useful and will be used in most of the structured statements in this chapter.
- **EXAMPLE 4-31:** Modify example 4-29 , so that it checks if the entered score is between 35 and 100 (inclusive). If the entered score is less than 35 it should print an error message and set the score to 35. If the entered score is greater than 100 it should print an error message and set the score to 100.

```
string name;
int score;
char gender;
cout << "Enter your name, gender and score..."<<endl;
cin >> name >> gender >> score;

if(score < 35)
{
    cout<<"Invalid score value"<<endl;
    score = 35;
}
if(score > 100)
{
    cout<<"Invalid score value"<<endl;
    score = 100;
}

if (gender == 'F' || gender == 'f')
    score += 5;

if(score >= 50)
    cout << "Well done "<<name;
else
    cout << "Sorry "<<name;
cout<<" you\'ve scored "<<score;
```

SAMPLE RUN:

```
Enter your name, gender and score...
Rana f 30
Invalid score value
Sorry Rana you've scored 40
```

Multiple Selections: Nested if

- Multiple selection statement is used to solve problems of more than two alternatives.
- You can include multiple selection paths in a program by using an `if. . .else` structure and the action statement itself is an `if` or `if. . .else` statement.
- When one control statement is located within another, it is said to be nested.

```
if (expression)
    statement;
else if (expression)
    statement;
```

```
if (expression)
    statement;
else
    if (expression)
        statement;
    else
          statement;
```

- **RECALL:** In C++,there is no stand-alone `else` statement. Every `else` must be paired with an `if`.
- In a nested if statement,C++ associates an `else` with the most recent incomplete `if`—that is, the most recent `if` that has not been paired with an `else`.
- **EXAMPLE 4-32:** Assume **x** and **y** are `int` variables. Study the following code segment and answer the questions below.

```
if (x > y)
    x += 3;
else if(x < y)
    y += 3;
else
    x = y = y + 3;
```

  - What are the values of **x** and **y** if initial values of **x** and **y** are 6 and 3 respectively?

    x = 9, y = 3
  - What are the values of **x** and **y** if initial values of **x** and **y** are 4 and 5 respectively?

    x = 4, y = 8
  - What are the values of **x** and **y** if initial values of **x** and **y** are 5 and 5 respectively?

    x = 8, y = 8

- **EXAMPLE 4-33:** Assume **balance** and **interestRate** are two `double` variables. Study the following code segment and answer the questions below.

```
if (balance > 50000.00)
    interestRate = 0.07;
else if (balance >= 25000.00)
    interestRate = 0.05;
else if (balance >= 1000.00)
    interestRate = 0.03;
else
    interestRate = 0.00;
```

- What is the value of **interestRate** if **balance** equals 20000?

  0.03
- What is the value of **interestRate** if **balance** equals 30000?

  0.05
- What is the value of **interestRate** if **balance** equals 100000?

  0.07
- What is the value of **interestRate** if **balance** equals 500?

  0.00

- **EXAMPLE 4-34:** Assume **temperature** is an `int` variable. Study the following code segment and answer the questions below.

```
if (temperature >= 50)
    if (temperature >= 80)
        cout << "Good day for swimming." << endl;
    else
        cout << "Good day for golfing." << endl;
else
    cout << "Good day to play tennis." << endl;
```

- What is the output if the **temperature** value is **40**?

  Good day to play tennis.
- What is the output if the **temperature** value is **70**?

  Good day for golfing.
- What is the output if the **temperature** value is **90**?

  Good day for swimming.

- **EXAMPLE 4-35:** Assume that all variables are properly declared. Study the following code segment and answer the questions below.

```
if (gender == 'M')
    if (age < 21 )
        policyRate = 0.05;
    else
        policyRate = 0.035;
else if (gender == 'F')
    if (age < 21 )
        policyRate = 0.04;
    else
        policyRate = 0.03;
```

- What will be the value stored in the variable `policyRate` if the customer is a 24 years old female? 0.03
- What will be the value stored in the variable `policyRate` if the customer is a 19 years old female? 0.04
- What will be the value stored in the variable `policyRate` if the customer is a 24 years old male? 0.035
- What will be the value stored in the variable `policyRate` if the customer is a 19 years old male? 0.05
- **PRACTICE:** Write the previous program using one-way selection.

- **EXAMPLE 4-36:** Study the following code segment and answer the questions below.

```
int num;
cout << "Enter an integer number... "<<endl;
cin >> num;
if (num >= 80)
    num++;
if (num >= 70)
    num++;
if (num >= 60)
    num++;
else
    num++;
```

- What is the value stored in the variable `num` if the user input is **55**?

  56
- What is the value stored in the variable `num` if the user input is **77**?

  79
- What is the value stored in the variable `num` if the user input is **95**?

  98

Comparing if...else Statements with a Series of if Statements

- **EXAMPLE 4-37:** Given a student test score (out of 100), write a program to calculate his/her grade (A, B, C, D, F). The grade is assigned as follows: If the average test score is greater than or equal to 90, the grade is A; if the average test score is greater than or equal to 80 and less than 90, the grade is B; if the average test score is greater than or equal to 70 and less than 80, the grade is C; if the average test score is greater than or equal to 60 and less than 70, the grade is D; otherwise, the grade is F.

```
int score;
char grade;
cout << "Enter your test score... "<<endl;
cin >> score;
if (score >= 90)
    grade = 'A';
else if (score >= 80)
    grade = 'B';
else if (score >= 70)
    grade = 'C';
else if (score >= 60)
    grade = 'D';
else
    grade = 'F';
cout << "student grade is: "<<grade;
```

SAMPLE RUN:

```
Enter your test score...
76
student grade is: C
```

- **EXAMPLE 4-38:** Rewrite the implementation of the previous program as a sequence of `if` statements. DO NOT use `else` statement in your implementation.

```
int score;
char grade;
cout << "Enter your test score... "<<endl;
cin >> score;
if (score >= 90)
    grade = 'A';
if (score >= 80 && score < 90)
    grade = 'B';
if (score >= 70 && score < 80)
    grade = 'C';
if (score >= 60 && score < 70 )
    grade = 'D';
if (score < 60)
    grade = 'F';
cout << "student grade is: "<<grade;
```

SAMPLE RUN:

```
Enter your test score...
76
student grade is: C
```

- **THINK:** The previous two examples do the exact same thing, but not in the same amount of time. Find out which of them requires less time to execute if the student grade is 92 in each of them?
    - In example 1, the expression in the `if` statement in Line 5 evaluates to true. The statement (in Line 6) associated with this `if` then executes; the rest of the structure, which is the else of this `if` statement, is skipped; and the remaining if statements are not evaluated.
    - In example 2, the computer has to evaluate the expression in each if statement because there is no else statement.
    - SO, the program in example 2 executes more slowly than does the program in example 1.

- **PRACTICE:** Write a program to calculate the monthly paycheck of a salesperson at a local department store. Knowing that every salesperson has a base salary. The salesperson also receives a bonus at the end of each month, based on the following criteria: If the salesperson has been with the store for five years or less, the bonus is $10 for each year that he or she has worked there. If the salesperson has been with the store for more than five years, the bonus is $20 for each year that he or she has worked there. The salesperson can earn an additional bonus as follows: If the total sales made by the salesperson for the month are at least $5,000 but less than $10,000, he or she receives a 3% commission on the sale. If the total sales made by the salesperson for the month are at least $10,000, he or she receives a 6% commission on the sale.

Comparing Floating-Point Numbers for Equality: A Precaution

- Comparison of floating-point numbers for equality may not behave as you would expect. For example, consider the following program:
- **EXAMPLE 4-39:** What is the output of the following code snippet?

```
double x = 1.0;
double y = 3.0 / 7.0 + 2.0 / 7.0 + 2.0 / 7.0;

cout << "x = " << x << endl;
cout << "y = " << y << endl;

if(x == y)
    cout<<"x and y are equal"<<endl;
else
    cout<<"x and y are not equal"<<endl;

if (abs(x - y) < 0.000001)
    cout << "x and y are almost the same" << endl;
```

```
else
    cout << "x and y are not the same" << endl;
```

OUTPUT:

```
x = 1
y = 1
x and y are not the same
x and y are almost the same
```

Confusion between the Equality Operator (==) and the Assignment Operator (=)

- No matter how experienced a programmer is, almost everyone makes the mistake of using `=` in place of `==` at one time or another.
- The appearance of `=` in place of `==` can cause serious problems. It is not a syntax error, so the compiler does not warn you of an error. Rather, it is a logical error.
- **EXAMPLE 4-40:** What is the output of the following code snippet?

```
int x = 12, y = 12;
cout << (x = 5) << endl;
cout << (x == 5) << endl;
cout << (x == y) << endl;
```

OUTPUT:

```
5
1
0
```

- **EXAMPLE 4-41:** Study the following code snippet and explain why would you get the same result regardless the value of x.

```
if (x = 5)
    cout << "The value is five." << endl;
```

  - This expression is evaluated as follows.
    - First, the right side of the operator `=` is evaluated, which evaluates to 5. The value 5 is then assigned to x.
    - The value 5—that is, the new value of x—also becomes the value of the expression in the if statement—that is, the value of the assignment expression.
    - Because 5 is nonzero, the expression in the if statement evaluates to true, so the statement part of the if statement outputs: `The value is five.`

- **EXAMPLE 4-42:** Study the following code segment, and answer the questions below.

```cpp
int num = 7, x;
cout << "Enter an integer number..."<<endl;
cin >> x;

if (num = x)
    num += 5;
else
    num -=5;
```

- What will be the value stored in the variable `num` if user input is 7?

  12

- What will be the value stored in the variable `num` if user input is 10?

  15

- What will be the value stored in the variable `num` if user input is 0?

  -5

Conditional Operator (`?:`)

- The conditional operator, written as `?:`, is a ternary operator, which means that it takes three operands.
- Certain `if. . .else` statements can be written in a more concise way by using C++'s conditional operator.
- The syntax for using the conditional operator is:

```
expression1 ? expression2 : expression3
```

- The conditional expression is evaluated as follows:
  - If expression1 evaluates to a nonzero integer (that is, to true), the result of the conditional expression is expression2.
  - Otherwise, the result of the conditional expression is expression3.
- **EXAMPLE 4-43:** Assume **a**, **b** and **max** are `int` variables. Study the following code snippet and answer the question below.

```cpp
if (a >= b)
    max = a;
else
    max = b;
```

  - Write an equivalent code using the conditional operator.

```cpp
max = (a >= b) ? a : b;
```

- **EXAMPLE 4-44:** Assume **A** and **B** are `int` variables. Study the following code snippet and answer the question below.

```
if(A > B)
  cout << A << " is greater \n";
else
  cout<<B<< " is greater\n";
```

- Rewrite the previous code using the conditional operator.

  ```
   cout << ( A > B ? A : B ) << " is greater \n";
  ```

- **EXAMPLE 4-45:** Assume **n** is an `int` variable. Study the following code snippet and answer the question below.

  ```
  (n % 2 == 0) ? cout << n <<" :Even number\n" : cout << n <<":Odd number\n";
  ```

  - Rewrite the previous code using `if..else` statement.

    ```
    if(n% 2 == 0)
        cout<<n<<" :Even number\n" ;
    else
        cout<<n<<" :Odd number\n";
    ```

Avoiding Bugs by Avoiding Partially Understood Concepts and Techniques

- Understanding a concept or technique completely before using it will save you an enormous amount of debugging time.
- So, Avoid bugs by avoiding partially understood concepts and techniques 🤓.

Input Failure and the if Statement

- In Chapter 3, you saw that an attempt to read invalid data causes the input stream to enter a fail state.
- Once an input stream enters a fail state, all subsequent input statements associated with that input stream are ignored, and the computer continues to execute the program, which produces erroneous results.
- You can use `if` statements to check the status of an input stream variable and, if the input stream enters the fail state, include instructions that stop program execution.
- One way to address these causes of input failure is to check the status of the input stream variable (`cin`).
- You can check the status by using the input stream variable `cin` as the logical expression in an if statement.
  - If the last input succeeded, the input stream variable evaluates to true.

    - if the last input failed, it evaluates to false.
- **EXAMPLE 4-46:** Study the following code snippet and answer the following questions.

```
int x;
cout << "Enter an integer number..."<<endl;
cin >> x;
if(cin)
    cout << "Input is OK." << endl;
else
   cout << "Input failed." << endl;
```

    - What is the output if user input is **10**?

```
  Enter an integer number...
  10
  Input is OK.
```

    - What is the output if user input is **.10**?

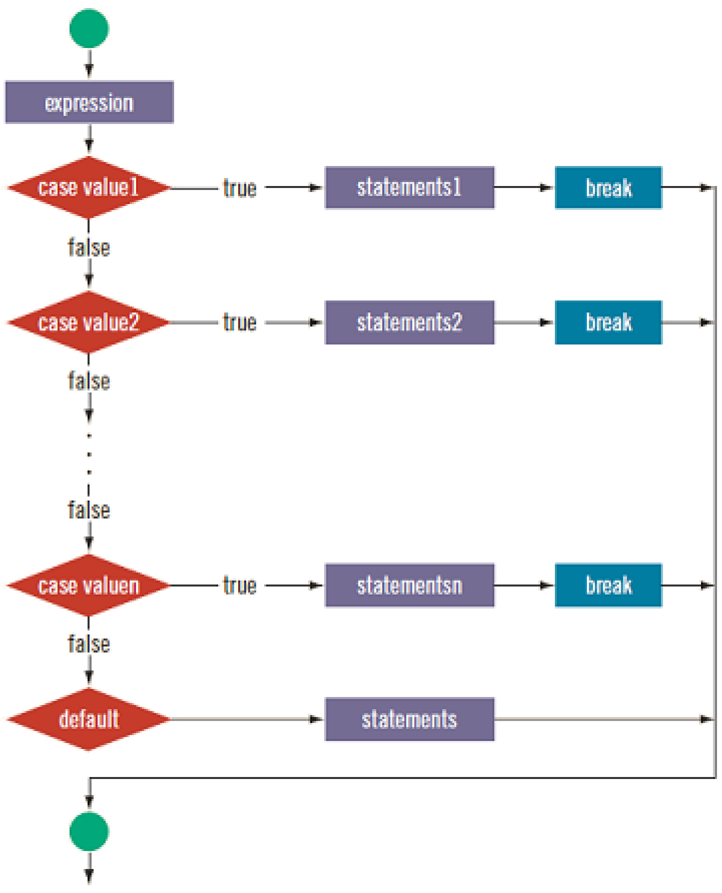
```
Enter an integer number...
.10
Input failed.
```

switch Structures

- **RECALL:** that there are two selection, or branch, structures in C++.
    - The first selection structure, which is implemented with `if` and `if. . .else` statements, usually requires the evaluation of a (logical) expression.
    - The second selection structure, which does not require the evaluation of a logical expression, is called the `switch` structure.
- C++'s **switch** structure gives the computer the power to choose from among many alternatives.
- A general syntax of the switch statement is:

```
switch (expression) //integral exp
{
    case value1:
        statements1
        break;
    case value2:
        statements2
        break;
        .
        .
        .
    case valuen:
        statementsn
        break;
    default:
        statements
}
```

- In C++, `switch`, `case`, `break`, and `default` are reserved words.
- In a switch structure, first the expression is evaluated. The value of the expression is then used to perform the actions specified in the statements that follow the reserved word `case`.
- Although it need not be, the expression is usually an identifier. Whether it is an identifier or an expression, the value can be only integral.
- The expression is sometimes called the **selector**. Its value determines which statement is selected for execution.
- A particular case value should appear only once.
- One or more statements may follow a case label, so you do not need to use braces to turn multiple statements into a single compound statement.
- The break statement may or may not appear after each statement.
- The figure below shows the flow of execution of the switch statement.



- The switch statement executes according to the following rules:
  1. When the value of the expression is matched against a case value (also called a label), the statements execute until either a break statement is found or the end of the switch structure is
     reached.
  2. If the value of the expression does not match any of the case values, the statements following the default label execute. If the switch structure has no default label and if the value of the expression
     does not match any of the case values, the entire switch statement is skipped.

3. A break statement causes an immediate exit from the switch structure.

- **EXAMPLE 4-47:** Study the following code snippet and answer the questions below.

```cpp
char day;
cout << "What day is today? Enter a number from 1 to 7.\n";
cin >> day;
switch (day)
{
    case '1':
        cout << "Sunday";
        break;
    case '2':
        cout << "Monday";
        break;
    case '3':
        cout << "Tuesday";
        break;
    case '4':
        cout << "Wednesday";
        break;
    case '5':
        cout << "Thursday";
        break;
    default:
    cout << "Weekend";
}
```

  - What is the output if user input is **3**?

```
What day is today? Enter a number from 1 to 7.
3
Tuesday
```

  - What is the output if user input is **7**?

```
What day is today? Enter a number from 1 to 7.
7
Weekend
```

- **EXAMPLE 4-48:** Study the following code snippet, and answer the questions below

```cpp
int num;
cout << "Enter an integer between 0 and 5: ";
cin >> num;

switch(num)
{
    case 0:
    case 1:
        cout << "One";
    case 2:
        cout << "Two";
    case 3:
        cout << "Three" << endl;
        break;
    case 4:
        break;
    case 5:
        cout << "Five";
    default:
        cout << "Invalid Input" << endl;
}
```

- What is the output if the user input is **1**?

  ```
  Enter an integer between 0 and 5: 1
  OneTwoThree
  ```

- What is the output if the user input is **5**?

  ```
  Enter an integer between 0 and 5: 5
  FiveInvalid Input
  ```

- **PRACTICE:** What is the output if the user input is 2, 3, 4, 6, and 7?

- **EXAMPLE 4-49:** Study the following code snippet and find out the line number that has syntax errors.

```
 1  double num;
 2  cout << "Enter an integer between 0 and 5: ";
 3  cin >> num;
 4
 5  switch(num)
 6  {
 7      case 2:
 8          cout << "Two";
 9      case 3:
10          cout << "Three" << endl;
11          break;
12      case 3:
13          break;
14      case 5:
15          cout << "Five";
16  }
```

  - Line 5: error: switch quantity not an integer
  - Line 12: error: duplicate case value

- **EXAMPLE 4-50:** Study the following C++ program, and rewrite it using nested `if-else` statements.

```
int i,n;
cout<< "Enter an integer"<<endl;
cin>>i;
switch(i)
{
    case 0:
    case 1:
        n=10;
        break;
    case 2:
        n=500;
        break;
    default:
        n = 0;
}
cout<<n;
```

- The following code performs the same operation

```
int i,n;
cout<< "Enter an integer"<<endl;
cin>>i;
if(i == 0 || i == 1)
    n=10;
else if(i == 2)
    n=500;
else
    n = 0;

cout<<n;
```

- When the value of the switch expression matches a case value, all statements execute until a break is encountered or reach the end of switch structure, and the program skips all case labels in between.
- **PRACTICE:** Rewrite the implementation of example 4-37 using switch statement:

```
int score;
char grade;
cout<<"Enter your score..."<<endl;
cin >> score;
switch (score / 10)
{
    case 0:
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        grade = 'F';
        break;
    case 6:
        grade = 'D';
        break;
    case 7:
        grade = 'C';
        break;
    case 8:
        grade = 'B';
        break;
    case 9:
    case 10:
        grade = 'A';
        break;
    default:
        cout << "Invalid test score." << endl;
}
cout<<"Student Grade is: "<<grade;
```

SAMPLE RUN:

```
Enter your score...
87
Student Grade is: B
```

- In addition to being a variable identifier or a complex expression, the switch expression can evaluate to a logical value. Consider the following statements:
- **EXAMPLE 4-51:** Study the following code snippet, and answer the questions below.

```
int score;
cout<<"Enter your score..."<<endl;
cin >> score;
switch (score >= 50)
{
case 1:
    cout << "Well done " << endl;
    cout << "You passed the exam." << endl;
    break;
case 0:
    cout << "Sorry" << endl;
    cout << "You failed the exam." << endl;
}
```

  - What is the output if score = 80?

```
Well done
You passed the exam.
```

  - What is the output if score = 40?

```
Sorry
You passed the exam.
```

  - Rewrite the previous code using `true` and `false`, instead of `1` and `0`, respectively, in the case labels.

- The `switch` statement is an elegant way to implement multiple selections.
- Even though no fixed rules exist that can be applied to decide whether to use an `if. . .else` structure or a `switch` structure to implement multiple selections, the following considerations should be remembered.
  - If multiple selections involve a range of values, you should use either an `if. . .else` structure or a `switch` structure, wherein you convert each range to a finite set of values.
  - If the range of values consists of infinitely many values and you cannot reduce them to a set containing a finite number of values, you must use the `if. . .else` structure.

C++ PROGRAMMING:

FROM PROBLEM ANALYSIS TO PROGRAM DESIGN

BY: D. S. MALIK

CHAPTER 5: CONTROL STRUCTURES II (REPETITION)

SUMMARY & EXAMPLES

PREPARED BY:

E. MANAR JARADAT

CONTROL STRUCTURES II (REPETITION)

- In this chapter, you will learn how repetitions are incorporated in programs.
- **EXAMPLE 5-1:** Write a C++ program to read five integer numbers from user and find their average .

  ```
  int num1, num2, num3, num4, num5;
  cout << "Enter 5 integer numbers"<<endl;
  cin >> num1 >> num2 >> num3 >> num4 >> num5;
  int sum = num1 + num2 + num3 + num4 + num5;
  int average = sum / 5;
  cout << "Average = "<<average;
  ```

  SAMPLE RUN: Assume user input is 5 7 8 2 7

  ```
  Enter 5 integer numbers
  5 7 8 2 7
  Average = 5
  ```

  - Extend the previous code to read 100 integers from user and find their average. 😒

- **EXAMPLE 5-2:** Rewrite the previous example using one variable to store the numbers read from user.

  ```
  int num, sum = 0;
  cout << "Enter 5 integer numbers"<<endl;
  cin >> num;
  sum += num;
  cin >> num;
  sum += num;
  cin >> num;
  sum += num;
  cin >> num;
  sum += num;
  cin >> num;
  sum += num;
  int average = sum / 5;
  cout << "Average = "<<average;
  ```

  SAMPLE RUN: Assume user input is 5 7 8 2 7

  ```
  Enter 5 integer numbers
  5 7 8 2 7
  Average = 5
  ```

  - Extend the previous code to read 100 integers from user and find their average.
  - What is the differences between the programs in examples 5.1 and 5.2?

Why Is Repetition Needed?

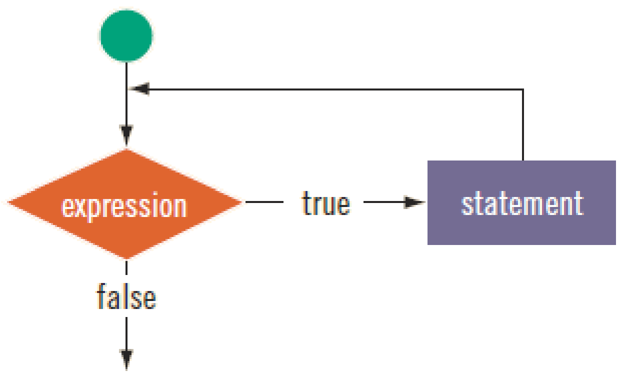
- There are many situations in which it is necessary to repeat a set of statements.
- C++ has three repetition, or looping, structures that let you repeat statements over and over until certain conditions are met.
  - `while` loop.
  - `for` loop.
  - `do..while` loop.

`while` Looping (Repetition) Structure

- This section discusses the first looping structure, called a `while` loop.
- The general form of the while statement is:

```
while (expression)
    statement
```

  - In C++, `while` is a reserved word.
  - The expression acts as a decision maker and is usually a logical expression.
  - Parentheses around the expression are part of the syntax.
  - The statement can be either a simple or compound statement, and it is called the **body of the loop**.
- The figure below shows the flow of execution of a while loop.
  - The expression provides an entry condition. If it initially evaluates to true, the statement executes.
  - The statement (body of the loop) continues to execute until the expression is no longer true.



- A loop that continues to execute endlessly is called an **infinite loop**.
- To avoid an infinite loop, make sure that the loop's body contains statement(s) that assure that the exit condition—the expression in the while statement—will eventually be false.
- **EXAMPLE 5-3:** Study the following code snippet and answer the questions below.

```
1  int i = 1;                   //initialize loop control variable
2  while (i <= 3)               // expression
3  {
4      cout << i << "\t";       // the statement to repeat
5      i++;                     // update loop control variable
6  }
7  cout << endl << "Done!";
```

- What is the output of the previous code snippet?

  ```
  1    2    3
  Done!
  ```

- How does the previous code works?

  ```
  int i = 1;                  // i = 1
  if (i <= 3)                 // true
  {
      cout << i << "\t";      // print(1)
      i++;                    // i = 2
  }
  if (i <= 3)                 // true
  {
      cout << i << "\t";      // print(2)
      i++;                    // i = 3
  }
  if (i <= 3)                 // true
  {
      cout << i << "\t";      // print(3)
      i++;                    // i = 4
  }
  if (i <= 3)                 // false
                              // Skip the body of the loop
  cout << endl << "Done!";    // print(Done)
  ```

- What will be the output if you do not initialize the variable `i` in the first statement?

  The loop may not execute at all

- What will be the output if you remove `i++` statement from the body of the loop?

  It will print 1 endlessly

- What will be the value stored in the variable `i` when the loop exit?

  4

- What will be the output if you interchange the statements at lines 4 and 5?

  ```
  2    3    4
  Done!
  ```

- Adding a semicolon (`;`) at the end of the while loop, (after the logical expression), makes the body of the `while` loop empty, that is, the statements within the braces do not form the body of the while loop.

- **EXAMPLE 5-4:** Study the following code snippet and answer the questions below.

```cpp
int i = 1;
while (i <= 3);
{
    cout << i << "\t";
    i++;
}
cout << endl << "Done!";
```

  - What are the statements that form the body of the loop?
    None of the statements, the loop body is empty
  - Does the while loop has any update statement for the loop control variable?
    No
  - What is the output of the previous program?
    The loop will repeated endlessly without doing anything

- **EXAMPLE 5-5:** Study the following code snippet and answer the questions below.

```cpp
int i = 1;
while (++i <= 3);
{
    cout << i << "\t";
    i++;
}
cout << endl << "Done!";
```

  - What are the statements that form the body of the loop?
    None of the statements, the loop body is empty
  - Does the while loop has any update statement for the loop control variable?
    Yes
  - What is the output of the previous program?

```
4
Done!
```

- **EXAMPLE 5-6:** Study the following code snippet, and answer the questions below.

```cpp
int i = 20;
while (i < 20)
{
    cout << i << " ";
    i = i + 5;
}
cout<<endl<<"Done!";
```

  - Does the loop entry condition evaluates to true?
    NO

- What is the output of the previous program?

```
Done!
```

- **EXAMPLE 5-7:** Write a C++ program to print all multiples of 5 between 150 and 200 (inclusive).

```cpp
int i = 150;
while (i <= 200)
{
    cout << i << "\t";
    i += 5;
}
```

  - Can you think in another way to solve this question?

- **EXAMPLE 5-8:** Write a C++ program to print all capital letters from 'Z' down to 'A' (inclusive).

```cpp
char ch = 'Z';
while (ch >= 'A')
{
    cout << ch << "\t";
    ch--;
}
```

Random Numbers Generation in C++

- In C++, you can use the function `rand` of the header file `cstdlib` to generate an integer random number between 0 and 32767.
- **EXAMPLE 5-9:** Study the following code snippet and answer the questions below.

```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    cout << rand() << ", " << rand() << endl;
}
```

  - What is the purpose of the previous code?

    Print two randomly generated integers whose values are between 0 and 32767

  - Execute the previous code multiple times, what do you notice?

    You will get the same random numbers every time

- The function `rand` uses an algorithm that produces the same sequence of random numbers each time the program is executed on the same system.
- To generate different random numbers each time the program is executed, you can use the function `srand` of the header file `cstdlib`.
- The function `srand` takes as input an `unsigned int`, which acts as the seed for the algorithm.
- By specifying different seed values, each time the program is executed, the function `rand` will generate a different sequence of random numbers.
- To specify a different seed, you can use the function `time` of the header file `ctime`, which returns the number of seconds elapsed since January 1, 1970.
- **EXAMPLE 5-10:** Modify the program in example 5-8 by adding `srand` function. Execute your program multiple times and notice the output.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    srand(time(0));
    cout << rand() << ", " << rand() << endl;
}
```

- You can use modulus (`%`) operator with `rand` function to generate random integer between A and B (inclusive). Where A and B are integer numbers.

```
int num = rand() % (B - A + 1) + A ;
```

- **EXAMPLE 5-11:** Write the required statement to randomly generate an integer number `num` between [0, 100].

```
// A = 0, B = 100
// B - A + 1 = 100 - 0 + 1 = 101
int num = rand() % 101;
```

- **EXAMPLE 5-12:** Write the required statement to randomly generate an integer number `num` between [50, 100].

```
// A = 50, B = 100
// B - A + 1 = 100 - 50 + 1 = 51
int num = rand() % 51 + 50;
```

Designing `while` Loops

- **RECALL:** The body of a `while` loop executes only when the expression, in the while statement, evaluates to true.
- **RECALL:** The expression checks whether a variable(s), called the loop control variable (LCV), satisfies certain conditions.
- **RECALL:** The LCV must be properly initialized before the while loop.
- **RECALL:** The LCV must be properly updated in the body of the while loop such that it eventually make the expression evaluate to false to avoid infinite loop.
- The general form of `while` loops is:

```
//initialize the loop control variable(s)
while (expression) //expression tests the LCV
{
    .
    .
    //update the loop control variable(s)
    .
    .
}
```

- while loops could take one of the following forms or a mix of them.
    - Counter-Controlled while Loops.
    - Sentinel-Controlled while Loops
    - flag-Controlled while Loops.

Case 1: Counter-Controlled while Loops

- A counter-controlled while loops are used when you know exactly how many times certain statements need to be executed.
- To execute a set of statements N times.
    - You can set up a counter (initialized to 0 before the while statement) to track how many items the loop executed.
    - The loop entry condition must compare the counter with N. If counter < N, the body of the while statement executes. The body of the loop continues to execute until the value of counter >= N.
    - The value of counter increments inside the body of the while statement.
- Counter-controlled while loop takes the following form:

```
int counter = 0;         //initialize the loop control variable
while (counter < N)      //test the loop control variable
{
    .
    .
    counter++;           //update the loop control variable
    .
    .
}
```

- **EXAMPLE 5-13:** Write a program to read 5 integers from user then calculate and print their sum and average.

```cpp
int N = 5;
double sum = 0;
int num;
int count = 0;          //initialize the loop control variable

while (count < N)
{
    cout<<"Enter an integer number"<<endl;
    cin >> num;

    sum += num;
    count++;            //update the loop control variable
}
cout << "Sum     = "<< sum << endl;
cout << "Average = "<< sum / N<< endl;
```

SAMPLE RUN: Assume user input is 4 6 2 4 7

```
Enter an integer number
4
Enter an integer number
6
Enter an integer number
2
Enter an integer number
4
Enter an integer number
7
Sum     = 23
Average = 4.6
```

- **EXAMPLE 5-14:** Students at a local middle school volunteered to sell fresh baked cookies to raise funds to increase the number of computers for the computer lab. Each student reported the number of boxes he/she sold. Write a program to calculate the total number of boxes of cookies sold, the total revenue generated by selling the cookies (assume cost of each box is $10), and the average number of boxes sold by each student.

```cpp
int numOfVolunteers;
int numOfBoxesSold;
int totalNumOfBoxesSold = 0;
int counter = 1;
double costOfOneBox = 10;
cout << "Enter the number of volunteers: ";
cin >> numOfVolunteers;

while (counter <= numOfVolunteers)
{
    cout << "Enter the number of boxes sold by volanteer "
         << counter << " : ";
    cin >> numOfBoxesSold;

    totalNumOfBoxesSold += numOfBoxesSold;
    counter++;
}

cout << "The total number of boxes sold: "
     << totalNumOfBoxesSold << endl;

cout << "The total money made by selling cookies: $"
    << totalNumOfBoxesSold * costOfOneBox << endl;

if (numOfVolunteers != 0)
    cout << "The average number of boxes sold by each volunteer: "
        << totalNumOfBoxesSold / numOfVolunteers << endl;
else
    cout << "No input." << endl;
```

SAMPLE RUN:

```
Enter the number of volunteers: 4
Enter the number of boxes sold by volanteer 1 : 6
Enter the number of boxes sold by volanteer 2 : 7
Enter the number of boxes sold by volanteer 3 : 3
Enter the number of boxes sold by volanteer 4 : 8
The total number of boxes sold: 24
The total money made by selling cookies: $240
The average number of boxes sold by each volunteer: 6
```

Case 2: Sentinel-Controlled while Loops

- A sentinel-controlled while loops are used when you need to keep read data until user enters a special value, called a `sentinel`.
- To read a set of values from user until the sentinel value is entered, do the following:
    - Read the first item before the while statement. If this item does not equal the sentinel, the body of the while statement executes.
    - In the body of the while loop keep read values from user, the while loop continues to execute as long as the program has not read the sentinel.
- Sentinel-Controlled while Loops might look as follows:

```cpp
cin >> variable;                //initialize the loop control variable
while (variable != sentinel)    //test the loop control variable
{
    .
    .
    cin >> variable;            //update the loop control variable
    .
    .
}
```

- **EXAMPLE 5-15:** Write a program to read integers from user continuously until he/she enters -99, then calculate and print their sum and average excluding -99.

```
//Declare variables
int count = 0;          // store the number of inserted integers
int sentinel = -99;
double sum = 0;         // store the sum of inserted integers
int num;

// Intialize the loop control variable
cout<<"Enter an integer number"<<endl;
cin >> num;

while (num != sentinel)
{
    sum += num;

    //update the loop control variable
    cout<<"Enter an integer number"<<endl;
    cin >> num;

    count++;
}
cout << "Sum     = "<< sum <<endl;
cout << "Average = "<< sum / count;
```

SAMPLE RUN:

```
Enter an integer number
4
Enter an integer number
6
Enter an integer number
9
Enter an integer number
10
Enter an integer number
-99
Sum     = 29
Average = 7.25
```

- **PRACTICE:** On a standard telephone keypad, the letters A-Z are mapped onto the phone number digits 0-9 as shown in the following diagram. A,B,C are mapped to 2, and D,E,F are mapped to 3, and so on. Write a program to read the letter codes A to Z from user and prints the corresponding telephone digit. Your program should continue read letters until **#** is read. You may ignore invalid inputs.

  Hint: The solution is available on textbook **EXAMPLE 5-5**

Case 3: Flag-Controlled while Loops

- A flag-controlled while loops are used when you need to execute a loop repeatedly until a condition is met.
- A flag-controlled while loop uses a bool variable to control the loop.
- To execute a loop repeatedly until a condition is met, do the following:
    - Initialize the flag variable to `false` .
    - The loop entry condition must ensure that the flag is not `true`. If the flag is not true the body of the while statement executes. The body of the loop continues to execute until the value of flag become true.
    - In the body of the while loop you must check for the stop condition, if the stop condition is true then set the flag to `true`.
- The flag-controlled while loop takes the following form:

```
bool found = false;      //initialize the loop control variable
while (!found)           //test the loop control variable
{
    .
    .
    if (expression)
        found = true;        //update the loop control variable
    .
    .
}
```

- **EXAMPLE 5-16:** Write a program to develop a number guessing game. In this game the program randomly generates an integer greater than or equal to 0 and less than or equal to 10. The program then prompts the user to guess the number. If the user guesses the number correctly, the program outputs an appropriate message. Otherwise, the program checks whether the guessed number is less than the random number. If the guessed number is less than the random number generated by the program, the program outputs the message “Your guess is lower than the number. Guess again!”; otherwise, the program outputs the message “Your guess is higher than the number. Guess again!”. The program then prompts the user to enter another number. The user is prompted to guess the random number until the user enters the correct number.

```cpp
//declare the variables
int secret;         // variable to store the random number
int guess;          // variable to store the number guessed by the user
bool isGuessed;     // boolean variable to control the loop

srand(time(0));
secret = rand() % 11;
isGuessed = false;
while (!isGuessed)
{
    cout << "Enter an integer greater within [0,10]"<<endl;
    cin >> guess;

    if (guess == secret)
    {
        cout << "You guessed the correct number." << endl;
        isGuessed = true;
    }
    else if (guess < secret)
        cout << "Your guess is lower than the number. Guess again!\n";
    else
        cout << "Your guess is higher than the number. Guess again!\n";
}
```

SAMPLE RUN:

```
Enter an integer greater within [0,10]
5
Your guess is higher than the number. Guess again!
Enter an integer greater within [0,10]
3
Your guess is higher than the number. Guess again!
Enter an integer greater within [0,10]
2
Your guess is higher than the number. Guess again!
Enter an integer greater within [0,10]
1
You guessed the correct number.
```

- **EXAMPLE 5- 17:** Modify the previous example such that it counts number of tries the user made to guess the correct number.

```cpp
//declare the variables
int secret;         // variable to store the random number
int guess;      // variable to store the number guessed by the user
bool isGuessed; // boolean variable to control the loop
int count = 0;  // variable to store number of tries.

srand(time(0));
secret = rand() % 11;
isGuessed = false;
while (!isGuessed)
{
    cout << "Enter an integer greater within [0,10]"<<endl;
    cin >> guess;
    count++;

    if (guess == secret)
    {
        cout << "You guessed the correct number." << endl;
        isGuessed = true;
    }
    else if (guess < secret)
        cout << "Your guess is lower than the number. Guess again!\n";
    else
        cout << "Your guess is higher than the number. Guess again!\n";
}
cout << endl << "You take "<<count
     <<" tries to guess the correct number";
```

SAMPLE RUN:

```
Enter an integer greater within [0,10]
5
Your guess is lower than the number. Guess again!
Enter an integer greater within [0,10]
8
Your guess is lower than the number. Guess again!
Enter an integer greater within [0,10]
9
You guessed the correct number.

You take 3 tries to guess the correct number
```

- **EXAMPLE 5-18:** Modify the previous code such that you give the user three tries at most to guess the number. If the user does not guess the number correctly within **three** tries, then the program outputs the random number generated by the program as well as a message that he have lost the game.

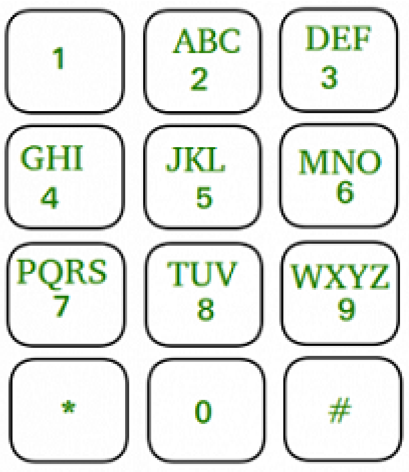
```
//declare the variables
int secret;     // variable to store the random number
int guess;      // variable to store the number guessed by the user
bool isGuessed; // boolean variable to control the loop
int count = 0;  // variable to store number of tries.

srand(time(0));
secret = rand() % 11;
isGuessed = false;
while (!isGuessed && count < 3)
{
    cout << "Enter an integer greater within [0,10]"<<endl;
    cin >> guess;
    count++;

    if (guess == secret)
    {
        cout << "You guessed the correct number." << endl;
        isGuessed = true;
```

```cpp
    }
    else if (guess < secret)
        cout << "Your guess is lower than the number. Guess again!\n";
    else
        cout << "Your guess is higher than the number. Guess again!\n";
}

if (count == 3 && !isGuessed)
    cout << endl << "Sorry! you loose the game"<<endl
         << "The secret number was "<<secret;
```

SAMPLE RUN

```
Enter an integer greater within [0,10]
2
Your guess is lower than the number. Guess again!
Enter an integer greater within [0,10]
4
Your guess is lower than the number. Guess again!
Enter an integer greater within [0,10]
5
Your guess is lower than the number. Guess again!

Sorry! you loose the game
The secret number was 7
```

- **PRACTICE:** On a standard telephone keypad, the letters A-Z are mapped onto the phone number digits 0-9 as shown in the following diagram. A,B,C are mapped to 2, and D,E,F are mapped to 3, and so on. Write a program to read the letter codes A to Z from user and prints the corresponding telephone digit. Your program should continue read letters until it reads any non-letter inputs.



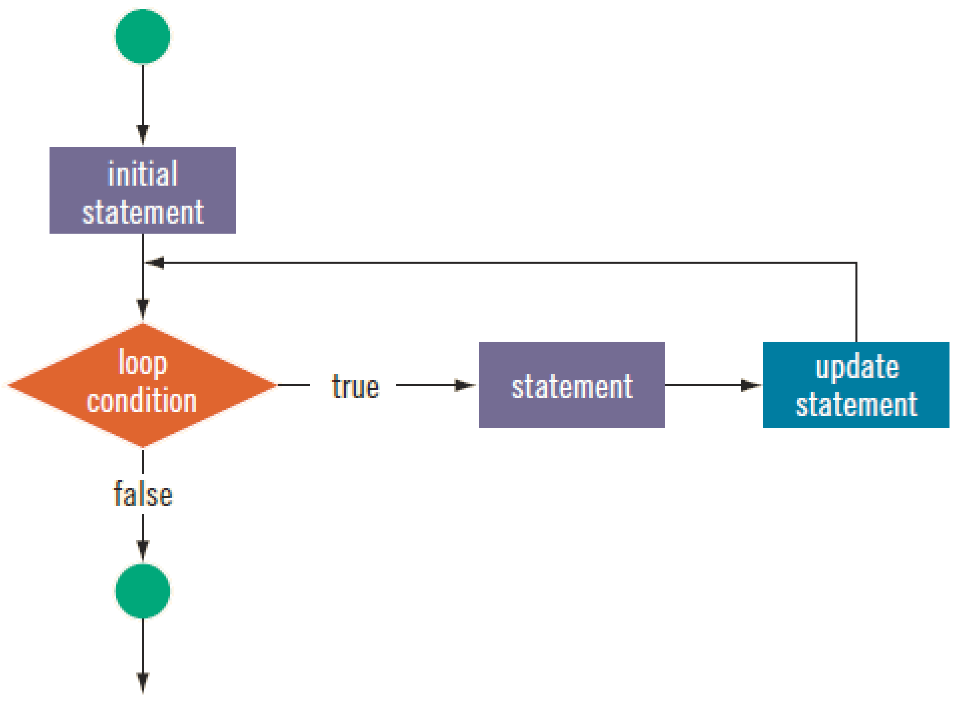
`for` Looping (Repetition) Structure

- The while loop discussed in the previous section is general enough to implement most forms of repetitions.

- The C++ `for` looping structure discussed here is a specialized form of the `while` loop. Its primary purpose is to simplify the writing of counter-controlled loops.
- The `for` loop is typically called a counted or indexed `for` loop.
- The general form of the `for` statement is:

```cpp
for (initial statement; loop condition; update statement)
    statement
```

  - The initial statement usually initializes a variable (called the for loop control variable).
  - The initial statement, loop condition, and update statement (called for loop control statements).
  - Control statements enclosed within the parentheses control the body (statement) of the `for` statement.
- The figure below shows the flow of execution of a `for` loop.



- The for loop executes as follows:
  1. The initial statement executes.
  2. The loop condition is evaluated. If the loop condition evaluates to true:
     1. Execute the for loop statement.
     2. Execute the update statement (the third expression in the parentheses).
  3. Repeat Step 2 until the loop condition evaluates to false.
- In C++, `for` is a reserved word.
- **NOTE:** The initial statement in the `for` loop is the first statement to execute; it executes only once.
- The syntax of the `for` loop, which is:

```cpp
for (initial expression; logical expression; update expression)
statement
```

- is functionally equivalent to the following `while` statement:

```
initial expression
while (logical expression)
{
    statement
    update expression
}
```

- **EXAMPLE 5-19:** Study the following code snippet, and answer the questions below.

```
int i = 0;
while (i < 10)
{
    cout << i << " ";
    i++;
}
```

  - Identify the initial statement, loop condition, and update statement in the previous code.
    - Initial statement: `int i = 0;`
    - Loop condition: `i < 10`
    - Update statement: `i++;`
  - Rewrite the previous code using `for` loop.

```
for (int i = 0; i < 10; i++)
cout << i << " ";
```

- **EXAMPLE 5-20:** Study the following code snippet and answer the questions below.

```
for (int i = 1; i <= 3; i++)
    cout << i << " ";
cout << endl<<"Done!";
```

- What is the output of the previous code snippet?

```
  1 2   3
  Done!
```

- How does the previous code works?

```
  i = 1;                        // i = 1
  if (i <= 3)                   // true
      cout << i << "\t";        // print(1)

  i++;                      // i = 2
  if (i <= 3)                   // true
      cout << i << "\t";        // print(2)
```

```

i++;                      // i = 3
if (i <= 3)                   // true
    cout << i << "\t";        // print(3)

i++;                      // i = 4
if (i <= 3)                   // false

cout << endl<<"Done!";
```

- Another valid forms of the `for` statement are:

1.
```
initial statement
for ( ; loop condition; update statement)
    statement
```

2.
```
for (initial statement; loop condition; )
{
    statement
    update statement
}
```

3.
```
initial statement
for ( ;loop condition ; )
{
    statement
    update statement
}
```

- The body of the `for` loop can be either a simple or a compound statement.
- **EXAMPLE 5-21:** What is the output of the following code segments?

```
int i = 4;
for ( ; i <= 5; i++)
    cout << "Hello!" << endl;
    cout << "*" << endl;
```

OUTPUT:

```
Hello!
Hello!
*
```

- **EXAMPLE 5-22:** What is the output of the following code segments.

```
int i = 4;
for ( ; i <= 5; )
{
    cout << "Hello!" << endl;
    cout << "*" << endl;
    i++;
}
```

OUTPUT:

```
Hello!
*
Hello!
*
```

- Adding a semicolon at the end of the for loop, (after the control statements), then the body of the for loop is empty, that is, the statements within the braces do not form the body of the for loop.
- A semicolon at the end of the for statement is a semantic error.
- **EXAMPLE 5-23:** Study the following code snippet and answer the questions below.

```
int i;
for (i = 3;i <= 5;i++);
    cout << i << " ";
```

  - What are the statements that form the body of the loop?
    The loop body is empty
  - Does the while loop has any update statement for the loop control variable?

    Yes
  - What is the output of the previous program?

    ```
    6
    ```

- **EXAMPLE 5-24:** Study the following code snippet and answer the questions below.

```
int i;
for (i = 3;i <= 5;);
{
    cout << i << " ";
    i++;
}
```

  - What are the statements that form the body of the loop?
    The loop body is empty
  - Does the while loop has any update statement for the loop control variable?

    NO
  - What is the output of the previous program?

The loop will execute endlessly

- If the loop condition is initially false, the loop body does not execute.
- **EXAMPLE 5-25:** Study the following code snippet, and find out how many times will statement at line 2 be executed?

```cpp
for (int i = 10; i <= 9; i++)
    cout << i << " ";
```

    - Zero times

- Omitting the loop condition from the for statement, makes the loop condition always `true`, and so,it will cause an infinite loop.
- **EXAMPLE 5-26:** What is the output of the following code snippet

```cpp
for (int i = 1; ; i++)
    cout << i << " ";
```

OUTPUT:

```
1 2 3 4 5 6 7 8 9 10 ...and so on
```

- In a `for` statement, you can omit all three statements—initial statement, loop condition, and update statement.
- **EXAMPLE 5-27:** What is the purpose of the following code snippet?

```cpp
for( ; ; )
    cout << "Hello" << endl;
```

    - This is an infinite for loop, it prints "Hello" endlessly.

- You can increment (or decrement) the loop control variable by any fixed number.
- **EXAMPLE 5-28:** What is the output of the following code snippet?

```cpp
for (int i = 10; i > 5; i--)
    cout << i << " ";
```

OUTPUT:

```
10 9 8 7 6
```

- **EXAMPLE 5-30:** What is the output of the following code segment?

```cpp
for (int i = 1; i <= 10; i = i + 3)
    cout << i << " ";
```

OUTPUT:

```
1 4 7 10
```

- C++ allows you to use fractional values for loop control variables of the double type (or any real data type). Because different computers can give these loop control variables different results, you should avoid using such variables.
- **EXAMPLE 5-31:** Write a program to read five numbers from user and count number of odd, even, and zeros in them.

```cpp
int i, num, odds = 0, evens = 0, zeros = 0;
cout<<"Enter 5 integers"<<endl;
for (i = 1; i <= 5; i++)
{
    cin >> num;
     if(num == 0)
         zeros++;
    else if( num % 2)
        odds++;
    else
        evens++;
}
cout << "# of zeros is " << zeros << endl;
cout << "# of odds is " << odds << endl;
cout << "# of evens is " << evens << endl;
```

SAMPLE RUN:

```
Enter 5 integers
0 6 0 3 7
# of zeros is 2
# of odds is 2
# of evens is 1
```

- **EXAMPLE 5-32:** Write to program to find the sum of the first **n** positive integers.

```cpp
int counter;        // loop control variable
int sum;            // variable to store the sum of numbers
int n;              // variable to store the number of
                    // first positive integers to be added

cout << "Enter the number of positive integers to be added: ";
cin >> n;

sum = 0;
for (counter = 1; counter <= n; counter++)
    sum = sum + counter;

```

```
cout << "The sum of the first " << n
     << " positive integers is " << sum << endl;
```

OUTPUT:

```
Enter the number of positive integers to be added: 4
The sum of the first 4 positive integers is 10
```

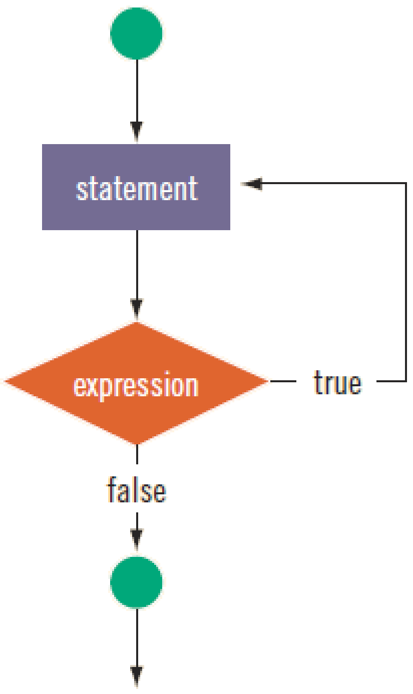
- **PRACTICE:** Write a program to read 10 integers and then prints the maximum number among them.

do...while Looping (Repetition) Structure

- This section describes the third type of looping or repetition structure, called a `do. . .while` loop.
- The general form of a `do. . .while` statement is as follows:

```
do
    statement
while (expression);
```

- statement can be either a simple or compound statement. If it is a compound statement, enclose it between braces.
- The figure below shows the flow of execution of a `do. . .while` loop.



- In C++, `do` and `while` are reserved words.

- The statement executes first, and then the expression is evaluated.
- If the expression evaluates to true, the statement executes again.
- As long as the expression in a `do...while` statement is true, the statement executes.
- To avoid an infinite loop, you must make sure that the loop body contains a statement that ultimately makes the expression false and assures that it exits properly.
- **EXAMPLE 5-33:**

```
int i = 0;
do
{
    cout << i << " ";
    i = i + 5;
}
while (i <= 20);
```

OUTPUT:

```
0 5 10 15 20
```

- **RECALL:** In a while and for loop, the loop condition is evaluated before executing the body of the loop. Therefore, while and for loops are called **pretest loops**.
- The loop condition in a `do. . .while` loop is evaluated after executing the body of the loop. Therefore, `do. . .while` loops are called **posttest loops**.
- Because the `while` and `for` loops both have entry conditions, these loops may never activate.
- The `do...while` loop, has an exit condition and therefore always executes the statement at least once.
- **EXAMPLE 5-34:** What is the output of the following code snippet?

```
int i = 11;
while (i <= 10)
{
    cout << i << " ";
    i = i + 5;
}
cout<<endl<<"Done!";
```

OUTPUT:

```

Done!
```

- **EXAMPLE 5-35:** What is the output of the following code snippet?

```cpp
int i = 11;
do
{
    cout << i << " ";
    i = i + 5;
}
while (i <= 10);
cout<<endl<<"Done!";
```

OUTPUT:

```
11
Done!
```

- A `do...while` loop can be used for input validation.
- **EXAMPLE 5-36:** To login to the student portal, a student should enter his/her password correctly. If the user forgets his password the system keeps prompting him/her to enter the password correctly. Once the student enters the correct password the system prints a greeting message. Write the required C++ to perform login operation.

```cpp
string username = "ahmad";
string password = "ah12345";
string tmpPassword;
do
{
    cout << "Enter your password ";
    cin >> tmpPassword;
}
while (tmpPassword != password);
cout<<endl<< "Welcome back "<<username<<" :)";
```

SAMPLE RUN:

```
Enter your password ahmad
Enter your password 12345
Enter your password ah12345

Welcome back ahmad :)
```

- **EXAMPLE 5-37:** Write a program to find the sum of digits of any positive integer.

  Hint: You can use modulus (`%`) and integer division (`/`) operators to extract all digits in the number.

```
int num, sum;
cout << "Enter an integer number "<<endl;
cin >> num;

sum = 0;
do
{
    sum = sum + num % 10;    //extract the last digit and add it to sum
    num = num / 10;          //remove the last digit
}
while (num > 0);
cout<< "Sum of digits in = "<<sum;
```

OUTPUT:

```
Enter an integer number
24968
Sum of digits = 29
```

- **PRACTICE:** It is known that an integer n is divisible by 3 if and only if the sum of its digits is divisible by 3. Write a program to check if a number entered from user is divisible by 3 or not, then print the proper message.

Choosing the Right Looping Structure

- If you know, or the program can determine in advance, the number of repetitions needed, the `for` loop is the correct choice.
- If you do not know, and the program cannot determine in advance the number of repetitions needed, and it could be 0, the `while` loop is the right choice.
- If you do not know, and the program cannot determine in advance the number of repetitions needed, and it is at least 1, the `do...while` loop is the right choice.

break and continue Statements

`break` statement

- The `break` statement, when executed in a `switch` structure, provides an immediate exit from the switch structure.
- Similarly, you can use the `break` statement in `while`, `for`, and `do. . .while` loops.
- When the break statement executes in a repetition structure, it immediately exits from the structure.
- The break statement is typically used for two purposes:
  - To exit early from a loop.
  - To skip the remainder of the switch structure.
- After the break statement executes, the program continues to execute with the first statement after the structure.
- **EXAMPLE 5-38:** What is the output of the following code segment.

```
int i = 1;
do
{
    if(i % 3 == 0)
        break;
    cout << i << " ";
    i++;
}
while(i < 5);
cout << endl << "Done @ i = "<<i;
```

OUTPUT:

```
1 2
Done @ i = 3
```

- **EXAMPLE 5-39:** A prime number is a number that is divisible only by the number 1 and itself. write a program to check if a number is prime or not.

```
int num;
cout<< "Enter an integer number"<<endl;
cin >> num;

bool isPrime = true;
for(int i = 2; i< num; i++)
{
    if(num % i == 0)
    {
        isPrime = false;
        break;
    }
}

cout<<x<<(isPrime?" is prime number":" is not prime number")<<endl;
```

SAMPLE RUN:

```
Enter an integer number
13
13 is a prime number
```

- **EXAMPLE 5-40:** Write a program that reads integer numbers continuously from user and calculate the sum of positive numbers only. Your program should stop reading once an input failure occurred or the user entered a negative number.

```
int num;
double sum = 0;

cout << "Enter an integer number"<<endl;
cin >> num;
while (cin)
{
    if (num < 0)
    {
        cout << "Negative number found in the data." << endl;
        break;
    }
    sum = sum + num;
    cout << "Enter an integer number"<<endl;
    cin >> num;
}
cout<<"sum = "<<sum;
```

OUTPUT:

```
Enter an integer number
7
Enter an integer number
9
Enter an integer number
3
Enter an integer number
-5
Negative number found in the data.
sum = 19
```

- **PRACTICE:** Rewrite the previous example using flag-controlled while loop.
- **NOTE:** The `break` statement is an effective way to avoid extra variables to control a loop and produce an elegant code.

`continue` statement

- The `continue` statement is used in `while`, `for`, and `do. . .while` structures.
- When the `continue` statement is executed in a loop, it skips the remaining statements in the loop and proceeds with the next iteration of the loop.
- In a `while` and `do. . .while` structure, the expression (that is, the loop-continue test) is evaluated immediately after the continue statement.
- In a `for` structure, the update statement is executed after the `continue` statement, and then the loop condition (that is, the loop-continue test) executes.
- **EXAMPLE 5-41:** Study the following code snippet and answer the following questions

```
int i = 1;
do
{
    if(i % 3 == 0)
        continue;
    cout << i << " ";
    i++;
}
while(i < 10);
cout << endl << "Done @ i = "<<i;
```

  - What is the output of the previous program?

It will print 1 2 and then start an infinite loop

- How to avoid the infinite loop in the previous program?

  By adding an increment statement before the continue statement

- Rewrite the previous code so that it print all numbers between 1 and 10 (inclusive) except multiples of three.

```
int i = 1;
do
{
    if(i % 3 == 0)
    {
        i++;
        continue;
    }
    cout << i << " ";
    i++;
}
while(i <= 10);
cout << endl << "Done @ i = "<<i;
```

- **EXAMPLE 5-42:** Write a program to print all numbers between 1 and 10 (inclusive) except multiples of three using for loop.

```
int i = 1;
for ( ;i <= 10; i++)
{
    if(i % 3 == 0)
        continue;
    cout << i << " ";
}
```

- **EXAMPLE 5-43:** Write a program that reads integer numbers continuously from user and calculate the sum of positive numbers only. Your program should skip the negative numbers and stop reading once an input failure occurred.

```
sum = 0;
cin >> num;
while (cin)
{
    if (num < 0)
    {
        cout << "Negative number found in the data." << endl;
        cin >> num;
        continue;
    }
    sum = sum + num;
    cin >> num;
}
```

- **NOTE:** When the continue statement is executed in a `while` or a `do. . .while` loop, the update statement may not execute. In a `for` structure, the update statement always executes (if the update statement written in the first line of `for` loop).

PROGRAMMING EXAMPLE: Fibonacci Number

- The following sequence of numbers is called the Fibonacci sequence.

```
1, 1, 2, 3, 5, 8, 13, 21, 34, ....
```

- Assumes that the first number of the Fibonacci sequence is less than or equal to the second number of the Fibonacci sequence, and both numbers are nonnegative. (say, `a1 = 1` and `a2 = 1`)
- Given the first two numbers of the sequence (`a1`, `a2`), the **nth** number `an`, `n >= 3`, of this sequence is given by:

```
an = an_1 + an_2
```

- The following steps are required to calculate the nth number in a Fibonacci series.
  1. Get the first two Fibonacci numbers.
  2. Get the desired Fibonacci number. That is, get the position, **n**, of the Fibonacci number in the sequence.
  3. Calculate the next Fibonacci number by adding the previous two elements of the Fibonacci sequence.
  4. Repeat Step 3 until the nth Fibonacci number is found.
  5. Output the nth Fibonacci number.
- The required code to calculate the nth number in a Fibonacci series is as follows.

```
//Declare variables
int previous1;
int previous2;
int current;
int counter;
int nthFibonacci;

cout << "Enter the first two Fibonacci numbers: "<<endl;
cin >> previous1 >> previous2;

cout << "Enter the position of the desired Fibonacci number: ";
cin >> nthFibonacci;

if (nthFibonacci == 1)
    current = previous1;
else if (nthFibonacci == 2)
    current = previous2;
else
{
    for( counter = 3; counter <= nthFibonacci; counter++)
    {
        current = previous2 + previous1;
        previous1 = previous2;
        previous2 = current;
    }
}

cout << "The Fibonacci number at position "
    << nthFibonacci << " is " << current;
```

SAMPLE RUN:

```
Enter the first two Fibonacci numbers:
1 1
Enter the position of the desired Fibonacci number: 7
The Fibonacci number at position 7 is 13
```

Nested Control Structures

- In this section, we give examples that illustrate how to use nested loops to achieve useful results and process data.
- **EXAMPLE 5-45:** Write a program to create the following pattern:

```
*
**
***
****
*****
```

```
int i, j;
for (i = 1; i <= 5; i++)
{
    for (j = 1; j <= i; j++)
        cout << "*";
    cout << endl;
}
```

- **PRACTICE:** Write a program to create the the following pattern.

```
*****
****
***
**
*
```

  - Modify your program so that you read number of rows from user.

- **EXAMPLE 5-46:** Consider the following multiplication table and answer the questions below

1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50

- Write a program to print the previous multiplication table.

```
for (i = 1; i <= 5; i++)
{
    for (j = 1; j <= 10; j++)
        cout << setw(3) << i * j;
    cout << endl;
}
```

- Modify your program so that you calculate and print the sum of numbers in each line next to it.

  ```
  int sum, tmp;
  for (int i = 1; i <= 5; i++)
  {
      sum = 0;
      for (int j = 1; j <= 10; j++)
      {
          tmp = i *j;
          sum+=tmp;
          cout << setw(3) << tmp;
      }
      cout << " = "<<sum<<endl;
  }
  ```

CHAPTER 6: USER-DEFINED FUNCTIONS

- **RECALL:** C++ program is a collection of functions. One such function is `main`.
- For large programs, it is not practical (although it is possible) to put the entire programming instructions into one function.
- In this chapter we will learn how to use functions to break the problem into manageable pieces.
- **Functions** are often called modules. They are like small programs; you can put them together to form a larger program.
- The following are some advantages of using functions:
  - They let you divide complicated programs into manageable pieces.
  - While working on one function, you can focus on just that part of the program and construct it, debug it, and perfect it.
  - Different people can work on different functions simultaneously.
  - If a function is needed in more than one place in a program or in different programs, you can write it once and use it many times.
  - Using functions greatly enhances the program's readability because it reduces the complexity of the function main.
- Functions flow of execution
  - When the program executes, the first statement in the function `main` always executes first, regardless of where in the program the function `main` is placed. Other functions execute only when they are called.
  - A function call statement transfers control to the first statement in the body of the function. In general, after the last statement of the called function executes, control is passed back to the point immediately following the function call.
  - A value-returning function returns a value. Therefore, after executing the value-returning function, when the control goes back to the caller, the value that the function returns replaces the function call statement. The execution continues at the point immediately following the function call.
- First we will review how to use predefined functions. then we will learn how to write our functions

Predefined Functions

- In C++, the concept of a function, either predefined or user-defined, is similar to that of a function in algebra.

```
f(x) = 3x + 5
```

  - Every function has a name and, depending on the values specified by the user, it does some computation.
- In C++, predefined functions are organized into separate libraries.
  - The header file `iostream` contains I/O functions.
  - The header file `cmath` contains math functions.
- To use these functions in your programs, you must know the following:

- The name of the header file that contains the functions' specification. You need to include this header file in your program using the `include` statement.

```
#include <cmath>
```

- The heading of the function (also called the function header)
  1. The name of the function.
  2. The number of parameters, if any, and the data type of each parameter.
  3. The data type of the value computed (that is, the value returned) by the function, called the type of the function

```
int abs(int number)
```

- The purpose of the function.
  - Return the absolution value of its argument.

- The following table lists some of the predefined functions.

Function	Header File	Purpose	Parameter(s) Type	Result
`abs(x)`	`<cmath>`	Returns the absolute value of its argument.	`int` (`double`)	`int` (`double`)
`pow(x, y)`	`<cmath>`	Returns x^y; if x is negative, y must be a whole number.	`double`	`double`
`sqrt(x)`	`<cmath>`	Returns the nonnegative square root of x; x must be nonnegative.	`double`	`double`
`ceil(x)`	`<cmath>`	Returns the smallest whole number that is not less than x.	`double`	`double`
`floor(x)`	`<cmath>`	Returns the largest whole number that is not greater than x.	`double`	`double`
`exp(x)`	`<cmath>`	Returns e^x, where e = 2.718	`double`	`double`
`islower(x)`	`<cctype>`	Returns 1 (true) if x is a lowercase letter; otherwise, it returns 0 (false).	`int`	`int`
`isupper(x)`	`<cctype>`	Returns 1 (true) if x is an uppercase letter; otherwise, it returns 0 (false).	`int`	`int`
`tolower(x)`	`<cctype>`	Returns the lowercase value of x if x is uppercase; otherwise, it returns x.	`int`	`int`
`toupper(x)`	`<cctype>`	Returns the uppercase value of x if x is lowercase; otherwise, it returns x.	`int`	`int`

- **EXAMPLE 6-1:** What is the output of each of the following C++ statements.

	Statement	Output
1.	`cout<<abs(-7);`	7

	Statement	Output
2.	`cout<<abs(3.9);`	3.9
3.	`cout<<pow(16, 0.5);`	4
4.	`cout<<sqrt(4.0);`	2
5.	`cout<<static_cast<int> (sqrt(28.00));`	5
6.	`cout<<ceil(56.34);`	57
7.	`cout<<ceil(-56);`	-56
8.	`cout<<floor(-56.34);`	-57
9.	`cout<<floor(56);`	56
10.	`cout<<exp(1.0);`	2.71828
11.	`cout<<islower('2');`	0
12.	`cout<<islower('b');`	1
13.	`cout<<islower('B');`	0
14.	`cout<<isupper('2');`	0
15.	`cout<<isupper('b');`	0
16.	`cout<<isupper('B');`	1
17.	`cout<<toupper('2');`	50
18.	`cout<<toupper('b');`	66
19.	`cout<<toupper('B');`	66
20.	`cout<<tolower('2');`	50
21.	`cout<<tolower('b');`	98
22.	`cout<<tolower('B');`	98
23.	`cout<<static_cast<char>(toupper('a'));`	A

User-Defined Functions

- C++ does not provide every function that you will ever need and designers cannot possibly know a user’s specific needs, so you must learn to write your own functions.
- User-defined functions in C++ are classified into two categories:
    - **Value-returning functions**—functions that have a return type. These functions return a value of a specific data type using the `return` statement.

- **Void functions**—functions that do not have a return type. These functions do not use a `return` statement to return a value.

Value-Returning Functions

- The previous section introduced some predefined C++ functions such as `pow`, `abs`, `islower`, and `toupper`. These are examples of value-returning functions.
- Because the value returned by a value-returning function is unique, the natural thing for you to do is to use the value in one of three ways:
  - In an assignment statement.

    ```
    area = PI * pow(radius, 2.0);
    ```

  - As a parameter in a function call.

    ```
    area = PI * pow(abs(radius), 2.0);
    ```

  - In an output statement.

    ```
    cout<< PI * pow(radius, 2.0);
    ```

Value-Returning Function Definition

- The syntax of a value-returning function is:

  ```
  functionType functionName(formal parameter list)  //function header
  {
      //function body
      statements
  }
  ```

  - **functionType** (data type) is the type of the value that the function returns.
  - **functionName** is any valid identifier.
  - **Formal parameter list** are list of variables declared in the function heading.
  - The **statements** enclosed between curly braces `{ }` form the body of the function.
- The syntax of the formal parameter list is:

  ```
  dataType identifier, dataType identifier, ...
  ```

- Once a value-returning function computes the desired value, the function returns this value via the `return` statement.
- The return statement has the following syntax:

  ```
  return expr;
  ```

  - In C++, `return` is a reserved word.
  - `expr` is a variable, constant value, or expression.

- The `expr` is evaluated, and its value is returned.
- The data type of the value that `expr` computes must match the function type.

- When a return statement executes in a function, the function immediately terminates and the control goes back to the caller. Moreover, the function call statement is replaced by the value returned by the return statement.
- When a `return` statement executes in the function main, the program terminates.
- **EXAMPLE 6-2:** Write the required C++ function definition to calculate the result of the following mathematical formula. $F(x) = 2x + 5$

```cpp
int Fun1(int x)
{
    return 2 * x + 5;
}
```

- **EXAMPLE 6-3:** The roots of a quadratic equation $ax^2 + bx + c$ are calculated using the following formula $F(a, b, c) = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$. Write a C++ function definition to calculate one of the quadratic equation roots.

```cpp
double Fun2(double a, double b, double c)
{
    double tmp = b*b - 4*a*c;
    return (-b + sqrt(tmp))/(2 * a);
}
```

Value-Returning Function Call

- To call a value returning function, you use its name, with the actual parameters (if any) in parentheses.
- The syntax to call a value-returning function is:

```
functionName(actual parameter list)
```

- **Actual parameter list** is the list of variables or expressions listed in a call to a function.
- The syntax of the actual parameter list is:

```
expression or variable, expression or variable, ...
```

  - Expression can be a single constant value.

- To execute a function call, the parameters are evaluated first.
- **EXAMPLE 6-4:** Study the definition of functions `Fun1` and `Fun2` and find out the output of each of the following function calls.

	Statement	**Output**
1.	`int x = 3;` `cout<<Fun1(x);`	11

	Statement	Output
2.	`cout<<Fun1(2.8);`	9
3.	`cout<<Fun1('2');`	105
4.	`cout<<Fun1(Fun1(1));`	19
5.	`double c = 5;` `cout<<Fun2(5%3, -1*Fun1(3), c);`	5

- A function's formal parameter list can be empty. However, if the formal parameter list is empty, the parentheses are still needed.

```
functionType functionName()
```

- If the formal parameter list of a value-returning function is empty, the actual parameter is also empty in a function call, and the empty parentheses are still needed.
- A call to a value-returning function with an empty formal parameter list is:

```
functionName()
```

- **EXAMPLE 6-5:** Study the following definition of function `pi` and answer the questions below.

```
double pi()
{
    return 22.0 / 7;
}
```

  - What is the purpose of function **pi**?

    Calculate and returns the value of π

  - Write the required C++ statement to calculate and print the area of a circle whose radius = 10.

```
cout << pi() * 10 * 10;
```

  - Modify the return type of function **pi** to `int` and notice the change in the circle area value.

- In a function call, you specify only the actual parameter, not its data type.
- In a function call, the number of actual parameters, together with their data types, must match with the formal parameters in the order given. That is, actual and formal parameters have a one-to-one correspondence.
- **EXAMPLE 6-6:** Study the following function definition and answer the questions below.

```cpp
char Fun(string str, int x)
{
    if( x >= 0 && x < str.length())
        return str[x];
    return '\0';
}
```

- What is the purpose of the previous function?

  Return the character at position **x** in string **str**

- Which of the following C++ statements is a correct function call for function `Fun`.

	Statement	**Is correct?**
1.	`cout<<Fun("spring", 3);`	True
2.	`cout<<Fun("spring", 3.7);`	True
3.	`cout<<Fun("spring", "Summer");`	False
4.	`cout<<Fun(2, "spring");`	False
5.	`cout<<Fun("spring");`	False
6.	`cout<<Fun("spring",3,2);`	False
7.	`cout<<Fun(string s,int b);`	False

- **EXAMPLE 6-7:** Write the definition of function `power` that takes two integers as parameters and returns the value of its first parameter to the power of the second parameter. Test the correctness of your function.

```cpp
int power(int x, int y)
{
    int result = 1;
    for(int i=0; i<y; i++)
        result *= x;
    return result;
}
int main()
{
    cout<<power(2, 5);
}
```

Value-Returning Functions: Some Peculiarities

- Having more than one expression in a `return` statement will not cause compilation error, however it may result in redundancy, wasted code, and a confusing syntax.
- A return statement returns only one value. Even if the return statement contains more than one expression, it will return the value of the last expression .

```cpp
return x, y;              // The value of y is returned.
```

- **EXAMPLE 6-8:** What is the output of the following program

```cpp
int funcRet1()
{
    int x = 45;
    return 23, x;       //only the value of x is returned
}

int funcRet2(int z)
{
    int a = 2;
    int b = 3;
    return 2 * a + b, z + b; //only the value of z + b is returned
}

int main()
{
    int num = 4;
    cout << funcRet1() << endl;
    cout << funcRet2(num) << endl;
}
```

OUTPUT:

```
45
7
```

- **EXAMPLE 6-9:** Study the following definition of function **secret** and answer the questions below.

```cpp
int secret(int x)
{
    if (x > 5)
        return 2 * x;
}
```

  - What is the output of the following C++ statement?

```cpp
cout<<secret(10);       //20
```

  - What is the output of the following C++ statement?

```cpp
cout<<secret(3);        //??
```

- For value-returning functions that have selection statements, the function should return a value from all possible paths.
- Because **secret** is a value-returning function that have a selection statement you need to return a value either if the formal parameter value is less than or greater than 5. A correct definition of the function **secret** is:

```cpp
int secret(int x)
{
    if (x > 5)
        return 2 * x;
    else
        return x;
}
```

OR: Because the execution of a `return` statement in a function terminates the function, the preceding function **secret** can also be written (without the word else) as:

```cpp
int secret(int x)
{
    if (x > 5)
        return 2 * x;
    return x;
}
```

- **PRACTICE:** Write the definition of function **absolute** that takes a decimal number as a parameter, and returns its absolute value. Test the correctness of your function.
- **EXAMPLE 6-10:** Write the definition of function `courseGrade`. This function takes as a parameter an `int` value specifying the score (value between 50 and 100 inclusive) for a course and returns the grade (type char). Test the correctness of your function.

```cpp
char courseGrade(int score)
{
    switch (score / 10)
    {
        case 5:
            return 'F';
        case 6:
            return 'D';
        case 7:
            return 'C';
        case 8:
            return 'B';
        case 9:
        case 10:
            return 'A';
        default:
           return 'F';
    }
}

int main()
{
    cout<<courseGrade(70)<<endl;
    cout<<courseGrade(95)<<endl;
}
```

- **EXAMPLE 6-11:** Write a function that rolls a pair of dice until the sum of the numbers rolled is a specific number, and return the number of times the dice are rolled to get the desired sum. Test the correctness of your function.

```cpp
int rollDice(int num)
{
    int die1, die2, sum;
    srand(time(0));

    int rollCount = 0;
    if(num >= 2 && num <= 12)
    {
        do
        {
            die1 = rand() % 6 + 1;
            die2 = rand() % 6 + 1;
            sum = die1 + die2;
            rollCount++;
        }
        while (sum != num);
    }
    return rollCount;
}

int main()
{
    cout<< "The number of times the dice are rolled to "
        << "get the sum 15 = " << rollDice(15) << endl;
    cout<< "The number of times the dice are rolled to "
        << "get the sum 10 = " << rollDice(10) << endl;
}
```

    - **PRACTICE:** Modify this program so that it allows the user to enter the desired sum of the numbers to be rolled.
- **PRACTICE:** A string is a palindrome if it reads forward and backward in the same way. For example, the strings "madam", "5", "434", and "789656987" are all palindromes. write the definition of a C++ function `isPalindrome` that returns true if a string is a palindrome and false otherwise.

Function Prototype

- **EXAMPLE 6-12:** Write the definition of function **Larger** that takes two decimal numbers as parameters, and returns the larger number between them. Test the correctness of your function.

```
double larger(double x, double y)
{
    if (x >= y)
        return x;
    return y;
}
int main()
{
    double one = 13.00;
    cout<< larger(one, 29) << endl;
}
```

- **EXAMPLE 6-13:** Use function **Larger** we studied earlier to write the definition of function **compareThree** that determine the largest of three numbers.

```
double larger(double x, double y)
{
    if (x >= y)
        return x;
    return y;
}

double compareThree(double x, double y, double z)
{
    return larger(x, larger(y, z));
}

int main()
{
    double one = 13.00;
    cout<< compareThree(one, 29,34) << endl;
}
```

- What is the order in which user-defined functions should appear in a program?
  - Place the definition of function `larger` after the definition of function `compareThree`. Compile the program and note what happened.
  - Following the rule that you must declare an identifier before you can use it and knowing that the function `compareThree` uses the identifier `larger`, logically you must place `larger` before `compareThree`.
- C++ programmers usually place the function `main` before all other user-defined functions. However, this organization could produce a compilation error because functions are compiled in the order in which they appear in the program.
- **EXAMPLE 6-14:** Study the following code snippet, and identify the line number that will cause a syntax error.

```
 1  int main()
 2  {
 3      double one = 13.00;
 4      cout<< larger(one, 29) << endl;
 5  }
 6  double larger(double x, double y)
 7  {
 8      if (x >= y)
 9          return x;
10      return y;
11  }
```

  - Line 4: error: 'larger' was not declared in this scope

- To work around the previous problem of undeclared identifiers, we place function prototypes before any function definition (including the definition of main).
- **Function Prototype:** The function heading without the body of the function.
- The general syntax of the function prototype of a value-returning function is:

```
functionType functionName(parameter list);
```

  - Note that the function prototype ends with a semicolon.
- For the function **larger**, the prototype is:

```
double larger(double x, double y);
```

- When writing the function prototype, you do not have to specify the variable name in the parameter list. However, you must specify the data type of each parameter.
- You can rewrite the function prototype of the function larger as follows:

```
double larger(double, double);
```

- **EXAMPLE 6-15:** Rewrite the previous example to solve undeclared identifiers problem.

```
double larger(double, double);
int main()
{
    double one = 13.00;
    cout<< larger(one, 29) << endl;
}

double larger(double x, double y)
{
    if (x >= y)
        return x;
    return y;
}
```

- **PRACTICE:** Rewrite example 6-13 so that you place the functions prototype before the definition of function `main`.

Void Functions

- Void functions and value-returning functions have similar structures, both have a heading and a body.
    - Like value-returning functions, can place user-defined void functions either before or after the function main.
    - If you place user-defined void functions after the function main, you should place the function prototype before the function main.
- A void function does not have a data type. Therefore, `functionType`—that is, the return type—in the heading part and the `return` statement in the body of the void functions are meaningless.
- The function definition of void functions with parameters has the following syntax:

```
void functionName(formal parameter list)
{
    statements
}
```

    - Statements are usually declaration and/or executable statements.
    - The formal parameter list may be empty, in which case, in the function heading, the empty parentheses are still needed.
- In a void function, you can use the `return` statement without any value; it is typically used to exit the function early.

- **EXAMPLE 6-16:** Write the definition of function **weekDay** that takes an integer number as a parameter and prints the corresponding day name.

```
void weekDay(int day)
{
    switch(day)
    {
        case 1:
            cout<<"Sunday";
            return;
        case 2:
            cout<<"Monday";
            return;
        case 3:
            cout<<"Tuesday";
            return;
        case 4:
            cout<<"Wednesday";
            return;
        case 5:
            cout<<"Thursday";
            return;
        case 6:
            cout<<"Friday";
            return;
        case 7:
            cout<<"Saturday";
            return;
```

```cpp
        default:
            cout<<"Invalid day";
            return;
    }
}
```

- Because void functions do not have a data type, they are not used (called) in an expression.
- A call to a void function is a stand-alone statement. Thus, to call a void function, you use the function name together with the actual parameters (if any) in a stand-alone statement.
- The function call has the following syntax:

```cpp
functionName(actual parameter list);
```

- **RECALL:** In a function call, Actual and formal parameters have a one-to-one correspondence, that is, the number of actual parameters together with their data types must match the formal parameters in the order given.
- **EXAMPLE 6-17:** Which of the following statements is correct function call to function **weekDay**.

	Statement	Is correct?
1.	`weekDay(3);`	true
2.	`weekDay(3, 5);`	false
3.	`cout<<weekDay(3);`	false
4.	`string day = weekDay(4);`	false

- **EXAMPLE 6-18:** Write a C++ function `printStars` that takes two integer numbers (`n` and `r`) as parameters. Your function should print `n` stars divided into `r` stars per row. For example the following function call `printStars(14, 6)` will print the following shape.

```
* * * * * *
* * * * * *
* *
```

```cpp
void printStars(int n, int r)
{
    for (int count = 1; count <= n; count++)
    {
        cout << "* ";
        if(count % r == 0)
            cout << endl;
    }
}
```

Parameter Types

- Parameters provide a communication link between the calling function (such as `main`) and the called function. They enable functions to manipulate different data each time they are called.
- In general, there are two types of formal parameters:
    - **Value parameter:** A formal parameter that receives a copy of the content of the corresponding actual parameter.
    - **Reference parameter:** A formal parameter that receives the location (memory address) of the corresponding actual parameter.
- When you attach `&` after the dataType in the formal parameter list of a function, the variable following that dataType becomes **a reference parameter**.
- **EXAMPLE 6-19:** Consider the following function definition and answer the questions below.

```
void areaAndPerimeter(double length, double width,
                      double& area, double& perimeter)
{
    area = length * width;
    perimeter = 2 * (length + width);
}
```

    - How many value parameters does the function **areaAndPerimeter** has?

      2 parameters, length and width

    - How many reference parameters does the function **areaAndPerimeter** has?

      2 parameters, area and perimeter

Value Parameters

- When a function is called, memory for its formal parameters and variables declared in the body of the function (called local variables) is allocated in the function data area.
- Value parameter copy the value of the actual parameter into the memory cell of its corresponding formal parameter.
    - The formal parameter has its own copy of the data. Therefore, during program execution, the formal parameter manipulates the data stored in its own memory space.
    - When the function executes, any changes made to the formal parameters do not in any way affect the actual parameters.
- Value parameters cannot pass information outside of the function.
- **EXAMPLE 6-20:** What is the output of the following program?

```
void Swap(int num1, int num2)
{
    int tmp = num1;
    num1 = num2;
    num2 = tmp;
    cout<< "Inside function Swap..."<<endl;
    cout<< "num1 = " << num1
        << ", num2 = " << num2<< endl;
}
int main()
{
    int number1 = 6;
```

```cpp
    int number2 = 13;
    cout<< "Before calling the function Swap"<<endl;
    cout<< "number1 = " << number1
        << ", number2 = " << number2<< endl;
    Swap(number1,number2);
    cout<< "After calling the function Swap"<<endl;
    cout<< "number1 = " << number1
        << ", number2 = " << number2<< endl;
}
```

OUTPUT:

```
Before calling the function Swap
number1 = 6, number2 = 13
Inside function Swap...
num1 = 13, num2 = 6
After calling the function Swap
number1 = 6, number2 = 13
```

Reference Variables as Parameters

Address-of Operator (`&`)

- The **address-of operator** is a unary **operator** represented by an ampersand (&).
- The **address-of** operator, `&`, is used to create aliases to a variable.

- Consider the following statements:

```cpp
int x;          // x = ??
int &y = x;     // x,y = ??
```

  - The first statement declares **x** to be an `int` variable.
  - The second statement declares **y** to be an alias of **x**, that is, both **x** and **y** refer to the same memory location.
- **EXAMPLE 6-21:** What is the output of the following code snippet.

```cpp
int x;
int &y = x;
y = 25;             // x,y = 25
x = 2 * x + 30;     // x,y = 80
cout<<x<<y;
```

OUTPUT:

```
8080
```

- Once you set an alias to a variable, you can't modify it to become an alias to another variable.

```cpp
int x = 3, w = 7;    // x = 3, w = 7
int &y = x;          // x,y = 3, w = 7
y = w;               // x,y = 7, w = 7
&y = w;              // error
```

- Any attempt to create an alias to a constant value will cause a compilation error.

```cpp
int &a = 15;     // error: invalid initialization of non-const reference
                 // of type 'int&' from an rvalue of type 'int'
```

Reference Parameter

- **RECALL:** When you attach `&` after the dataType in the formal parameter list of a function, the variable following that dataType becomes a reference parameter.
- **RECALL:** When a function is called, memory for its formal parameters and variables declared in the body of the function (called local variables) is allocated in the function data area.
- Reference parameter receives the address (memory location) of the actual parameter. That is, the content of the formal parameter is an address.
- In reference parameter, both the actual and formal parameters refer to the same memory location. Consequently, during program execution, changes made by the formal parameter permanently change the value of the actual parameter.
- Reference parameters are useful in three situations:
  - When the value of the actual parameter needs to be changed.
  - When you want to return more than one value from a function (recall that the return statement can return only one value).
  - When passing the address would save memory space and time relative to copying a large amount of data.
- **EXAMPLE 6-22:** What is the output of the following program?

```cpp
void Swap(int &num1, int &num2)
{
    int tmp = num1;
    num1 = num2;
    num2 = tmp;
    cout<< "Inside function Swap..."<<endl;
    cout<< "num1 = " << num1
        << ", num2 = " << num2<< endl;
}
int main()
{
    int number1 = 6;
    int number2 = 13;
    cout<< "Before calling the function Swap"<<endl;
    cout<< "number1 = " << number1
        << ", number2 = " << number2<< endl;
    Swap(number1,number2);
    cout<< "After calling the function Swap"<<endl;
    cout<< "number1 = " << number1
        << ", number2 = " << number2<< endl;
}
```

OUTPUT:

```
Before calling the function Swap
number1 = 6, number2 = 13
Inside function Swap...
num1 = 13, num2 = 6
After calling the function Swap
number1 = 13, number2 = 6
```

- **EXAMPLE 6-23:** What is the output of the following program?

```cpp
void funOne(int a, int& b, char v)
{
    cout << "Inside funOne:" << endl;
    int one;
    one = a;
    a++;
    b = b * 2;
    v = 'B';
    cout<< "a = " << a <<endl
        << "b = " << b <<endl
        << "v = " << v <<endl
        << "one = " << one << endl;
}
void funTwo(int& x, int y, char& w)
{
    cout << "Inside funTwo:"<<endl;
    x++;
    y = y * 2;
    w = 'G';
    cout<< "x = " << x<<endl
        << "y = " << y<<endl
        << "w = " << w << endl; //Line 17
}
int main()
{
    int num1, num2;
    char ch;
    num1 = 10;
    num2 = 15;
    ch = 'A';
    cout<< "Inside main:" <<endl
        << "num1 = " << num1 <<endl
        << "num2 = " << num2 <<endl
        << "ch = "   << ch << endl;

    funOne(num1, num2, ch);
    cout<< "After funOne: " <<endl
        << "num1 = " << num1<<endl
        << "num2 = " << num2<<endl
        << "ch = "   << ch << endl;
    funTwo(num2, 25, ch);
    cout<< "After funTwo:"<<endl
        << "num1 = " << num1<<endl
        << "num2 = " << num2 <<endl
```

```
          << "ch = "   << ch << endl;
}
```

OUTPUT:

```
Inside main:
num1 = 10
num2 = 15
ch = A
Inside funOne:
a = 11
b = 30
v = B
one = 10
After funOne:
num1 = 10
num2 = 30
ch = A
Inside funTwo:
x = 31
y = 50
w = G
After funTwo:
num1 = 10
num2 = 31
ch = G
```

- **PRACTICE:** What is the output of the following program.

```
void addFirst(int& first, int& second);
void doubleFirst(int one, int two);
void squareFirst(int& ref, int val);
int main()
{
    int num = 5;
    cout << "main: num = " << num << endl;
    addFirst(num, num);
    cout << "main after addFirst: num = " << num << endl;
    doubleFirst(num, num);
    cout << "main after doubleFirst: num = " << num << endl;
    squareFirst(num, num);
    cout << "main after squareFirst: num = " << num << endl;
}
void addFirst(int& first, int& second)
{
    cout << "Inside addFirst:"<<endl;
    cout << "first = "<< first << ", second = " << second << endl;
    first = first + 2;
    second = second * 2;
    cout << "first = "<< first << ", second = " << second << endl;
}
void doubleFirst(int one, int two)
{
    cout << "Inside doubleFirst:"<<endl;
    cout<<"one = "<< one << ", two = " << two << endl;
    one = one * 2;
    two = two + 2;
    cout<<"one = "<< one << ", two = " << two << endl;
}
void squareFirst(int& ref, int val)
{
    cout << "Inside squareFirst: "<<endl;
    cout <<"ref = "<< ref << ", val = " << val << endl;
    ref = ref * ref;
    val = val + 2;
    cout <<"ref = "<< ref << ", val = " << val << endl;
}
```

- By definition, a value-returning function returns a single value; this value is returned via the **return** statement. If a function needs to return more than one value, as a rule of good programming style, you should change it to a void function and use the appropriate reference parameters to return the values.
- **EXAMPLE 6-24:** Write the definition of a void function that takes as input two parameters of type `int`, say **sum** and **testScore**. The function updates the value of **sum** by adding the value of **testScore**. The new value of sum is reflected in the calling environment. Test the correctness of your function

```
void sumScores(int &sum, int testScore)
{
    sum += testScore;
}
int main()
{
    int score = 85;
    int totalScores = 0;
    sumScore(totalScores, score);
    cout<<totalScores;
}
```

EXAMPLE 6-25: Write the definition of a void function that takes as input two parameters of type `double`, that represents the length and width of a rectangle. The function should return the area and perimeter of the rectangle. Test the correctness of your function.

```
void AreaPermiter(double length, double width, double &area, double
&perimeter)
{
    area = length * width;
    perimeter = 2 *(length + width);

}
int main()
{
    double w = 10;
    double ar, pe;
    AreaPermiter(2*20/ 4,w, ar,pe );
    cout<<"area = "<<ar<<endl;
    cout<<"perimeter = "<<pe<<endl;
}
```

Scope of an Identifier

- **RECALL:** An identifier is the name of something in C++, such as a variable or function name.
- Identifiers are declared in a function heading, within a block, or outside a block.
- Are you allowed to access any identifier anywhere in the program?
    - The answer is no, you must follow certain rules to access an identifier.
- **The scope of an identifier** refers to where in the program an identifier is accessible (visible).
    - **Local identifier:** Identifiers declared within a function (or block), and they are not accessible outside of the function (block).
    - **Global identifier:** Identifiers declared outside of every function definition.
- In general, the following rules apply when an identifier is accessed:
    - Global identifiers (such as variables) are accessible by a function or a block if:
        1. The identifier is declared before the function definition (block).

```cpp
const double RATE = 10.50;
int z = 1;
void one(int , char );
int main()
{
    int first = 8;
    char last = 'd';
    cout<<"main: "<<RATE<<"\t"<<z<<"\t"
        <<first<<"\t"<<last<<endl;
    one(first, last);
}
int w = 9;
void one(int x, char y)
{
        cout<<"one : "<<RATE<<"\t"<<z<<"\t"
            <<x<<"\t"<<y<<"\t"<<w<<endl;
}
```

OUTPUT:

```
main: 10.5      1       8       d
one : 10.5       1       8       d       9
```

2. The function name is different than the identifier.

```cpp
void one(int x, char y);
void three(int one, double y, int z);

int main()
{
    int num = 7, first = 8;
    double x = 13.5;
    char last = 'd';
    cout<<"main : "<<num<<"\t"<<first<<"\t"
        <<x<<"\t"<<last<<endl;
    one(first, last);
    three(num,x,first);
}

void one(int x, char y)
{
    cout<<"one  : "<<x<<"\t"<<y<<endl;
}

void three(int one, double y, int z)
{
    //one(3,'2');
    cout<<"three: "<<one<<"\t"<<y<<"\t"<<z<<endl;
}
```

OUTPUT:

```
main : 7        8       13.5    d
one  : 8        d
three: 7        13.5    8
```

3. All parameters of the function have names different than the name of the identifier.
4. All local identifiers (such as local variables) have names different than the name of the identifier.

```
int z = 1;
double t = 11.5;
void one(int, char);
void two(int, int, char);

int main()
{
    int num1 = 7, num2 = 8;
    char last = 'd';
    cout<<"main: "<<z<<"\t"<<t<<"\t"
        <<num1<<"\t"<<num2<<"\t"<<last<<endl;
    one(num1, last);
    two(num1,num2, 'r');
}

void one(int x, char y)
{
    double z = 12;
    cout<<"o : "<<z<<"\t"<<t<<"\t"
        <<x<<"\t"<<y<<endl;
}
void two(int a, int b, char t)
{
    cout<<"two : "<<z<<"\t"<<t<<"\t"
```

```
        <<a<<"\t"<<b<<endl;
}
```

OUTPUT:

```
main: 1 11.5    7       8       d
one : 12        11.5    7       d
two : 1 r       7       8
```

- (Nested Block) An identifier declared within a block is accessible:
    1. Only within the block from the point at which it is declared until the end of the block.
    2. By those blocks that are nested within that block if the nested block does not have an identifier with the same name as that of the outside block (the block that encloses the nested block).

```cpp
int z = 1;
void three(int, double, int);

int main()
{
    int num1 = 7;
    double x = 13.5;
    cout<<"main   : "<<z<<"\t"<<num1<<"\t"
        <<x<<"\t"<<endl;
    three(num1,x,num1+3);
}

void three(int one, double x, int z)
{
    char ch = 'e';
    int a = 11;
    cout<<"three 1: "<<z<<"\t"<<one<<"\t"
        <<x<<"\t"<<ch<<"\t"<<a<<endl;

    { //Block Three 2
        int x = 12;
        char a = 'f';
        cout<<"three 2: "<<z<<"\t"<<one<<"\t"
            <<x<<"\t"<<a<<endl;
    }//end Block Three 2
    cout<<"three 1: "<<z<<"\t"<<one<<"\t"
        <<x<<"\t"<<ch<<"\t"<<a<<endl;
}
```

OUTPUT:

```
main   : 1      7       13.5
three 1: 10     7       13.5    e       11
three 2: 10     7       12      f
three 1: 10     7       13.5    e       11
```

- The scope of a function name is similar to the scope of an identifier declared outside any block. That is, the scope of a function name is the same as the scope of a global variable.

- Any attempt to access an identifier outside its scope will cause a compilation error.
- C++ allows the programmer to declare a variable in the initialization statement of the `for` statement, and its scope is limited only to the body of the `for` loop.

```
for (int count = 1; count < 10; count++)
    cout << count << endl;
 cout << count << endl; //error: 'count' was not declared in this scope
```

- Note the following about global variables:
  1. Usually C++ does not automatically initialize variables. However, some compilers initialize global variables to their default values. For example, if a global variable is of type `int`, `char`, or `double`, it is initialized to zero.

```
int x;
int main()
{
    int a;
    cout<<a<<" "<<x;    //4354206 0
}
```

  2. In C++, `::` is called **the scope resolution operator**. By using the scope resolution operator, a global variable declared before the definition of a function (block) can be accessed by the function (or block) even if the function (or block) has an identifier with the same name as the variable.

```
int x = 1;
int main()
{
    int x = 2;
    x++;
    ::x++;
    cout<<x<<" "<<::x;  //3 2
}
```

- **PRACTICE:** Use the scope resolution operator in the example given in the second rule to access global identifiers to call function one in function three.

Global Variables, Named Constants, and Side Effects

- A C++ program can contain global variables and you might be tempted to make all of the variables in a program global variables so that you do not have to worry about what a function knows about which variable.

- If more than one function uses the same global variable and something goes wrong, it is difficult to discover what went wrong and where. Problems caused by global variables in one area of a program might be misunderstood as problems caused in another area.
- It is not recommend to use global variables; instead, use the appropriate parameters.
- For example, consider the following program:

```
int t;
void funOne(int& a);
int main()
{
    t = 15;
    cout << "t = " << t << endl;
    funOne(t);
    cout << "after funOne: "<< " t = " << t << endl;
}
void funOne(int& a)
{
    cout << "In funOne: a = " << a
         << ", t = " << t << endl;
    a = a + 12;
    cout << "In funOne: a = " << a
         << ", t = " << t << endl;
    t = t + 13;
    cout << "In funOne: a = " << a
         << ", t = " << t << endl;
}
```

- Declaring named constants as global named constants have no side effects because their values cannot be changed during program execution. Moreover, placing a named constant in the beginning of the program can increase readability, even if it is used only in one function.

Static and Automatic Variables

- The variables discussed so far have followed two simple rules:
  1. Memory for global variables remains allocated as long as the program executes.
  2. Memory for a variable declared within a block is allocated at block entry and deallocated at block exit.
- **Automatic variables** are variables for which memory is allocated at block entry and deallocated at block exit.
  - Variables declared within a block (local variables) are automatic variables.
- **Static variables** are variables for which memory remains allocated as long as the program executes is called .
  - Global variables are static variables.
- You can declare a static variable within a block by using the reserved word `static`.
- The syntax for declaring a static variable is:

```
static dataType identifier;
```

- The following statement declares `x` to be a static variable of type `int`:

```cpp
static int x;
static int y = 2;
cout << x<<" "<<y;
```

  - Most compilers initialize static variables to their default values.
  - For example, `static` `int` variables are initialized to `0`.
  - However, it is a good practice to initialize static variables yourself, especially if the initial value is not the default value.
- Static variables declared within a block are local to the block, and their scope is the same as that of any other local identifier of that block.
- **EXAMPLE 6-26:** Study the following code snippet and answer the questions below.

```cpp
int count;
for (count = 1; count <= 5; count++)
{
    int x = count;
    x += count;
    cout<<count<<". x = "<<x<<endl;
}
```

  - How many times does the declaration statement of the variable **x** is executed?

    It will be executed every time the for-loop body is executed (5 times)
  - What is the output of the previous code snippet?

```
1. x = 2
2. x = 4
3. x = 6
4. x = 8
5. x = 10
```

- **EXAMPLE 6-27:** Study the following code snippet and answer the questions below.

```cpp
int count;
for (count = 1; count <= 5; count++)
{
    static int x = count;
    x += count;
    cout<<count<<". x = "<<x<<endl;
}
```

  - How many times does the declaration statement of the variable **x** is executed?

    It will be executed only the first time the for-loop body is executed
  - What is the output of the previous code snippet?

```
1. x = 2
2. x = 4
3. x = 7
4. x = 11
5. x = 16
```

- **EXAMPLE 6-28:** Study the following function definition and answer the questions below.

```cpp
void test()
{
    static int x = 0;
    int y = 10;
    x = x + 2;
    y = y + 1;
    cout << "Inside test x = " << x << " and y = "
         << y << endl;
}
```

  - How many times does the declaration statement of the static variable **x** is executed?

    It will be executed only in the first function call to function test
  - How many times does the declaration statement of the automatic variable **y** is executed?

    It will be executed in every function call to function test
  - What is the output of the following code snippet?

```cpp
int main()
{
    int count;
    for (count = 1; count <= 5; count++)
        test();
}
```

OUTPUT:

```
Inside test x = 2 and y = 11
Inside test x = 4 and y = 11
Inside test x = 6 and y = 11
Inside test x = 8 and y = 11
Inside test x = 10 and y = 11
```

Function Overloading

- In C++, **function overloading** (overloading a function name): is creating several functions with the same name and different formal parameter lists.
- Two functions are said to have different formal parameter lists in one of the following cases:
  - If both functions have different number of formal parameters.
  - If both functions have the same number of formal parameters but different data type of the formal parameters, in the order you list them, must differ in at least one position.
- The following functions have different formal parameter lists.

```cpp
void functionOne(int x)
void functionTwo(int x, double y)
void functionThree(double y, int x)
int functionFour(char ch, int x, double y)
int functionFive(char ch, int x, string name)
```

- The following two functions have the same formal parameter lists.

```cpp
void functionSix(int x, double y, char ch)
void functionSeven(int one, double u, char firstCh)
```

- The following two functions are overloaded functions.

```cpp
int functionXYZ(char ch, int x, double y);
double functionXYZ(char ch, int x, string name);
```

- Having multiple functions with the same signature [name + formal parameter list] will cause a compilation error.

```cpp
void functionXYZ(int x, double y)
double functionXYZ(int v, double w)
// error: ambiguating new declaration of 'double functionXYZ(int,
double)'
```

- If a function is overloaded, then in a call to that function, the signature—that is, the formal parameter list of the function—determines which function to execute.
- **EXAMPLE 6-29:** Study the following functions definitions and answer the question below.

```cpp
int larger(int x, int y)
{
    return x >=y ? x : y;
}
char larger(char x, char y)
{
    return x >=y ? x : y;
}
double larger(double x, double y)
{
    return x >=y ? x : y;
}
string larger(string x, string y)
{
    return x >=y ? x : y;
}
```

  - What is the output of the following C++ statements.

	Statement	Output
1.	`cout<<larger(3,7)<<endl;`	7
2.	`cout<<larger("hi","hello")<<endl;`	hi
3.	`cout<<larger(30.6,7.89)<<endl;`	30.6

	Statement	Output
4.	`cout<<larger('a','D')<<endl;`	a
5.	`cout<<larger(3,7.5)<<endl;`	error: call of overloaded 'larger(int, double)' is ambiguous

Functions with Default Parameters

- **RECALL:** When a function is called, the number of actual and formal parameters must be the same.
- C++ relaxes this condition for functions with default parameters.
- The syntax for formal parameter list in functions with default parameter is:

```
dataType identifier = initialValue1,
...,
dataType identifier = initialValueN_1,
dataType identifier = initialValueN
```

- You specify the value of a default parameter when the function name appears for the first time, such as in the prototype.
- The following rules apply for functions with default parameters:

  1. In the definition of a function with default parameters, all of the default parameters must be the far-right parameters of the function.

```
int Sum(int val1 = 1 ,int val2 = 1,int val3 = 1);   // correct prototype
int Sum(int val1 ,int val2 = 1,int val3 = 1);       // correct prototype
int Sum(int val1 = 1 ,int val2,int val3 = 1);       // incorrect prototype
int Sum(int val1 = 1,int val2 = 1,int val3);        // incorrect prototype
```

  2. Default values can be constants, global variables, or value returning function calls.

```
int Sum(int val1,int val2 = abs(-5),int val3 = 2);
```

  3. In the function call, if you do not specify the value of a default parameter, the default value is used for that parameter.
  4. In the function call, you can specify a value other than the default for any default parameter.

```cpp
int Sum(int val1 = 1,int val2 = 1,int val3 = 1)
{
    return val1 + val2 + val3;
}
int main()
{
    cout<<Sum()<<endl;
    cout<<Sum(3)<<endl;
    cout<<Sum(3,4)<<endl;
    cout<<Sum(3,4,5)<<endl;
}
```

OUTPUT:

```
3
5
8
12
```

5. Suppose a function has more than one default parameter. In a function call, if a value to a default parameter is not specified, then you must omit all of the arguments to its right.

```cpp
int Sum(int val1,int val2 = abs(-5),int val3 = 2)
{
    return val1 + val2 + val3;
}
int main()
{
    cout<<Sum(3)<<endl;
    cout<<Sum(3,4)<<endl;
    cout<<Sum(2,,4)<<endl;//illegal function call
}
```

OUTPUT:

```
10
9
```

6. You cannot assign a constant value as a default value to a reference parameter.

```cpp
void Fun(int val1,int &val2=1,int val3 = 2);
//illegal function declaration
```

- **EXAMPLE 6-30:** Consider the following function prototype and answer the question below.

```cpp
void funcExp(int x, int y, double t, char z = 'A', int u = 67, char v =
'G', double w = 78.34);
```

- Which of the following function calls are correct?

	Statement	Is correct?
1.	`funcExp(5, 10, 12.5);`	true
2.	`funcExp(5, 15, 34.6, 'B', 87, 'h');`	true
3.	`funcExp(14.6, 12, 14.56, 'D');`	true
4.	`funcExp(10, 15);`	false
5.	`int b = 5;` `funcExp(b, 25, 48.76, "D", 4567, 78.34);`	false

- **NOTE:** In programs in this book, and as is recommended, the definition of the function main is placed before the definition of any user-defined functions. You must, therefore, specify the default value for a parameter in the function prototype and in the function prototype only, not in the function definition because this must occur at the first appearance of the function name.
- **EXAMPLE 6-31:** What is the output of the following program.

```cpp
void funcOne(int& x, double y = 12.34, char z = 'B');
int main()
{
    int a = 23;
    funcOne(a);
    funcOne(a, 42.68);
    funcOne(a, 34.65, 'Q');
    cout<< "a = " << a << endl;
}

void funcOne(int& x, double y, char z)
{
    x = 2 * x;
    cout<< "x = " << x << ", y = "
        << y << ", z = " << z << endl;
}
```

OUTPUT:

```
x = 46, y = 12.34, z = B
x = 92, y = 42.68, z = B
x = 184, y = 34.65, z = Q
a = 184
```

CHAPTER 8: ARRAYS AND STRINGS

- **RECALL:** C++ data types fall into three categories.
    - Simple data types (integers, floating point numbers).
    - Structured data type.
    - Pointers
- **RECALL:** A data type is called simple if variables of that type can store only one value at a time.
- In structured data type, each data item is a collection of other data items. Usually, simple data types are building blocks of structured data types.
- In this chapter we will study the first structured data type that is **array**.
- **EXAMPLE 8-1:** Write a C++ program that reads five numbers, finds their sum, and then print them in reverse order.

```
int num1, num2, num3, num4, num5;
cout << "Enter five integers"<<endl;
cin >> num1 >> num2 >> num3 >> num4 >> num5;

double sum = num1 + num2 + num3 + num4 + num5;
cout <<" Sum = " << sum<<endl;

cout<<"Numbers in reverse order "
    << num5 << num4 << num3 << num2 << num1;
```

    - If you try to extend the previous program to process 100 (or more) numbers, you would have to declare 100 variables and write many `cin`, `cout`, and addition statements. Thus, for large amounts of data, this type of program is not desirable.
- When you try to process a group of variables of the same type, you should be able to specify how many variables must be declared—and their data type—with a simpler statement than the one we used earlier.
    - The data structure that lets you do this in C++ is called **array**.

Arrays

- An array is a collection of a fixed number of components all of the same data type and stored at contiguous memory locations.
- A one-dimensional array is an array in which the components are arranged in a list form.
- The general form for declaring a one-dimensional array is:

```
dataType arrayName[intExp];
```

    - `intExp` specifies the number of components in the array.
    - `intExp` is any constant expression that evaluates to a positive integer.
- **EXAMPLE 8-2:** Which of the following C++ statements are correct to declare arrays.

	Statement	Is correct?
1.	`int List1[5];`	correct
2.	`const int ARRAY_SIZE = 10;` `int List2[ARRAY_SIZE];`	correct
3.	`int List3[-10];`	error: size of array 'List3' is negative
4.	`int List4[5.5];`	error: size of array 'List4' has non-integral type

- The general form (syntax) used for accessing an array component is:

```
arrayName[indexExp];
```

  - `indexExp`, called the index, is any expression whose value is a nonnegative integer.
  - The index value specifies the position of the component in the array.
  - In C++, `[]` is an operator called the **array subscripting operator**.
- In C++, the array index starts at **0**.
- **EXAMPLE 8-3:** Study the following C++ statements and note the effect of each of them.

1.
```
int nums[5];
```

   - The statement declares an array `nums` of five components of type `int`. The components are `nums[0]`, `nums[1]`, `nums[2]`, `nums[3]`, and `nums[4]`.

`nums[0]`	`nums[1]`	`nums[2]`	`nums[3]`	`nums[4]`
??	??	??	??	??

2.
```
nums[0] = 7;
nums[2] = 10;
nums[4] = nums[0] * 3;
```

`nums[0]`	`nums[1]`	`nums[2]`	`nums[3]`	`nums[4]`
7	??	10	??	21

3.
```
int i = 3;
nums[i] = nums[4] / 5;
nums[i * 3 % 4] = 12;
```

`nums[0]`	`nums[1]`	`nums[2]`	`nums[3]`	`nums[4]`
7	12	10	4	21

4.
```
cin >> nums[2];
cout<< nums[0] <<" "<< nums[1] <<" "<< nums[2]
    <<" "<< nums[3] <<" "<< nums[4];
```

OUTPUT: Assume user input is: 5

```
5
7 12 5 4 21
```

- In C++ standard, when you declare an array, its size must be known. For example, you cannot do the following:

```
int arraySize;
cout << "Enter the size of the array: ";
cin >> arraySize;
cout << endl;
int list[arraySize];        //not allowed in earlier versions of C+
```

- Some compilers provide extensions to allow the previous code.

Array Initialization During Declaration

- Like any other simple variable, an array can be initialized while it is being declared.
- The following C++ statement declares an array of 5 integers and initialize them.

```
int List1[5] = {12, 32, 16, 23, 45};
```

- If number of initializers is greater than array size then the compiler will issue a syntax error.

```
int List1[5] = {12, 32, 16, 23, 45, 25};
// error: too many initializers for 'int [5]'
```

- When initializing arrays as they are declared, it is not necessary to specify the size of the array. The size is determined by the number of initial values in the braces. However, you must include the brackets following the array name.

```
int List2[] = {12, 32, 16, 23, 45, 25};  // array size = 6
```

- When you declare and initialize an array simultaneously, you do not need to initialize all components of the array. This procedure is called **partial initialization** of an array during declaration.
- If not all values are specified in the initialization statement, the array components for which the values are not specified are initialized to 0
- **NOTE:** The size of the array in the declaration statement does matter.

```
int List3[5] = {0};         // List3 {0, 0, 0, 0, 0}
int List4[5] = {12, 32};    // List4 {12, 32, 0, 0, 0}
```

- When you partially initialize an array, then all of the elements that follow the last uninitialized elements must be uninitialized. Therefore, the following statement will result in a syntax error:

```
int List5[10] = {2, 5, 6, , 8}; //illegal
```

Processing One-Dimensional Arrays

- Some of the basic operations performed on a one-dimensional array are initializing, inputting data, outputting data stored in an array, and finding the largest and/or smallest element.
- If the data is numeric, some other basic operations are finding the sum and average of the elements of the array.
- Each of these operations requires the ability to step through the elements of the array. This is easily accomplished using a loop.

```
for (int i = 0; i < SIZE; i++)
    //process list[i]
```

- Consider the following definition of array **sales** that has 10 `double` numbers.

```
double sales[10];
```

Initializing an array

- The following loop initializes every component of the array **sales** to 0.0.

```
for (int index = 0; index < 10; index++)
    sales[index] = 0.0;
```

Reading data into an array

- The following loop inputs the data into the array **sales** from the keyboard.

```
for (int index = 0; index < 10; index++)
    cin >> sales[index];
```

Printing an array

- The following loop outputs the components of array **sales** to the screen.

```
for (int index = 0; index < 10; index++)
    cout << sales[index] << " ";
```

Copy one array into another array

- To copy an array into another array, you must copy it component-wise—that is, one component at a time.
- The following program copies the content of array **sales** into array **newSales**.

```
double newSales[10];
for (int index = 0; index < 10; index++)
    newSales[index] = sales[index];
```

- **PRACTICE:** Write the required C++ program to check if two arrays are equal (has the same elements in the same order).

Finding the sum and average of an array

- Because the array **sales** has numeric data, you can calculate the total sales and average sales amounts.
- The following C++ code finds the sum and average of all elements of array **sales**:

```
double sum = 0;
for (int index = 0; index < 10; index++)
    sum = sum + sales[index];

double average = sum / 10;
```

Largest element in the array

- The following C++ code finds the **first occurrence** of the largest element in the array **sales**—that is, the first array component with the largest value.

```
double maxValue = sales[0];
for (int index = 1; index < 10; index++)
    if (sales[index] > maxValue)
        maxValue = sales[index];

cout<<"largest sale = "<< maxValue;
```

- To determine the index of the first occurrence of the largest element in an array use the following code.

```cpp
int maxIndex = 0;   // store the index of the first occurrence
                    // of the largest element in the array

for (index = 1; index < 10; index++)
    if (sales[maxIndex] > sales[index])
        maxIndex = index;

double largestSale = sales[maxIndex];
```

- **PRACTICE:** Write the required code to find the index of the last occurrence of the smallest element in the array **sales**

- **EXAMPLE 8-4:** Write a C++ program to read five test scores, finds the average test score, and print all the test scores that are less than the average test score.

```cpp
int test[5];
int sum = 0;
double average;
int index;
cout << "Enter five test scores: "<<endl;
for (index = 0; index < 5; index++)
{
    cin >> test[index];
    sum = sum + test[index];
}
average = sum / 5.0;

cout << "The average test score = " << average << endl;

cout << "Test scores less than the average test score." << endl;
for (index = 0; index < 5; index++)
    if (test[index] < average)
        cout << test[index]<<"\t";
```

OUTPUT: Assume user input is 70 83 89 78 90

```
Enter five test scores:
70 83 89 78 90
The average test score = 82
Test scores less than the average test score.
70      78
```

- The index of an array is in bounds if `index >= 0 and index <= ARRAY_SIZE - 1`.
- Array Index is said to be out of bounds if either `index < 0 or index > ARRAY_SIZE - 1`, then we say that the index is out of bounds.
- Unfortunately, in C++, there is no guard against out-of-bound indices.
  - If the index goes out of bounds and the program tries to access the component specified by the index, then whatever memory location is indicated by the index that location is accessed.

- This situation can result in altering or accessing the data of a memory location that you never intended to modify or access
- It is solely the programmer's responsibility to make sure that the index is within bounds.

- **EXAMPLE 8-5:** Study the output of the following code snippet.

```
int i;
int List[5];
for (i = 0; i < 5; i++)
    List[i] = 10;

for (i = 0; i <= 6; i++)
    cout<<List[i]<<" ";
```

OUTPUT:

```
0 0 0 0 0 5 4354144
```

Searching an Array for a Specific Item

- Searching a list for a given item is one of the most common operations performed on a list.
- In this chapter we will use the sequential search or linear search algorithm to search an array for a specific element.
- The sequential search algorithm search the array sequentially as follows:
  1. Start from the first array element.
  2. Compare search item with the elements in the array (the list)
  3. Repeat step 2 until either you find the item or no more data is left in the list to compare with search item.
- The pseudocode for sequential search algorithm.

```
found is set to false
loc = 0;
while (loc < listLength and not found)
    if (list[loc] is equal to searchItem)
        found is set to true
    else
        increment loc

if (found)
    print element found at location loc
else
    print element not found
```

- **EXAMPLE 8-6:** Write C++ program to read a number from the user and determine whether it is on the array `List` or not. if the number is found in the list print its location otherwise print a proper message.

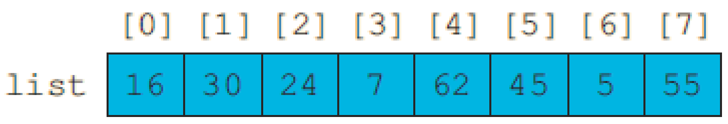
```
const int LIST_SIZE = 5;
int List[LIST_SIZE] = {12,4,34,27,44};
int searchItem;
```

```
cout << "Enter a number to search for"<<endl;
cin >> searchItem;

bool found = false;
int loc = 0;
while (loc < LIST_SIZE && !found)
    if (List[loc] == searchItem)
        found = true;
    else
        loc++;

if (found)
    cout<<"Element found at location "<< loc<<endl;
else
    cout<<"Element not found"<<endl;
```

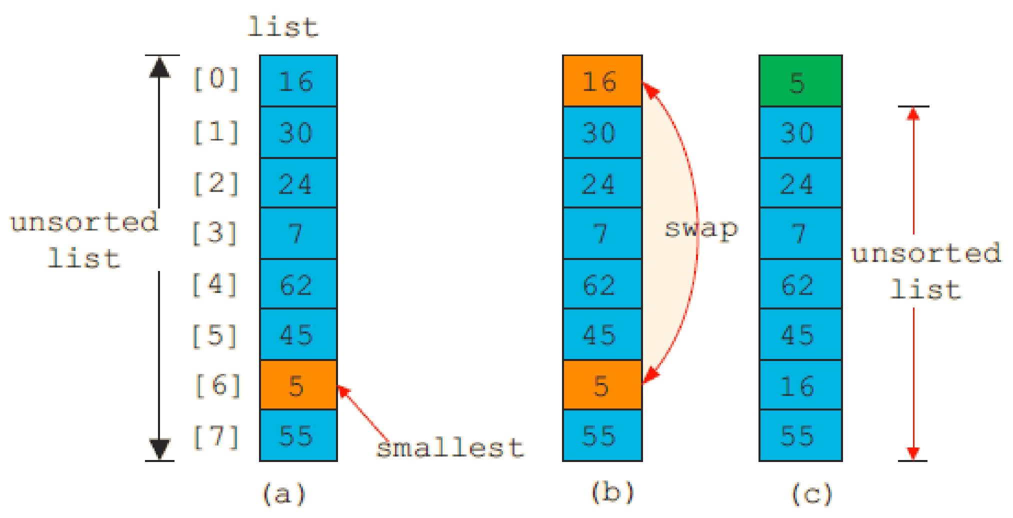
- **PRACTICE:** Rewrite the sequential search program using loop and **break** statement.

Selection Sort

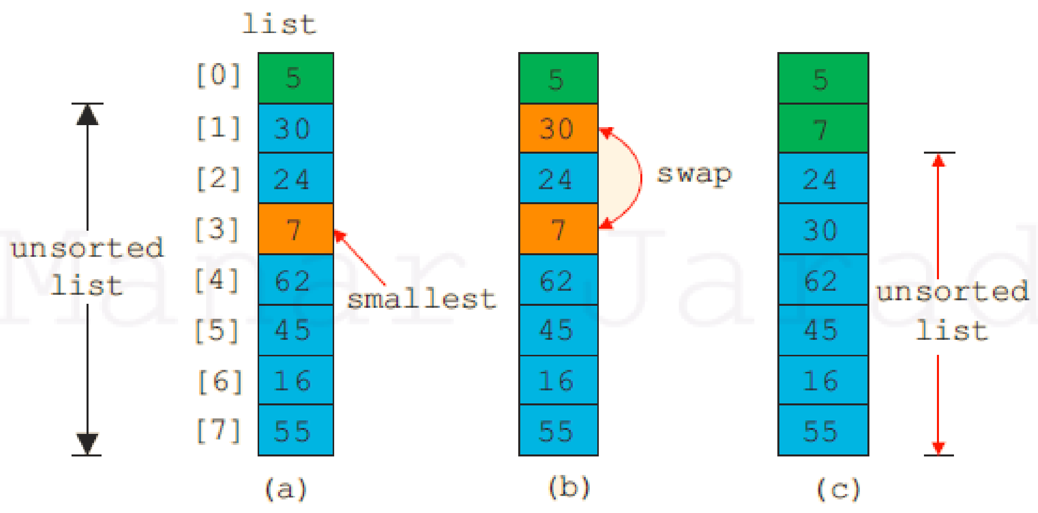
- In this section, we discuss how to sort an array using an algorithm, called **selection sort**.
- The selection sort algorithm, rearrange the list by selecting an element in the list and moving it to its proper position.
- Selection sort thus involves the following steps.
  1. Find the location of the smallest element In the **unsorted portion** of the list.
  2. Move the smallest element to the beginning of the unsorted list.
- **EXAMPLE 8-7:** Consider the following example that discuss the selection sort algorithm.
  1. Initially, the entire list is unsorted.



  2. Find the smallest item in the list. Move it from position 6 to position 0. So, we swap 16 (that is, `list[0]`) with 5 (that is, `list[6]`), as shown in the figure below.



3. Now the unsorted list is `list[1]`... `list[7]`. So, we find the smallest element in the unsorted list. Move it from position 3 to position 1. So, we swap 30 (that is, `list[1]`) with 7 (that is, `list[3]`), as shown in the figure below.



4. Now, the unsorted list is `list[2]`... `list[7]`. So, we repeat the preceding process of finding the (position of the) smallest element in the unsorted portion of the list and moving it to the beginning of the unsorted portion of the list.

- The following is the pseudocode for **selection sort** algorithm:

```
for (index = 0; index < length - 1; index++)
{
    a. Find the location, smallestIndex, of the smallest element in
        list[index]...list[length - 1].
    b. Swap the smallest element with list[index]. That is, swap
        list[smallestIndex] with list[index].
}
```

- The following program implements the selection sort algorithm:

```
int smallestIndex;
int location;
int temp;

int list[]= {2, 56, 34, 25, 73, 46, 89, 10, 5, 16};
for (int index = 0; index < 9; index++)
{
    //Step a
    smallestIndex = index;
    for (location = index + 1; location < length; location++)
        if (list[location] < list[smallestIndex])
            smallestIndex = location;

    //Step b
    temp = list[smallestIndex];
    list[smallestIndex] = list[index];
    list[index] = temp;
}
```

Base Address of an Array and Array in Computer Memory

- **RECALL:** An *array* is a sequence of components of the same type placed in contiguous memory locations.
- The **base address** of an array is the address (that is, the memory location) of the first array component.
- Consider the following statements:

```
int List1[5] = {0, 4, 8, 12, 16};
```

  - This statement declares `List1` to be an array of five components of type `int`.
  - The computer allocates five contiguous memory spaces, each large enough to store an `int` value, for these components.
  - The base address of the array `List1` is the address of the component `List1[0]`.
  - There is also a memory space associated with the identifier `List1`, and the base address of the array is stored in that memory space and it cannot be changed during program execution (constant).

Address		**330**	**331**	**332**	**333**	**334**			**501**
Variable		List1[0]	List1[1]	List1[2]	List1[3]	List1[4]			List1
Content		0	4	8	12	16			330

- Consider the following statement:

```
cout << List1 << endl;
```

  - This statement outputs the value of `List1`, which is the base address of the array. This is why the statement will not generate a syntax error.
- Consider the following code snippet:

```cpp
int List2[5];
if (List1 <= List2)
    //Action statement
```

  - The expression `List1 <= List2` evaluates to true if the base address of the array `List1` is less than the base address of the array `List2`; and evaluates to false otherwise.

- When you declare an array, the only things about the array that the computer remembers are:
  - Its name
  - Its base address.
  - The data type of each component
- Using the base address of the array and the index of an array component, the computer determines the address of a particular component, so it can access its content directly.
- When you declare an array, the base address of this array is stored in a memory space whose name is the same as the array name. The content of this memory space is **constant** and any attempt to modify it will cause the compiler to generate a syntax error.
- C++ does not allow aggregate operations on an array.
- An **aggregate operation** on an array is any operation that manipulates the entire array as a single unit.
  - Copy an array into another array using assignment operator.
  - Read data into array using input statement .
  - Determine whether two arrays have the same elements using equality operator or compare array elements using relational operators.
  - Print the contents of an array using output statement .

- **EXAMPLE 8-8:** Study the following code snippet and identify the line that will cause a syntax error.

```cpp
1  int myList[5] = {0, 4, 8, 12, 16};
2  int yourList[5];
3  cout<<(myList == yourList);
4  yourList = myList;
5  cin >> yourList;
```

  - Statement at line 4 and 5 will cause syntax error.
  - Statement at line 3 is illegal in the sense that it <u>dose not generate a syntax error;</u> however, they do not give the desired results.

C-Strings (Character Arrays)

- **Character array**: An array whose components are of type char.
- Character arrays are of special interest, and you process them differently than you process other arrays.
- The first character in the ASCII character set is the **null** character, which is **nonprintable** character, and it plays an important role in processing character arrays.
- **RECALL:** In C++, the null character is represented as `'\0'`, a backslash followed by a zero.
- The most commonly used term for character arrays is **C-strings**.

- There is a subtle difference between character arrays and C-strings.
    - C-strings are null terminated character array; that is, the last character in a C-string is always the null character.

    ```cpp
    char name1[] = {'A','h','m','a','d','\0'};  //Array size = 6
    char name2[] = "Omar";                      //Array size = 5
    ```

    - A character array might not contain the null character.

    ```cpp
    char name1[] = {'A','h','m','a','d'};       //Array size = 5
    char name2[] = "Omar";                      //Array size = 5
    ```

- From the definition of C-strings, it is clear that there is a difference between `'A'` and `"A"`.
    - `'A'` is character A. To store 'A', we need only one memory cell of type char;
    - `"A"` is C-string A, to store "A", we need two memory cells of type char—one for 'A' and one for `'\0'`.
- The following statement declares an array `name` of 8 components of type `char`.

```cpp
char name[8];
```

- Because C-strings are null terminated and `name` has 8 components, the largest string that can be stored in name is of length 7.
- If you store a C-string of length 4 in `name`, the first 4 components of `name` are used and the last 4 are set to `'\0'`.

```cpp
char name[8] = "Omar";
```

name[0]	name[1]	name[2]	name[3]	name[4]	name[5]	name[6]	name[7]
'O'	'm'	'a'	'r'	'\0'	'\0'	'\0'	'\0'

- Most rules that apply to other arrays also apply to character arrays.
    - Using aggregate operations, such as assignment and comparison, are also not allowed on character arrays.

    ```cpp
    char name1[20];
    char name2[20];
    name1 = "Ahmad Ali";    // error: incompatible types in assignment of
    // 'const char [10]' to 'char [20]'
    cout<<(name1 < name2);  // legal statement but it does not give the
    // desired results.
    ```

- **EXAMPLE 8-9:** Write the required C++ program to find the length of a C-string.

```cpp
char str[20] = "Good Day";
int length = 0;
while(str[length] != '\0')
    length++;

cout << "length = "<<length;
```

- **EXAMPLE 8-10:** Write the required C++ program to compare two C-strings and set a `result` variable to -1 if the first C-string is less than the second C-string, 1 if the first C-string is greater than the second C-string and 0 if the two C-strings are the same.

```cpp
char str1[10] = "Ahmad";
char str2[10] = "Ali";
int result = 0;
for(int i = 0; i < 10 ;i++)
{
    if(str1[i] > str2[i])
    {
        result = 1;
        break;
    }
    else if(str1[i] < str2[i])
    {
        result = -1;
        break;
    }
}
cout << "result = "<<result<<endl;
```

- **EXAMPLE 8-11:** Use the value stored in the variable `result` from the program to print the value of the greatest string.

```cpp
if(result > 0)
    cout<< str1 <<" is greater than "<<str2<<endl;
else if (result < 0)
     cout<< str1 <<" is less than "<<str2<<endl;
else
     cout<< str1 <<" is equal "<<str2<<endl;
```

- **PRACTICE:** Write the required C++ program to copy a C-string into another C-string variable.
- C++ provides a set of functions that can be used for C-string manipulation. The header file `cstring` describes these functions.
    - `strcpy` (string copy): Copy a C-string into another C-string variable—that is, assignment.
    - `strcmp` (string comparison): Compare C-strings.
    - `strlen` (string length): Find the length of a C-string.
- The following table summarize the previous functions

Function	Effect
`strcpy (s1, s2)`	Copies the string s2 into the string variable s1 The length of s1 should be at least as large as s2
`strcmp (s1, s2)`	Returns a value< 0 if s1 is less than s2 Returns 0 if s1 and s2 are the same Returns a value > 0 if s1 is greater than s2
`strlen (s)`	Returns the length of the string s, excluding the null character

- To use these functions, the program must include the header file `cstring` via the include statement.

```
#include <cstring>
```

- **EXAMPLE 8-12:** What is the output of the following C++ program.

```
char studentName[21];
char myname[16];
char yourname[16];

strcpy(myname, "John Robinson");
cout<< myname<<" has ";
cout<< strlen(myname) <<" letters"<<endl;

int len = strlen("Sunny Day");
cout<<"Sunny Day has "<< len <<" letters"<<endl;

strcpy(yourname, "Lisa Miller");
strcpy(studentName, yourname);
cout<<"Student name = "<<studentName<<endl;
cout<<"Your name = "<<yourname<<endl;

cout<< strcmp("Bill", "Lisa")<<endl;

strcpy(yourname, "Kathy Brown");
strcpy(myname, "Mark G. Clark");
cout<<strcmp(myname, yourname)<<endl;
```

OUTPUT:

```
John Robinson has 13 letters
Sunny Day has 9 letters
Student name = Lisa Miller
Your name = Lisa Miller
-1
1
```

- **EXAMPLE 8-13:** Write a C++ program to compare two C-strings and print the greatest one.

```cpp
char str1[10] = "Ahmad";
char str2[10] = "Ali";

if(strcmp(str1, str2) > 0)
    cout<< str1 <<" is greater than "<<str2<<endl;
else if (strcmp(str1, str2) < 0)
     cout<< str1 <<" is less than "<<str2<<endl;
else
     cout<< str1 <<" is equal "<<str2<<endl;
```

Reading and Writing Strings

- Aggregate operations, such as assignment and comparison, are not allowed onto C-strings as well as arrays.
- The input/ output of arrays is done component-wise. However, the one place where C++ allows aggregate operations on arrays is the input and output of C-strings (that is, character arrays).

String Output

- **RECALL:** The null character should not appear anywhere in the C-string except the last position.
- Because aggregate operations are allowed for C-string output, you can output C-strings by using an output statement `cout <<`.

```cpp
char name[10] = "Ahmad";
cout << name;
```

- The insertion operator, `<<`, continues to write the contents of a C-string until it finds the null character.
- If a C-string does not contain the null character, then you will see strange output because the insertion operator continues to output data from memory adjacent to name until `'\0'` is found.
- **EXAMPLE 8-14:** What is the output of the following code segment

  -
    ```cpp
    char name[10] = "Ahmad Ali";
    cout << name;                  //Ahmad Ali
    ```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]			
'A'	'h'	'm'	'a'	'd'	' '	'A'	'l'	'i'	'\0'	'3'	'Z'	'\0'

  -
    ```cpp
    name[9] = '*';
    cout << name;                  //Ahmad Ali*3Z
    ```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]			
'A'	'h'	'm'	'a'	'd'	' '	'A'	'l'	'i'	'*'	'3'	'Z'	'\0'

  - **PRACTICE:** What is the output of the following code snippet?

```cpp
char name[10] = "Ahmad Ali";
name[4] = 0;
cout << name;
```

String Input

- Because aggregate operations are allowed for C-string input, the input statement `cin >>` can be used to store the next input in a C-string.

  ```cpp
  char name[20];
  cin >> name;
  ```

  - The length of the input C-string must be less than 20.
  - If the length of the input string is is less than the character array size, the computer stores the input and the null character `'\0'` after it.
  - In C++ there is no check on the array index bounds, so if the input is greater than the character array size, the computer continues storing the string in whatever memory cells follow name.
    - This process can cause serious problems, because data in the adjacent memory cells will be corrupted.
- **RECALL:** The extraction operator, `>>`, skips all leading whitespace characters and stops reading data into the current variable as soon as it finds the first whitespace character or invalid data.
- **EXAMPLE 8-15:** What is the output of the following code segment

  ```cpp
  char name1[5];
  char name2[5];
  cout << "Enter your name ... "<<endl;
  cin  >> name1 >> name2;
  cout<<name1<<endl<<name2;
  ```

  - What is the output assume user input is Rana Ali

    ```
    Enter your name ...
    Rana Ali
    Rana
    Ali
    ```

  - What is the output Assume user input is Adel Ibrahim

    ```
    Adel Ibrahim
    im
    Ibrahim
    ```

    - Can you explain the previous result.

- The function `get` can be used to **input C-strings with blanks** into a character array.

- To read any character including white spaces, use the form of the function get that takes one parameter of type character.

```cpp
char ch;
cin.get(ch);
```

- To read C-strings, use the form of the function get that has the following two parameters.
  - The first parameter is a C-string variable;
  - The second parameter specifies how many characters to read into the string variable.
- To read C-strings, the general form (syntax) of the get function is:

```cpp
cin.get(str, m + 1);
```

  - This statement stores the next `m` characters, or all characters until the newline character `'\n'` is found, into **str**.
  - If the input C-string has fewer than `m` characters, then the reading stops at the newline character.
- **EXAMPLE 8-16:** Study the following code snippet, and answer the questions below.

```cpp
char str1[10];

cout <<"Enter a string..."<<endl;
cin.get(str1, 10);
cout <<"str1 = "<<str1<<"\t";
```

  - What is the output? Assume user input is Good Evening

```
Enter a string...
Good Evening
str1 = Good Even
```

- **NOTE:** if you are planning to use `get` function multiple times in your program, then you must read and discard the newline character `'\n'` after each time you use the function `get`.
- **EXAMPLE 8-17:** Study the following code snippet, and answer the questions below.

```cpp
char str1[10];
char str2[10];
char discard;
cout <<"Enter a string..."<<endl;
cin.get(str1, 10);
cin.get(discard);
cin.get(str2, 10);

cout <<"str1 = "<<str1<<"\t";
cout <<discard;
cout <<"str2 = "<<str2<<"\t";
```

  - What is the output? Assume user input is

    Good

Evening

```
Enter a string...
Good
Evening
str1 = Good
str2 = Evening
```

- What is the output? Assume user input is Good Evening

```
Enter a string...
Good Evening
str1 = Good Even        i        str2 = ng
```

- Comment `cin.get(discard);` statement and execute the previous code for the same input and note the output each time.

- To read and store a line of input, including whitespace characters, you can also use the stream function `getline`.

```
char textLine[100];
cin.getline(textLine, 100);
```

  - The previous program will read and store the next 99 characters, or until the newline character, into **textLine**.
  - The null character will be automatically appended as the last character of **textLine**.

Parallel Arrays

- Two (or more) arrays are called parallel if their corresponding components hold related information.
- **EXAMPLE 8-18:** The following table has the id numbers of 5 students together with their scores in computer programming course.

ID	Score
202101	53
202102	90
202103	87
202104	69
202105	75

  - Write a C++ program to keep track of students' Id numbers, and their scores in computer programming course and then perform the following:
    1. Print the id number and the test score for each student in a separate line.
    2. Print the Id numbers for students who get a score less than 70.

3. Create a new array to store students grades in computer programming course. (to calculate the grade use the algorithm in example 1-5)
4. Print the id number, the test score , and grades for each student in a separate line.

```cpp
// Arrays declaration
const int NO_OF_STUDENTS = 5;
int studentId[NO_OF_STUDENTS]={202101, 202102, 202103, 202103,202104};
int courseScore[NO_OF_STUDENTS] = {53, 90, 87,69,75};
int index;

// 1- Print info
cout<<"Part 1"<<endl;
for(index = 0; index < NO_OF_STUDENTS; index++)
    cout<<setw(10)<<studentId[index]<<setw(10)<<courseScore[index]
<<endl;

// 2- Print Ids for students who get a score less than 70.
for(index = 0; index < NO_OF_STUDENTS; index++)
    if(courseScore[index] < 70)
        cout<<setw(10)<<studentId[index]<<setw(10)
            <<courseScore[index]<<endl;


// 3- Calculate Grades
char courseGrade[NO_OF_STUDENTS];
for(index = 0; index < NO_OF_STUDENTS; index++)
    if(courseScore[index] >= 90)
        courseGrade[index] = 'A';
    else if(courseScore[index] >= 80)
         courseGrade[index] = 'B';
    else if(courseScore[index] >= 70)
         courseGrade[index] = 'C';
    else if(courseScore[index] >= 60)
         courseGrade[index] = 'D';
    else
         courseGrade[index] = 'F';

// 4- Print info
cout<<endl<<"Part 4"<<endl;
for(index = 0; index < NO_OF_STUDENTS; index++)
    cout<<setw(10)<<studentId[index]
    <<setw(10)<<courseScore[index]
    <<setw(10)<<courseGrade[index]<<endl;
```

- **NOTE:** to use `setw` function you have to include `iomanip` header file.

Two- and Multidimensional Arrays

- If the data is provided in a list form, you can use one-dimensional arrays.
- If the data is provided in a table form you can use multidimensional arrays.
- For example, suppose that you want to track the number of cars in a particular color that are in stock at a local dealership. The dealership sells 4 types of cars in five different colors.
- The following table shows sample data.

	Red	Brown	Black	White	Gray
Ford	10	7	12	10	4
Toyota	18	11	15	17	10
BMW	12	6	9	5	12
Mercedes	10	9	12	6	4

  - You can see that the data is in a table format.
  - The table has 20 entries, and every entry is an integer.
  - Because the table entries are all of the same type, you can declare a one-dimensional array of 20 components of type `int`. In other words, you can simulate the data given in a table format in a one-dimensional array.
  - If you do so, the algorithms to manipulate the data in the one-dimensional array will be somewhat complicated, because you must know where one row ends and another begins.
- **Two-dimensional array:** A collection of a fixed number of components arranged in rows and columns (that is, in two dimensions), where all components are of the same type.
- The syntax for declaring a two-dimensional array is:

```
dataType arrayName[intExp1][intExp2];
```

  - `intExp1` and `intExp2` are constant expressions yielding positive integer values.
  - `intExp1`, specify the number of rows in the 2D array
  - `intExp2`, specify the number of columns in the 2D array.
- To access the components of a two-dimensional array, you need a pair of indices: one for the row position and one for the column position.
- The syntax to access a component of a two-dimensional array is:

```
arrayName[indexExp1][indexExp2]
```

  - `indexExp1` and `indexExp2` are expressions yielding positive integer values.
  - `indexExp1` specifies the row position.
  - `indexExp2` specifies the column position.
- **EXAMPLE 8-19:** Study the following C++ statements and note the effect of each of them.

  1.
     ```
     int cars[4][5];
     ```
     - The statement declares a two-dimensional array **cars** of 4 rows and 5 columns, in which every component is of type `int`.
     - The components in the first row are `cars[0][0]`, `cars[0][1]`, `cars[0][2]`, `cars[0][3]`, and `cars[0][4]`.

	[0]	**[1]**	**[2]**	**[3]**	**[4]**
[0]	??	??	??	??	??
[1]	??	??	??	??	??
[2]	??	??	??	??	??
[3]	??	??	??	??	??

2.

```cpp
cars[1][1] = 11;
cars[1][4] = cars[3][0] = 10;
cars[0][2] = cars[2][0] = cars[2][4] = 12;
```

	[0]	[1]	[2]	[3]	[4]
[0]	??	??	**12**	??	??
[1]	??	**11**	??	??	**10**
[2]	**12**	??	??	??	**12**
[3]	**10**	??	??	??	??

3.

```cpp
int i = 3;
int j = 4;
cars[i][j] = j;
```

	[0]	[1]	[2]	[3]	[4]
[0]	??	??	**12**	??	??
[1]	??	**11**	??	??	**10**
[2]	**12**	??	??	??	**12**
[3]	**10**	??	??	??	**4**

Two-Dimensional Array Initialization During Declaration

- Two-dimensional arrays can be initialized when they are declared.
- To initialize a two-dimensional array when it is declared:
  - The elements of each row are enclosed within curly braces `{ }`, and separated by commas.
  - All rows are enclosed within curly braces `{ }`.
  - For number arrays, if all components of a row are not specified, the unspecified components are initialized to 0. In this case, at least one of the values must be given to initialize all the components of a row.
- **EXAMPLE 8-20:** Study the following C++ statements and note the effect of each of them.

```cpp
int board1[4][3] = {{2, 3, 1}, {15, 25, 13},
                    {20, 4, 7},{11, 18, 14}};

int board2[4][3] = {2, 3, 1, 15, 25, 13, 20, 4, 7, 11, 18, 14};
```

  - The previous statements declare two-dimensional arrays **board1** and **board2** of 4 rows and 3 columns, both initialized to the same values.

	[0]	[1]	[2]
[0]	2	3	1
[1]	15	25	13
[2]	20	4	7
[3]	11	18	14

- **EXAMPLE 8-21:** Study the following C++ statement and note the effect of each of them.

```
int board3[3][4] = {{2, 3}, {15, 25, 13},
                    {20, 4, 7, 11}};
```

  - The previous statement declares a two-dimensional array **board3** of 3 rows and 4 columns, both initialized to the same values.

	[0]	[1]	[2]	[3]
[0]	2	3	0	0
[1]	15	25	13	0
[2]	20	4	7	11

PROCESSING TWO-DIMENSIONAL ARRAYS

- A two-dimensional array can be processed in three ways:
  1. Process the entire array.
  2. Process a particular row of the array, called row processing.
  3. Process a particular column of the array, called column processing.
- Initializing and printing the array are examples of processing the entire two-dimensional array.
- Finding the largest element in a row (column) or finding the sum of a row (column) are examples of row (column) processing.
- We will use the following declaration for our discussion:

```
const int NUMBER_OF_ROWS = 4;       //This can be set to any number.
const int NUMBER_OF_COLUMNS = 5;    //This can be set to any number.
int matrix[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
int row;
int col;
int sum;
int largest;
int temp;
```

- Each row and each column of a two-dimensional array is a one-dimensional array. Therefore, when processing a particular row or column of a two-dimensional array, we use algorithms similar to those that process one-dimensional arrays.

- You can use the following for loop to process the third row of matrix (index = 2):

```
row = 2;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    process matrix[row][col]
```

- You can use the following for loop to process the second column of matrix (index = 1):

```
col = 1;
for (row = 0; row < NUMBER_OF_ROWS; row++)
    process matrix[row][col]
```

Initialization

- You can use the following nested for loops to initialize the entire matrix to 0:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        matrix[row][col] = 0;
```

- To initialize the second row (index = 1) of matrix to 0.

```
row = 1;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    matrix[row][col] = 0;
```

Print

- You can use the following nested for loops to print the components of matrix, one row per line:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cout << setw(5) << matrix[row][col] << " ";
    cout << endl;
}
```

Input

- You can use the following nested for loops to input data into each component of matrix:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cin >> matrix[row][col];
```

- You can use the following for loop to input data into all components in the third row (index = 2) of matrix:

```cpp
row = 2;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    cin >> matrix[row][col];
```

Sum by Row

- You can use the following for loop to find the sum of row number 3 of matrix; that is, it adds the components of row number 3:

```cpp
sum = 0;
row = 3;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    sum = sum + matrix[row][col];
```

- **EXAMPLE 8-22:** Write the required C++ code to find the sum of each row separately and store them in an array.

```cpp
//Sum of each individual row
int sum[NUMBER_OF_ROWS]={0};
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        sum[row] = sum[row] + matrix[row][col];
    cout << "Sum of row " << row + 1 << " = " << sum[row] << endl;
}
```

- **PRACTICE:** Write the required C++ code to find the sum of each column separately and store them in an array.

Largest Element in Each Row and Each Column

- You can use the following for loop to determine the largest element in row number 1:

```cpp
row = 1;
largest = matrix[row][0];   //Assume that the first element of
                            //the row is the largest.
for (col = 1; col < NUMBER_OF_COLUMNS; col++)
    if (largest < matrix[row][col])
        largest = matrix[row][col];
```

- **PRACTICE:** Write the required C++ program to determine the largest element in each row and print them.

Arrays of Strings

- Strings in C++ can be manipulated using either the data type string or character arrays (C-strings).
- This section illustrates both ways to manipulate a list of strings.

Arrays of Strings and the string Type Processing

- You can declare an array of 100 components of type string as follows:

```
string list[100];
```

  - Basic operations, such as assignment, comparison, and input/output, can be performed on **values of the string type**.
- **EXAMPLE 8-23:** What is the output of the following code segment.

```
string names[5] = {"Ahmad", "Aya", "Ali", "Omar"};
names[4] = names[1];
names[1] = "Rana";

cout<<"Enter your full name"<<endl;
getline(cin, names[3]);

cout<<"Enter your friend first name"<<endl;
cin >> names[0];

for (int i = 0; i < 5; i++)
    cout<<endl<<names[i];

for (int i = 0; i < 5; i++)
    cout<<endl<<names[i].length();
```

SAMPLE RUN:

```
Enter your full name
Manar Jaradat
Enter your friend first name
Ayat

Ayat
Rana
Ali
Manar Jaradat
Aya
```

Arrays of Strings and C-Strings (Character Arrays)

- Suppose that the largest string (for example, name) in your list is 15 characters long and your list has 100 strings.
- You can declare a two-dimensional array of characters of 100 rows and 16 columns as follows

  ```
  char list[100][16];
  ```

  - `list[j]` for each `j`, `0 <= j <= 99`, is a C-string of at most 15 characters in length.
- You can use C-string functions (such as `strcmp` and `strlen`) and for loops to manipulate rows in the two-dimensional array of characters.
- **EXAMPLE 8-24:** What is the output of the following code segment.

  ```
  char names[5][20] = {"Ahmad", "Aya", "Ali", "Omar"};
  char discard;
  strcpy(names[4], names[1]);
  strcpy(names[1], "Rana");

  cout<<"Enter your full name"<<endl;
  cin.get(names[3],20);
  cin.get(discard);

  cout<<"Enter your friend first name"<<endl;
  cin >> names[0];

  for (int i = 0; i < 5; i++)
      cout<<endl<<names[i];

  for (int i = 0; i < 5; i++)
      cout<<endl<<strlen(names[i]);
  ```

  SAMPLE RUN:

  ```
  Enter your full name
  Manar Jaradat
  Enter your friend first name
  Ayat

  Ayat
  Rana
  Ali
  Manar Jaradat
  Aya
  ```

- **NOTE:** The data type string has operations such as assignment, concatenation, and relational operations defined for it. If you use Standard C++ header files and the data type string is available on your compiler, we recommend that you use the data type string to manipulate lists of strings.

Arrays as Parameters to Functions

- In C++, arrays are passed as parameters to functions by reference only.
- Because arrays are passed by reference only, you do not use the symbol `&` when declaring an array as a formal parameter.
- When declaring a one-dimensional array as a formal parameter, the size of the array is usually omitted.
- If you specify the size of a one-dimensional array when it is declared as a formal parameter, the size is ignored by the compiler.
- The prototype for a function that takes an array as parameter is:

```
void funcArrayAsParam(double listOne[]);
```

  - **listOne**, a one-dimensional array of type `double`.
- Usually when we declare an array as a formal parameter, we declare another formal parameter specifying the number of elements in the array, as in the following function:

```
void funcArrayAsParam(double listOne[], int size);
```

  - **listOne**, a one-dimensional array of type `double` .
  - **size**, an `int` variable that specifies the size of the array **ListOne**
- When you pass an array as a parameter, the base address of the actual array is passed to the formal parameter.

- The function call for the previous function is:

```
const int SIZE = 5;
double myList[SIZE] = {3,4,5,6,7};
funcArrayAsParam(myList, SIZE);
```

  - In this statement, the base address of **myList** is passed to the formal parameter list.
  - Both **myList** and **listOne** modify the same memory locations.
- **EXAMPLE 8-25:** What is the output of the following program?

```
void fillArray(int list[], int listSize)
{
    int index;
    cout<<"Enter "<<listSize<<" integers"<<endl;
    for (index = 0; index < listSize; index++)
        cin >> list[index];
}

void printArray(int list[], int listSize)
{
    int index;
    for (index = 0; index < listSize; index++)
        cout << list[index] << " ";
    cout<<endl;
}

void copyArray(int list1[], int src, int list2[],
               int tar, int numOfElements)
```

```
{
    for (int index = tar; index < tar + numOfElements; index++)
    {
        list2[index] = list1[src];
        src++;
    }
}

int main()
{
    int myList[5];
    int yourList[5]= {0};
    fillArray(myList,5);
    cout<<"After calling fillArray function"<<endl;
    printArray(myList,5);
    copyArray(myList,1,yourList,2,3);
    cout<<"After calling copyArray function"<<endl;
    printArray(yourList,5);
}
```

OUTPUT:

```
Enter 5 integers
4 8 7 6 9
After calling fillArray function
4 8 7 6 9
After calling copyArray function
0 0 8 7 6
```

- Array elements can be passed to functions either by value or by reference.
- **EXAMPLE 8-26:** What is the output of the following program?

```
void swap(int , int &);
void printArray(int list[], int);

int main()
{
    int myList[5] = {2,4,6,8,10};
    swap(myList[0],myList[3]);
    cout<<"After calling swap function"<<endl;
    printArray(myList,5);
}
void swap(int x, int &y)
{
    int tmp = x;
    x = y;
    y = tmp;
}

void printArray(int list[], int listSize)
{
    int index;
    for (index = 0; index < listSize; index++)
        cout << list[index] << " ";
    cout<<endl;
}
```

OUTPUT:

```
After calling swap function
2 4 6 2 10
```

Constant Arrays as Formal Parameters

- **RECALL:** When a formal parameter is a reference parameter, then whenever the formal parameter changes, the actual parameter changes as well.
- Even though an array is always passed by reference, you can still prevent the function from changing the actual parameter by using the reserved word `const` in the declaration of the formal parameter.
- Any attempt to change a constant array results in a compile-time error.
- **EXAMPLE 8-27:** Study the following function, and identify the line number that will cause syntax error.

```
1  void example(int x[], const int y[], int value)
2  {
3      x[0] = value;
4      y[0] = value;
5  }
```

  - Line 4: error: assignment of read-only location '* y'

- **EXAMPLE 8-28:** What is the output of the following program?

```
void printArray(const int list[], int listSize)
{
    int index;
    for (index = 0; index < listSize; index++)
        cout << list[index] << " ";
    cout<<endl;
}

int sumArray(const int list[], int listSize)
{
    int index;
    int sum = 0;
    for (index = 0; index < listSize; index++)
        sum = sum + list[index];
    return sum;
}

int main()
{
    int myList[5]={3,5,7,8,4};
    printArray(myList,5);
    cout<<"Sum = "<<sumArray(myList,5)<<endl;
}
```

OUTPUT:

```
3 5 7 8 4
Sum = 27
```

- **NOTE:** If C++ allowed arrays to be passed by value, the computer would have to allocate memory for the components of the formal parameter and copy the contents of the actual array into the corresponding formal parameter when the function is called. If the array size was large, this process would waste memory as well as the computer time needed for copying the data. That is why in C++ arrays are always passed by reference.
- C++ does not allow functions to return a value of the type array.

```
int[] arrayFunction(int list[], int size);
```

- **PRACTICE:** Use function `Larger` we studied earlier to write the definition of function `Largest` that determine the largest number in the array.

Passing Two-Dimensional Arrays as Parameters to Functions

- Two-dimensional arrays can be passed as parameters to a function, and they are passed by reference.
  - The base address is passed to the formal parameter. If matrix is the name of a two-dimensional array, then `matrix[0][0]` is the first component of matrix.
  - When storing a two-dimensional array in the computer's memory, C++ uses the row order form. That is, the first row is stored first, followed by the second row, followed by the third row, and so on.
- In the case of a one-dimensional array, when declaring it as a formal parameter, we usually omit the size of the array.
- Because C++ stores two-dimensional arrays in row order form, to compute the address of a component correctly, the compiler must know where one row ends and the next row begins.
- Thus, when declaring a two-dimensional array as a formal parameter, you can omit the size of the first dimension, but not the second; that is, you must specify the number of columns.

```
void matrixFun(int matrix[][5], int noOfRows); // Correct prototype
void matrixFun(int matrix[5][], int noOfRows); // Incorrect prototype
```

- This function takes as a parameter a two-dimensional array of an unspecified number of rows and five columns, and outputs the content of the two-dimensional array.
- During the function call, the number of columns of the actual parameter **must match** the number of columns of the formal parameter.
- **EXAMPLE 8-29:** Study the following functions that manipulate a two-dimensional array and find out the output of the program.

```cpp
const int NUMBER_OF_ROWS = 6;
const int NUMBER_OF_COLUMNS = 5;
void printMatrix(const int matrix[][NUMBER_OF_COLUMNS],
                 int noOfRows)
{
    int row, col;
    for (row = 0; row < noOfRows; row++)
    {
        for (col = 0; col < NUMBER_OF_COLUMNS; col++)
            cout << setw(5) << matrix[row][col] << " ";
        cout << endl;
    }
}

void sumRows(int matrix[][NUMBER_OF_COLUMNS], int noOfRows)
{
    int row, col;
    int sum;
    for (row = 0; row < noOfRows; row++)
    {
        sum = 0;
        for (col = 0; col < NUMBER_OF_COLUMNS; col++)
            sum = sum + matrix[row][col];
        cout << "Sum of row " << (row + 1) << " = " << sum
             << endl;
    }
}

int main()
{
    int board[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS]
    = {{23, 5, 6, 15, 18},
        {4, 16, 24, 67, 10},
        {12, 54, 23, 76, 11},
        {1, 12, 34, 22, 8},
        {81, 54, 32, 67, 33},
        {12, 34, 76, 78, 9}
    };
    printMatrix(board, NUMBER_OF_ROWS);
    cout << endl;
    sumRows(board, NUMBER_OF_ROWS);
    cout << endl;
    return 0;
}
```

OUTPUT:

```
    23     5     6    15    18
     4    16    24    67    10
    12    54    23    76    11
     1    12    34    22     8
    81    54    32    67    33
    12    34    76    78     9

Sum of row 1 = 67
Sum of row 2 = 121
Sum of row 3 = 176
Sum of row 4 = 77
Sum of row 5 = 267
Sum of row 6 = 209
```

C++ PROGRAMMING:

FROM PROBLEM ANALYSIS TO PROGRAM DESIGN

BY: D. S. MALIK

CHAPTER 12: POINTERS

SUMMARY & EXAMPLES

PREPARED BY:

E. MANAR JARADAT

Pointer Data Type and Pointer Variables

- **Recall**: C++'s data types are classified into three categories: simple, structured, and pointers.
- **Pointer variable**: A variable whose content is a memory address.

Declaring Pointer Variables

- The value of a pointer variable is an address. That is, the value refers to another memory space, where the data is typically stored.
- When you declare a pointer variable, you also specify the data type of the value to be stored in the memory location pointed to by the pointer variable.
- How do you declare pointer variables?
    - In C++, you declare a pointer variable by using the asterisk symbol (*) between the data type and the variable name.
- The general syntax to declare a pointer variable is:

```
dataType *identifier;
```

- Consider the following statements:

```
int *p;
char *ch;
```

    - `p` is called a pointer variable of type `int`, and `ch` is called a pointer variable of type `char`.
    - The content of `p` (when properly assigned) points to a memory location of type `int`.
    - The content of `ch` points to a memory location of type char.
- NOTE: The character `*` can appear anywhere between the data type name and the variable name. and so the following three statements are equivalent.

```
int *p;
int* p;
int * p;
```

- The general syntax to declare multiple pointer variables of the same type.

```
datatype *identifier1, *identifier2, ... , *identifiern;
```

- **EXAMPLE 12-1:** What is the difference between the following 2 statements.

```
int *p, q;
int *p, *q;
```

    - The first statement declares **p** to be a pointer variable of type `int`, while **q** in an integer variable.
    - The second statement declares both **p** and **q** to be pointer variables of type `int`.

Address of Operator (&)

- In C++, the ampersand, `&`, called **the address of operator**, is a unary operator that returns the address of its operand.
- For example, to assigns the address of the integer variable **x** to the pointer **p** use the following statements.

  ```cpp
  int x;
  int *p;
  p = &x;
  ```

  - After executing the third statement, both x and the value of p refer to the same memory location.

Dereferencing Operator (*)

- C++ uses `*` as a binary operator (for multiplication) or as a unary operator.
- When the , `*`, used as a unary operator, it is commonly referred to as the dereferencing operator or indirection operator.
- The , `*`, used to refer to the object to which its operand (that is, the pointer) points.
- **EXAMPLE 12-2:** Consider the following code segments and answer the questions below.

  ```cpp
  int x = 25;
  int *p;
  p = &x;          //store the address of x in p
  ```

  - What is the effect of the following statement?

    ```cpp
    cout << *p << endl;
    ```

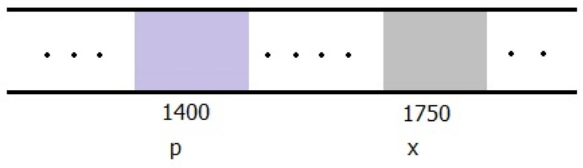
    - Prints the value stored in the memory space pointed to by p, which is the value of x
  - What is the output of the following statement?

    ```cpp
    *p = 55;
    cout << *p << endl;      //55
    ```

- **EXAMPLE 12-3:** Consider the following statements and write down the value of `&p`, `p`, `*p`, `&x`, `x` after executing each of the statements in the table below.

  ```cpp
  int *p;
  int x;
  ```

  Suppose that we have the memory allocation for `p` and `x` as shown in the figure below



Statement	&p	p	*p	&x	x
Initialization	1400	??	undefined	1750	??
`x = 50;`	1400	??	undefined	1750	50
`p = &x;`	1400	1750	50	1750	50
`*p = 38;`	1400	1750	38	1750	38

- A pointer can be modified to point to another variable during program execution.
- **EXAMPLE 12-4:** What is the output of the following code segment?

```
double val1 = 12.5, val2 = 6;
double *ptr = &val1;
*ptr = *ptr * 2;
ptr = &val2;
*ptr /= 2;
cout<<val1<<"  "<<val2<<"  "<<*ptr;
```

OUTPUT:

```
25  3  3
```

Initializing Pointer Variables

- Because C++ does not automatically initialize variables, pointer variables must be initialized if you do not want them to point to anything.
- Pointer variables are initialized using the constant value 0, or the named constant `NULL`, that are called the null pointer.
- The following two statements are equivalent and store the null pointer in **p**, that is, **p** points to nothing.

```
p = NULL;
p = 0;
```

- The number 0 is the only number that can be directly assigned to a pointer variable.

```
int *p = 10;    // error: invalid conversion from 'int' to 'int*'
```

Operations on Pointer Variables

- The operations that are allowed on pointer variables are the assignment and relational operations and some limited arithmetic operations.
- The value of one pointer variable can be assigned to another pointer variable **of the same type.**

```
int *p, *q;
int x = 8;
p = &x;
q = p;
cout<<*p<<" "<<*q;      //8 8
```

- Two pointer variables **of the same type** can be compared for equality.

```
int *p, *q;
int x = 8, y = 12;
p = &x;
q = p;
cout<<(p == q);     //1
q = &y;
cout<<(p != q);     // 1
```

- Integer values can be added and subtracted from a pointer variable.
- The value of one pointer variable can be subtracted from another pointer variable.
- The previous two type of operations usually performed when the pointers points to array elements.
- **EXAMPLE 12-5:** What is the output of the following code segment?

```
int *p, *q;
int myList[5] = {6,3,12,8,9};
p = &myList[0];
q = &myList[3];
cout<<*p<<" "<<*q<<endl;
p += 2;
q--;
cout<<*p<<" "<<*q<<endl;
cout<<p-q;
```

OUTPUT:

```
6  8
12  12
0
```

- **EXAMPLE 12-6:** What is the output of the following code segment?

```
double val1 = 12.5, val2 = 6;
double *ptr1, *ptr2;
ptr1 = &val1;
ptr2 = &val2;
double *ptr3 = ptr1;
ptr1 = ptr2;
ptr2 = ptr3;
cout<<*ptr1<<"  "<<*ptr2<<"  "<<*ptr3;
```

OUTPUT:

```
6  12.5  12.5
```

- **EXAMPLE 12-7:** Study the following code segment and identify the line(s) that will cause syntax error.

```
 1  double num = 12.5;
 2  int x;
 3  double *dptr;
 4  int *iptr;
 5  dptr = &x;
 6  iptr = &x;
 7  *iptr = 3;
 8  dptr = &num;
 9  dptr = iptr;
10  *dptr = *iptr;
```

  - Line 5: cannot convert 'int*' to 'double*' in assignment.
  - Line 9: cannot convert 'int*' to 'double*' in assignment.