MRSL 1163
SCIENTIFIC COMPUTING FOR SYSTEM ENGINEER

Lecture 1

Dr. Nelidya Md Yusoff

Razak Faculty of Technology and Informatics

Why Program?

- Computer – Programmable machine designed to follow instructions.
  - Instructed by programs.
- Program - is a set of instructions that a computer follows to perform a task.
  - Commonly referred to as software (e.g., word, power point).
- Programmer – person who writes programs.
  - Without programmers, no programs; Without programs, a computer does nothing

Computer System

- A computer system consists of similar hardware devices and software components

  – Hardware : Refers to physical components that a computer is made of.

  – Software : Makes the physical components to function and perform a task.

Main Hardware Component:

1. Central Processing Unit (CPU)
2. Main Memory
3. Secondary Memory / Storage
4. Input Devices
5. Output Devices

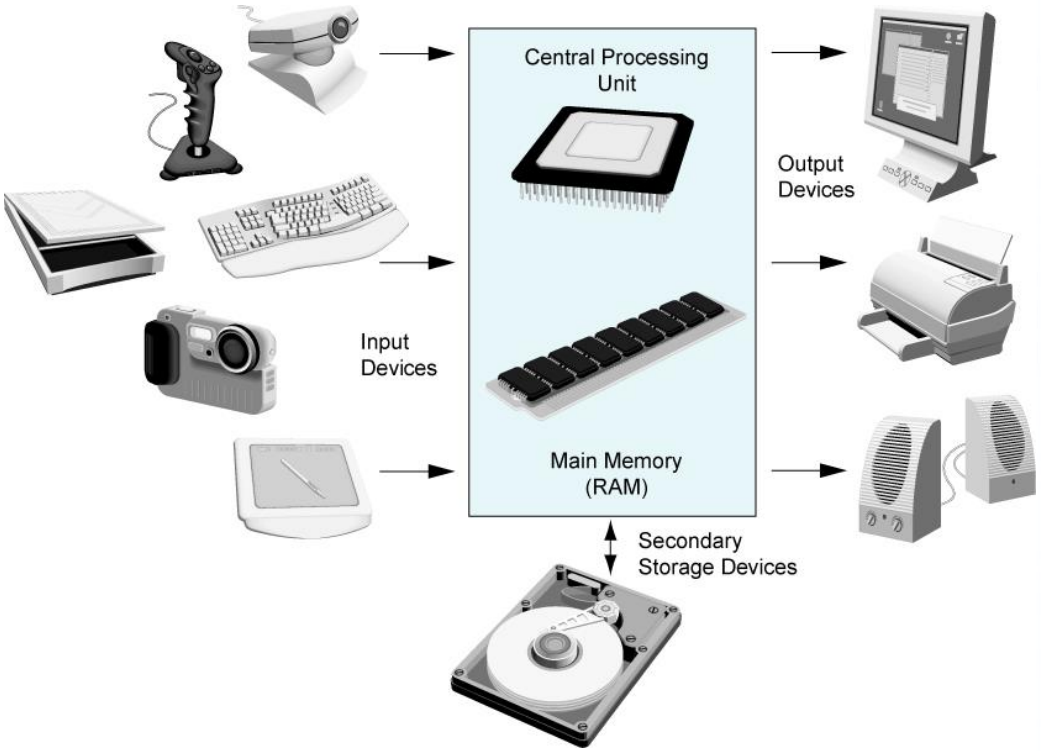


Figure 1-2: Organization of a computer system

CPU

The CPU (Brain of a computer) is the part of a computer that actually runs programs.

- Fetch Instructions
- Follow instructions
- Produce some results

- Comprised of:
  - Control Unit
    - Retrieves and decodes program instructions
    - Coordinates activities of all other parts of computer
  - Arithmetic & Logic Unit
    - It is designed to perform mathematical operations

CPU Organization

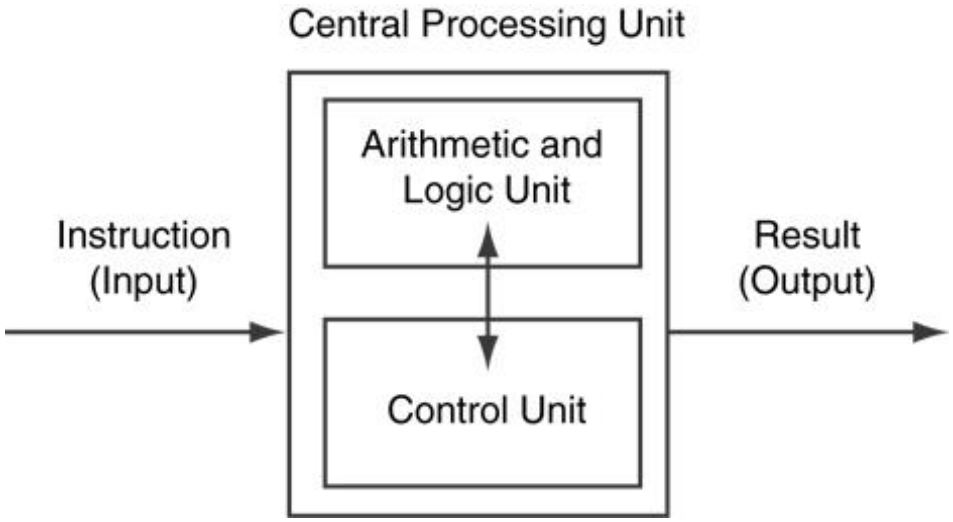


Figure 1-3

When a computer is running a program, the CPU is engaged in a process Known formally as the fetch/decode/execute cycle.

Fetch : CPU's control unit fetches the next instruction from main memory.
Decode: The instruction is encoded in the form of numbers. The **control unit decodes it and generates an electrical signal**
Execute: The **signal is routed to the appropriate component** of the computer to perform an operation.

Main Memory

- Computer's work area. This is where the computer stores program and data while the program is running.

- It is volatile. Main memory is erased when program terminates or computer is turned off
- Also called Random Access Memory (RAM)

- A computer's memory is divided into tiny storage locations known as bytes. Each byte is divided into eight smaller storage locations known as bits.
  - bit: smallest piece of memory. Has values 0 (off, false) or 1 (on, true)
  - byte: 8 consecutive bits. Bytes have addresses.

Main Memory

- **Addresses – Each byte in memory is identified by a unique number known as an *address*.**
- **Addresses are ordered from lowest to highest.**

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16 **149**	17	18	19
20	21	22	23 **72**	24	25	26	27	28	29

- In the above Figure, the number 149 is stored in the byte with the address 16, and the number 72 is stored at address 23.

Secondary Storage

- Non-volatile: data retained when program is not running or computer is turned off
- Comes in a variety of media:
  - magnetic: floppy disk, hard drive
  - optical: CD-ROM, DVD
  - Flash drives, connected to the USB port

Input Devices

- Devices that send information to the computer from outside
- Many devices can provide input:
  - Keyboard, mouse, scanner, digital camera, microphone
  - Disk drives, CD drives, and DVD drives

Software-Programs

That Run on a Computer

- Categories of software:
  - System software: programs that manage the computer hardware and the programs that run on them. *Examples*: operating systems, utility programs, software development tools
  - Application software: programs that provide services to the user. *Examples* : word processing, games, programs to solve specific problems

Programs and Programming Languages

- A program is a set of instructions that the computer follows to perform a task.
- A Programming language is a special language used to write computer programs.
- An *algorithm* - a set of well-defined steps.

Example Algorithm for Calculating Gross Pay

1. Display a message on the screen asking "How many hours did you work?"
2. Wait for the user to enter the number of hours worked. Once the user enters a number, store it in memory.
3. Display a message on the screen asking "How much do you get paid per hour?"
4. Wait for the user to enter an hourly pay rate. Once the user enters a number, store it in memory.
5. Multiply the number of hours by the amount paid per hour, and store the result in memory.
6. Display a message on the screen that tells the amount of money earned. The message must include the result of the calculation performed in Step 5.

Machine Language

- The previous algorithm defines the steps for calculating the gross pay, however, it is not ready to be executed on the computer.

- The computer only executes *machine language* instructions

Machine Language

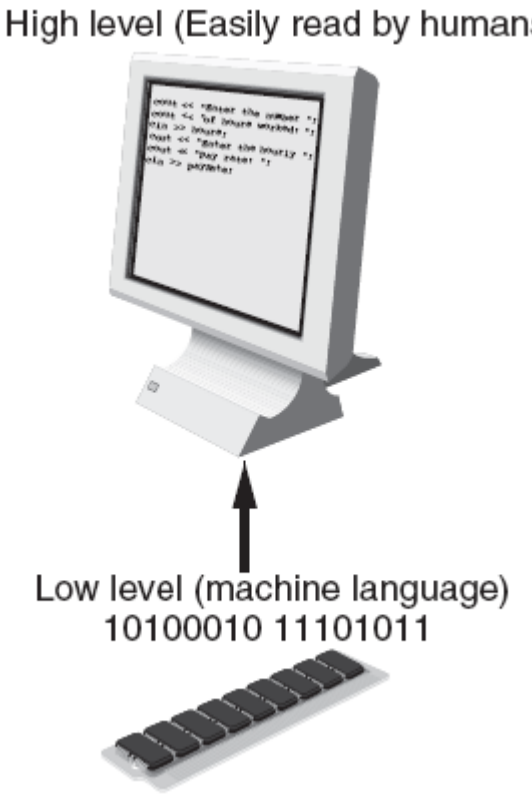
- Machine language instructions are binary numbers, such as

  1011010000000101

- Rather than writing programs in machine language, programmers use *programming languages*.

Programs and Programming Languages

- Types of languages:
  - **Low-level**: used for communication with computer hardware directly. Often written in binary machine code (0's/1's) directly.
  - **High-level**: closer to human language



Some Well-Known Programming Languages

C++

BASIC

Ruby

Java

FORTRAN

Visual Basic

COBOL

C#

JavaScript

C

Python

Algorithms

An algorithm is

- a step by step sequence of instructions
- to solve a specific problem
- in a finite amount of time

3 Different methods for constructing algorithms:

- **Pseudo-code**
- **Flowcharts**
- **Language**

Pseudo-code (or pseudocode)

- Pseudo-code is writing an algorithm as close as possible to computing language
  - Makes use of simple English-like statements
  - Useful for describing algorithms in a structured way.

Example 1:

- Turn off the tap when filling the bath tab.

PSEUDO-CODE

1. START
2. TURN ON TAPS
3. REPEAT
   - 3.1 LET THE WATER FLOW
   - 3.2 UNTIL BATH IS FULL
4. TURN OFF TAPS
5. STOP

Example 2 :

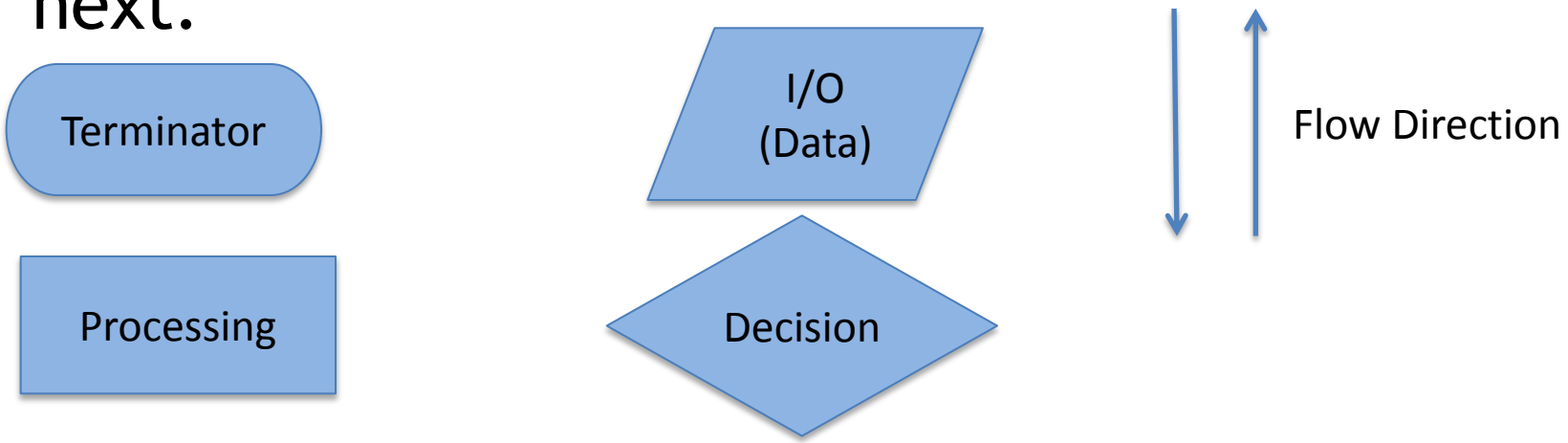
- Calculate gross pay.

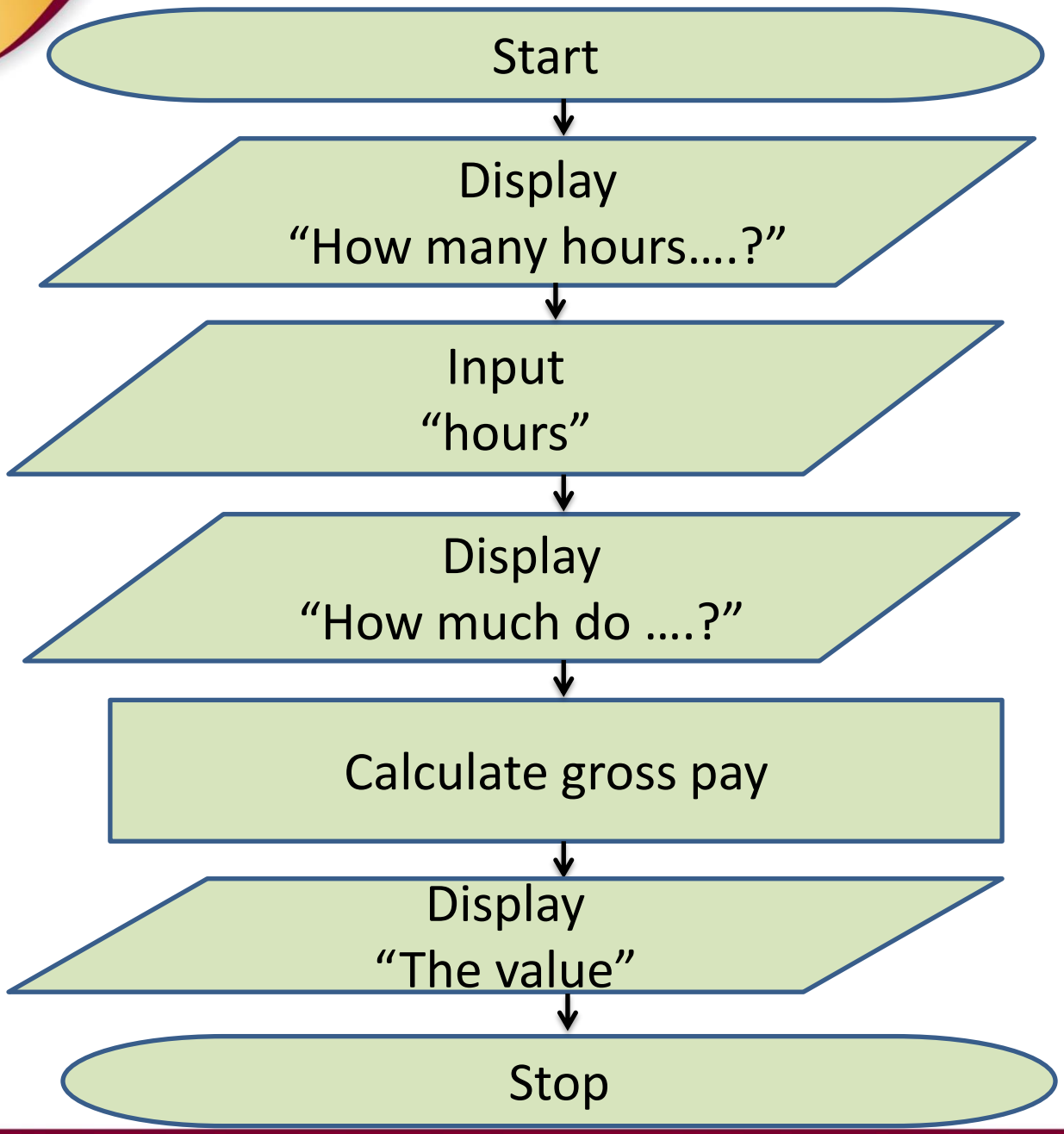
PSEUDO-CODE

1. START
2. DISPLAY “HOW MANAY HOURS DID YOU WORK?”
3. INPUT HOURS
4. DISPLAY “HOW MUCH DO YOU GET PAID PER HOUR?”
5. INPUT RATE
6. CALCULATE GROSS PAY i.e., HOURS * RATE
7. DISPLAY THE VALUE
5. STOP

Flowcharts

- A Flowchart is a graphical representation for an algorithm.
  - The programmers use flowcharts to display their ideas easily.
  - Flowcharts easily display the way a program naturally flows from one statement to the next.




Start
Display
"How many hours….?"
Input
"hours"
Display
"How much do ….?"
Calculate gross pay
Display
"The value"
Stop

Exercise

– Example 3:

- Student's grading System.

PSEUDO-CODE

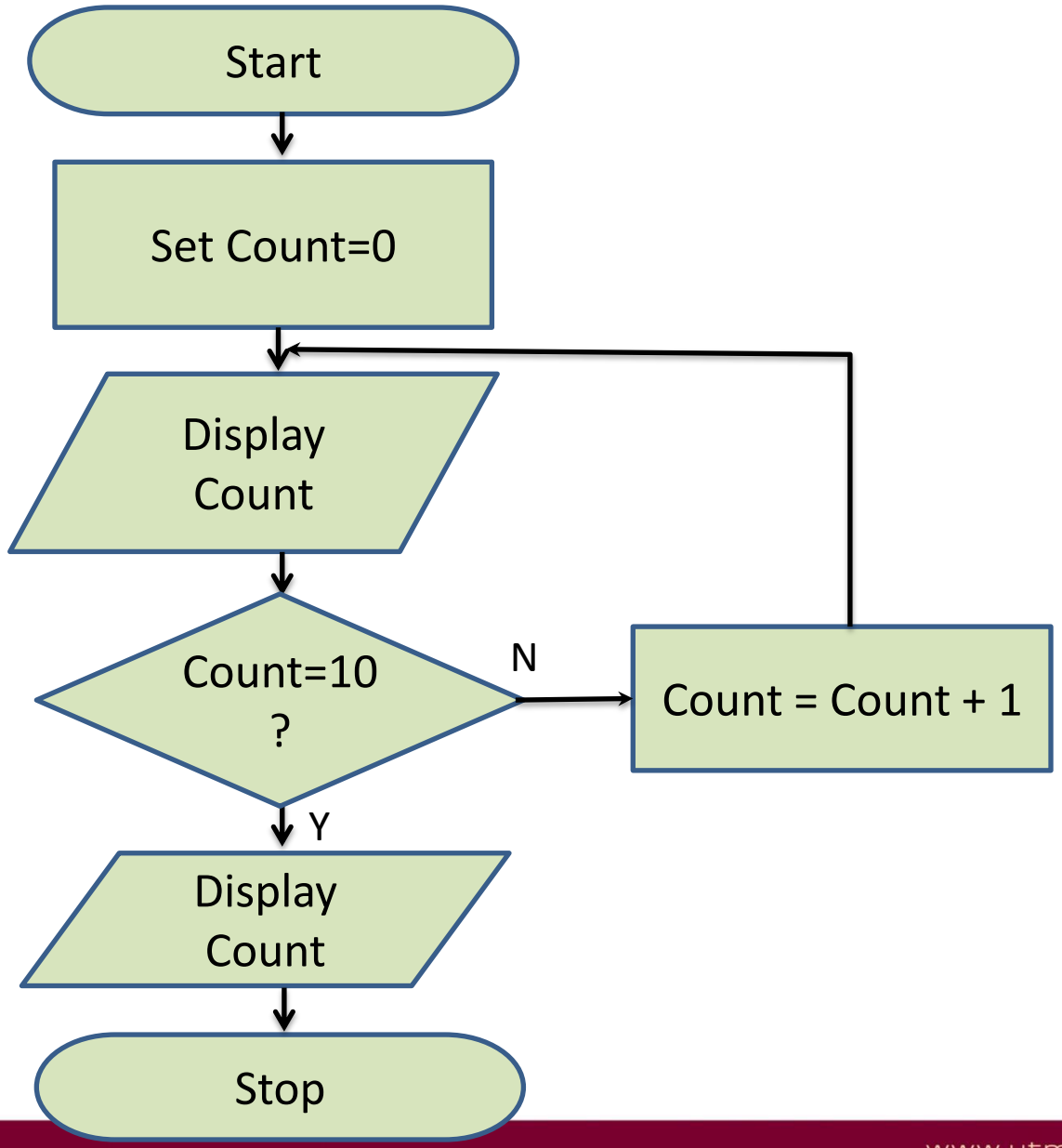
```
1.  START
2.    DISPLAY "ENTER MARK"
3.    INPUT MARK
4.    If MARK < 60
5.            DISPLAY "FAIL"
6.    ELSE
7.       DISPLAY "PASS"
5.   STOP
```

Exercise

- Example 4:
  - Input two numbers and display their average. If the average is positive display “VALID NUMBERS” otherwise show “INVALID NUMBER(S)”.

Exercise

Example 5:



From a High-Level Program to an Executable File

a) Create file containing the program with a text editor (i.e., source code).

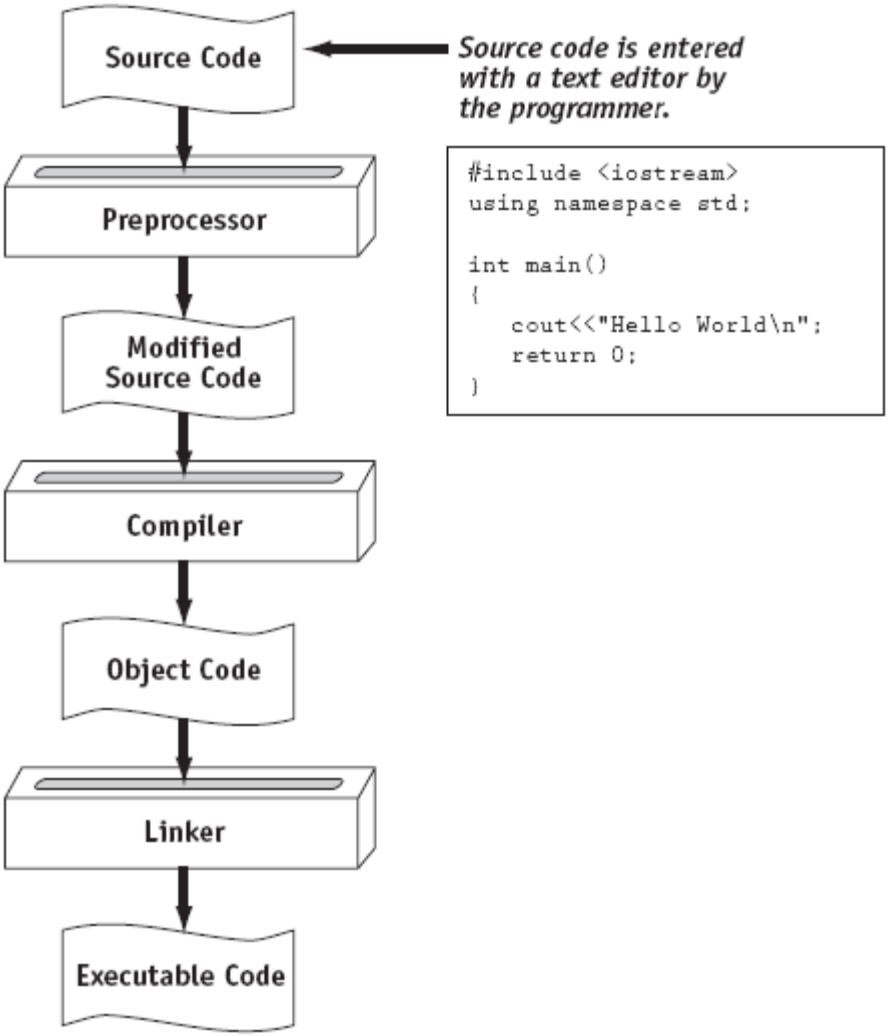
b) Run **preprocessor** to convert source file directives to source code program statements (i.e., modified source code).

c) Run **compiler** to convert source program into machine instructions (i.e., object code) – **translation process**

a) Run **linker** to connect hardware-specific code to machine instructions, producing an executable file (i.e., object file + run-time library).

- Steps b–d are often performed by a single command or button click.
- **Errors** detected at any step will prevent execution of following steps.

From a High-Level Program to an Executable File



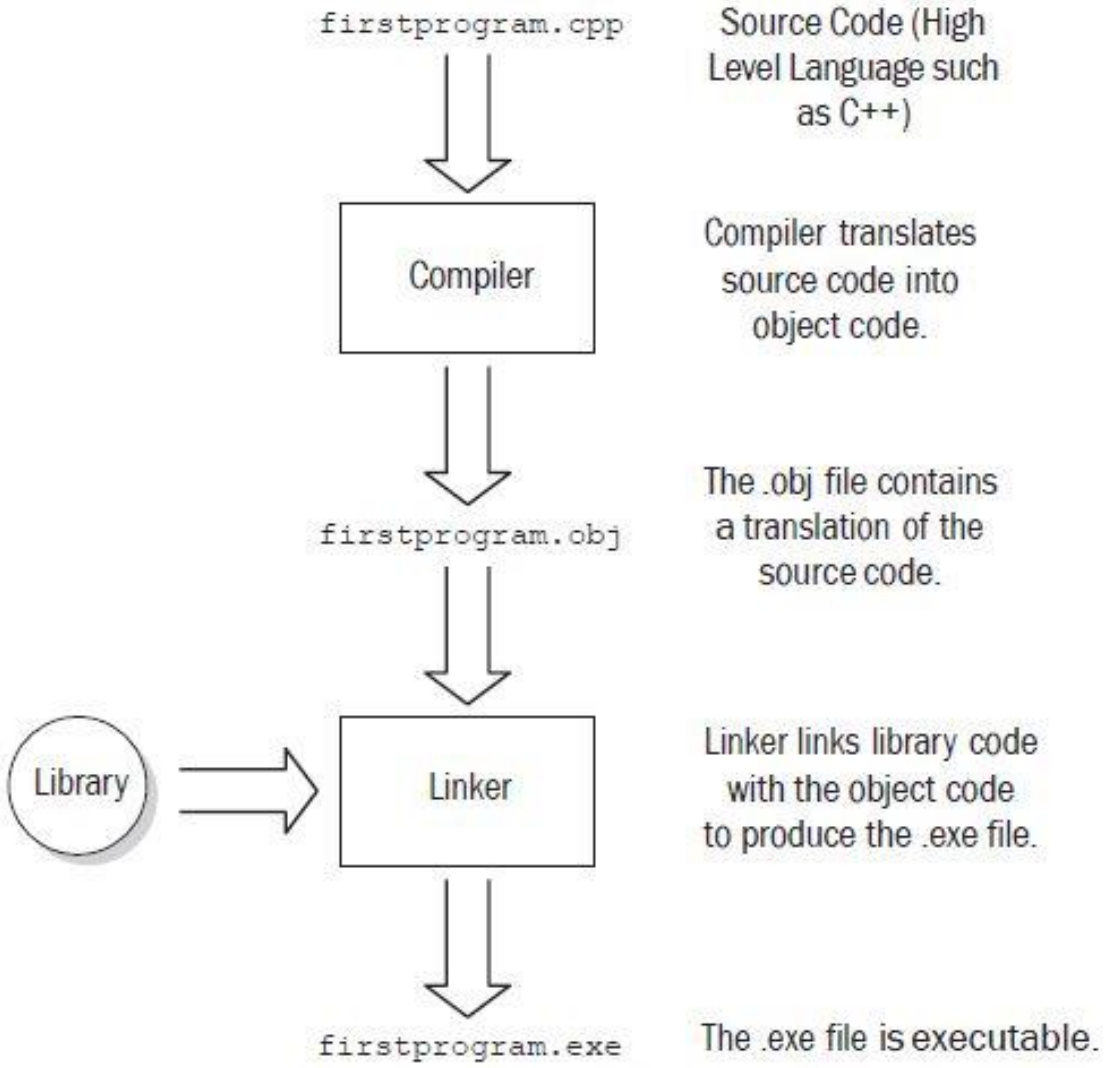
Translation Process

- Computers only understand the machine language (binary code)
- **Compiler** will do the translation from high level language (source code .cpp) to the machine code (object file .obj).
- Compiler also catches grammatical error called **syntax errors** in the source code.

Translation Process

- After the compile process is completed, **linking process** is required to ensure that the machine code is ready to be executed.
- The **linker** will attach an appropriate code to the program that is taken from the “software library” of programs.
- This produces what is called the executable code, generated in a file with .exe

Translation Process



Translation Process

- Once the executable code has been produced, the program is ready to be run.
- During the run time, we may encounter a second kind of error called a **run time error**.
- This error occurs when we ask the computer to do something that it cannot do (impossible to do)
- E.g.
  - Asking computer to divide by 0 – program will stop with run time error.
  - Running out of memory

Translation Process

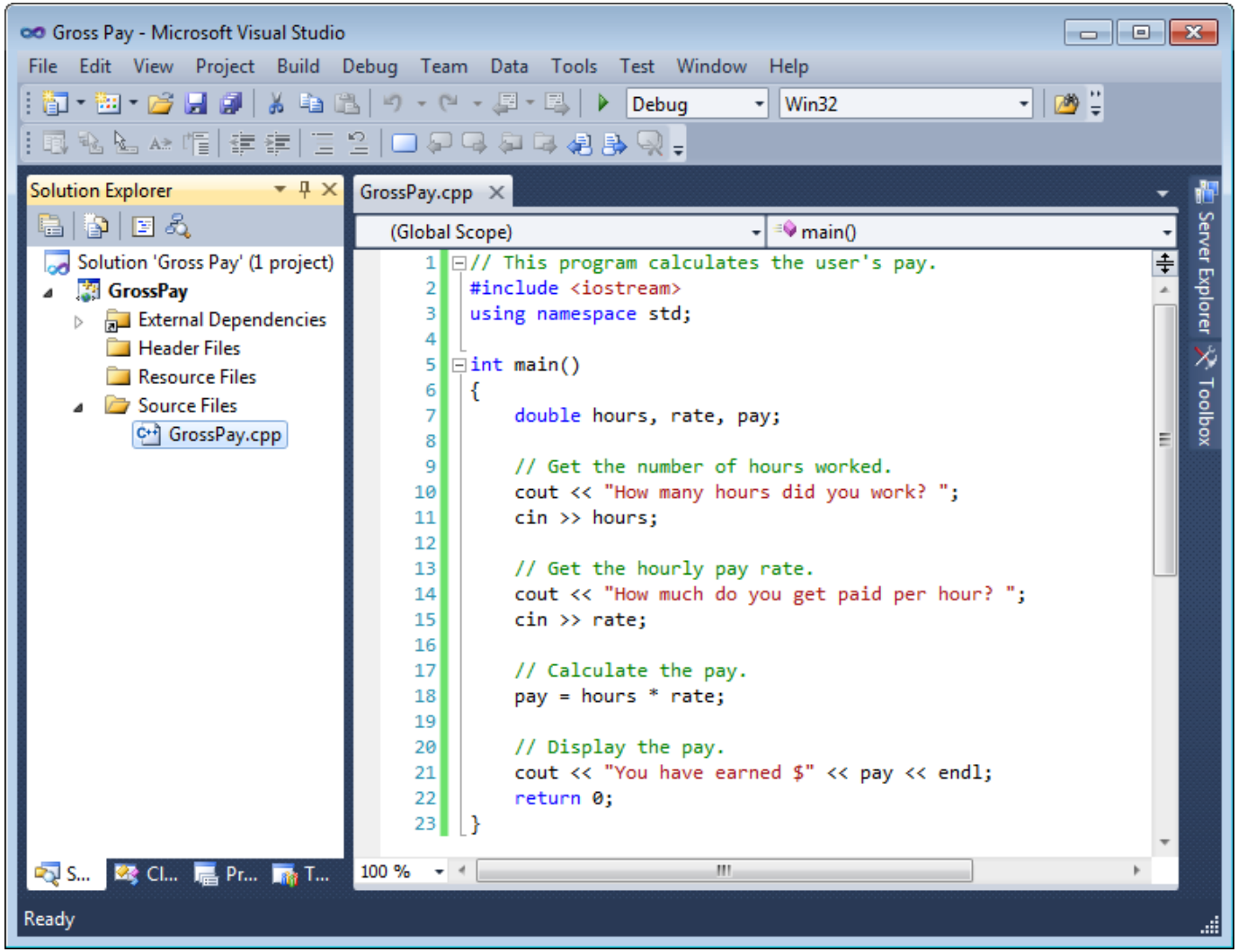
- Run time errors are usually more challenging to find than syntax errors.
- The worst type of error is the **logic errors**.
- Occurs when we ask the computer to do something, but mean for it to do something else.
- E.g.
  - We ask the computer to multiply a number by 3 but we actually want the number to be doubled.
- Logic errors are the **most difficult to find and correct** because there are no error message.

Integrated Development Environments (IDEs)

- An integrated development environment (IDE), combines all the tools needed to write, compile, and debug a program into a single software application.

- Examples are Microsoft Visual C++, Turbo C++ Explorer, CodeWarrior, etc.

Integrated Development Environments (IDEs)



Language (Example)

Display Hello World on the screen.

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

int main (){
        cout << “Hello World \n”;
        return 0;
}
```

Summary

- All computer systems consist of similar hardware devices and software components
  - **Hardware**: Refers to physical components that a computer is made of.
    1. Central Processing Unit (CPU)
    2. Main Memory
    3. Secondary Memory / Storage
    4. Input Devices
    5. Output Devices
  - **Software**: Makes the physical components to function and perform a task.
    - System Software
    - Application Software

MRSL 1163
SCIENTIFIC COMPUTING FOR SYSTEM ENGINEER

Lecture 2

Dr. Nelidya Md Yusoff

Razak Faculty of Technology and Informatics

From a High-Level Program to an Exe File

a) Create file containing the program with a text editor (i.e., source code).

b) Run preprocessor to convert source file directives to source code program statements (i.e., modified source code).

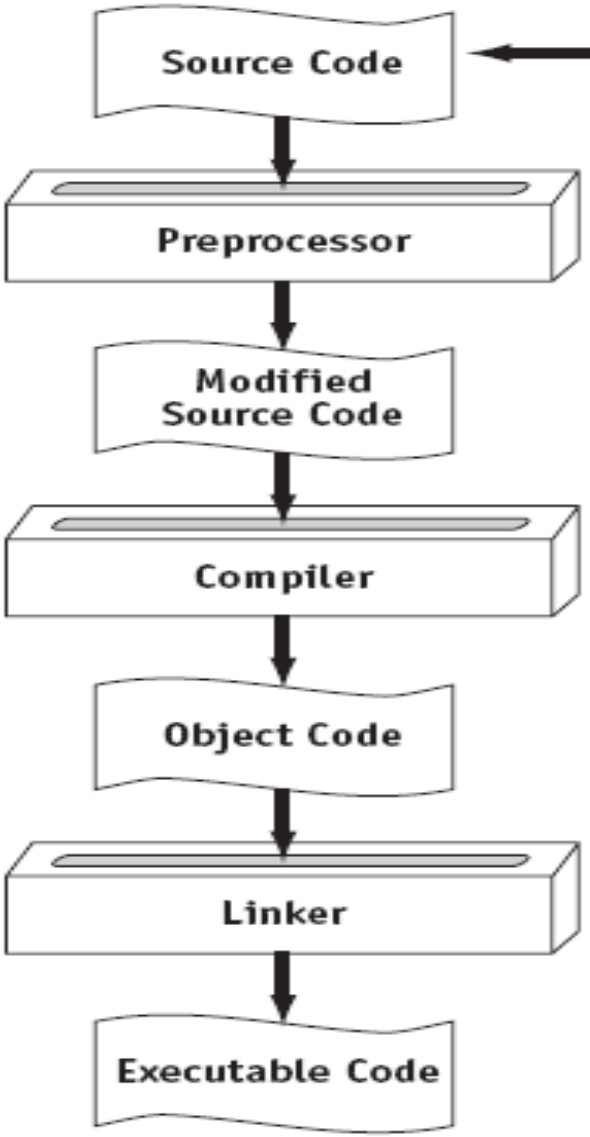
c) Run compiler to convert source program into machine instructions (i.e., object code).

d) Run linker to connect hardware-specific code to machine instructions, producing an executable file (i.e., object file + run-time library).

Using IDE

- Steps b-d are often performed by a single command or button click.
- Errors detected at any step will prevent execution of following steps.

From a High-Level Program to an Executable File



Source code is entered with a text editor by the programmer.

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

int main()
{
    cout<<"Hello World\n";
    return 0;
}
```

The Parts of a C++ Program

```
//A simple C++ program
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, world\n";
    return 0;
}
```

comment

preprocessor directive

which namespace to use

beginning of function named `main`

beginning of block for `main`

output statement

string literal

send `0` to operating system

end of block for `main`

- Preprocessor-
  - It setups the source code for the compiler. It reads your program before it is compiled.
  - Include the contents of another file in the program.
  - iostream - called header file and must be included in the top of the program.

What is a Program Made of?

- Common elements in programming languages:
  - Key Words
  - Programmer-Defined Identifiers
  - Operators
  - Punctuation
  - Syntax

Program 1-1

```
1  // This program calculates the user's pay.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     double hours, rate, pay;
8
9     // Get the number of hours worked.
10    cout << "How many hours did you work? ";
11    cin >> hours;
12
13    // Get the hourly pay rate.
14    cout << "How much do you get paid per hour? ";
15    cin >> rate;
16
17    // Calculate the pay.
18    pay = hours * rate;
19
20    // Display the pay.
21    cout << "You have earned $" << pay << endl;
22    return 0;
23 }
```

Key Words

- Also known as reserved words
- Have a special meaning in C++
- Can not be used for any other purpose
- Key words in the Program 1-1: `using`, `namespace`, `int`, `double`, **and** `return`

Key Words

```
1  // This program calculates the user's pay.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     double hours, rate, pay;
8
9     // Get the number of hours worked.
10    cout << "How many hours did you work? ";
11    cin >> hours;
12
13    // Get the hourly pay rate.
14    cout << "How much do you get paid per hour? ";
15    cin >> rate;
16
17    // Calculate the pay.
18    pay = hours * rate;
19
20    // Display the pay.
21    cout << "You have earned $" << pay << endl;
22    return 0;
23 }
```

C++ Key Words

You cannot use any of the C++ key words as an identifier. These words have reserved meaning.

Table 2-4 The C++ Key Words

and	continue	goto	public	try
and_eq	default	if	register	typedef
asm	delete	inline	reinterpret_cast	typeid
auto	do	int	return	typename
bitand	double	long	short	union
bitor	dynamic_cast	mutable	signed	unsigned
bool	else	namespace	sizeof	using
break	enum	new	static	virtual
case	explicit	not	static_cast	void
catch	export	not_eq	struct	volatile
char	extern	operator	switch	wchar_t
class	false	or	template	while
compl	float	or_eq	this	xor
const	for	private	throw	xor_eq
const_cast	friend	protected	true	

Programmer-Defined Identifiers

- Names made up by the programmer
- Not part of the C++ language
- Used to represent various things: variables (memory locations), functions, etc.
- In Program 1-1: `hours`, `rate`, **and** `pay`.

Identifiers

```
1  // This program calculates the user's pay.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     double hours, rate, pay;
8
9     // Get the number of hours worked.
10    cout << "How many hours did you work? ";
11    cin >> hours;
12
13    // Get the hourly pay rate.
14    cout << "How much do you get paid per hour? ";
15    cin >> rate;
16
17    // Calculate the pay.
18    pay = hours * rate;
19
20    // Display the pay.
21    cout << "You have earned $" << pay << endl;
22    return 0;
23 }
```

Operators

- Used to perform operations on data
- Many types of operators:
  - Arithmetic - ex: `+, -, *, /`
  - Assignment – ex: `=`

- Some operators in Program1-1:
  `>>` (*extraction operator*)
  `<<` (*insertion operator*)
  `=` `*`

Operators

```
1  // This program calculates the user's pay.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     double hours, rate, pay;
8
9     // Get the number of hours worked.
10    cout << "How many hours did you work? ";
11    cin >> hours;
12
13    // Get the hourly pay rate.
14    cout << "How much do you get paid per hour? ";
15    cin >> rate;
16
17    // Calculate the pay.
18    pay = hours * rate;
19
20    // Display the pay.
21    cout << "You have earned $" << pay << endl;
22    return 0;
23 }
```

Punctuation

```
1  // This program calculates the user's pay.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     double hours, rate, pay;
8
9     // Get the number of hours worked.
10    cout << "How many hours did you work? ";
11    cin >> hours;
12
13    // Get the hourly pay rate.
14    cout << "How much do you get paid per hour? ";
15    cin >> rate;
16
17    // Calculate the pay.
18    pay = hours * rate;
19
20    // Display the pay.
21    cout << "You have earned $" << pay << endl;
22    return 0;
23 }
```

- Characters that mark the end of a statement, or that separate items in a list
- In Program 1-1: , and ;

Syntax

- The rules of grammar that must be followed when writing a program
- Controls the use of key words, operators, programmer-defined symbols, and punctuation

Variables

- A variable is a named storage location in the computer's memory for holding a piece of data.
- In Program 1-1 we used three variables:
  - The `hours` variable was used to hold the hours worked
  - The `rate` variable was used to hold the pay rate
  - The `pay` variable was used to hold the gross pay

Variable Definitions

- To create a variable in a program you must write a variable definition (also called a variable declaration)

- Here is the statement from Program 1-1 that defines the variables:

```
double hours, rate, pay;
int number;
string myName;
```

Variable Definitions

- There are many different types of data
- A variable holds a specific type of data.
- The variable definition specifies the type of data a variable can hold, and the variable name.

Variable Definitions

- Once again, line 7 from Program 1-1:

```
double hours, rate, pay;
```

- The word `double` specifies that the variables can hold double-precision floating point numbers.

Exercise

- There are total 100 students in ESE. However, only 33 of them took SMJE1013 course.

Write a C++ code to display how may percentage is attending the course?

using namespace std;

- Programs usually contain several items with unique names.
- Variables, functions, and objects – example of program entities that must have names.
- C++ uses namespace to organize the names of program entities.
- The statement declares that the program will be accessing entities whose names are part of the namespace called std.
- Every name created by the iostream file is part of std namespace. To use the entities in iostream, the program must have access to std.

Special Characters

Character	Name	Meaning
//	Double slash	Beginning of a comment
#	Pound sign	Beginning of preprocessor directive
< >	Open/close brackets	Enclose filename in #include
( )	Open/close parentheses	Used when naming a function
{ }	Open/close brace	Encloses a group of statements
" "	Open/close quotation marks	Encloses string of characters
;	Semicolon	End of a programming statement

The `cout` Object

- Displays output on the computer screen
- You use the stream insertion operator `<<` to send output to `cout`:

```
cout << "Programming is fun!";
```

The `cout` Object

- Can be used to send more than one item to cout:

```
cout << "Hello " << "there!";
```

Or:

```
cout << "Hello ";
cout << "there!";
```

The `cout` Object

- This produces one line of output:

```
cout << "Programming is ";
cout << "fun!";
```

The `endl` Manipulator

- You can use the `endl` manipulator to start a new line of output. This will produce two lines of output:

```
cout << "Programming is" <<
endl;
cout << "fun!";
```

The `endl` Manipulator

```
cout << "Programming is" << endl;
cout << "fun!";
```

The `endl` Manipulator

- You do NOT put quotation marks around `endl`

- The last character in `endl` is a lowercase L, not the number 1.

endl ← This is a lowercase L

The \n Escape Sequence

- You can also use the \n escape sequence to start a new line of output. This will produce two lines of output:

```
cout << "Programming is\n";
cout << "fun!";
```

Notice that the \n is INSIDE the string.

The \n Escape Sequence

```
cout << "Programming is\n";
cout << "fun!";
```

The `#include` Directive

- Inserts the contents of another file into the program
- This is a preprocessor directive, not part of C++ language
- `#include` lines not seen by compiler
- Do not place a semicolon at end of `#include` line

Variables and Literals

- Variable: a storage location in memory
  - Has a name and a type of data it can hold
  - Must be defined before it can be used:

```
int item;
```

Variable Definition in Program 2-7

Program 2-7

```
1  // This program has a variable.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int number;
8
9     number = 5;
10    cout << "The value in number is " << number << endl;
11    return 0;
12 }
```

Variable Definition

Program Output

```
The value in number is 5
```

Literals

- Literal: a value that is written into a program's code.

  `"hello, there"` (string literal)

  `12` (integer literal)

Integer Literal in Program 2-9

Program 2-9

```
1  // This program has literals and a variable.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int apples;
8
9     apples = 20;
10    cout << "Today we sold " << apples << " bushels of apples.\n";
11    return 0;
12 }
```

20 is an integer literal

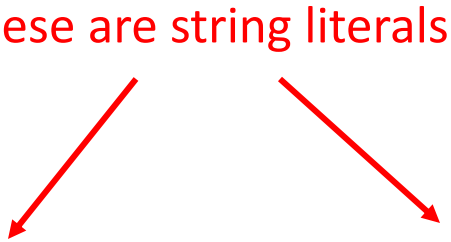
Program Output

```
Today we sold 20 bushels of apples.
```

String Literals in Program 2-9

Program 2-9

```
1  // This program has literals and a variable.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int apples;
8
9     apples = 20;
10    cout << "Today we sold " << apples << " bushels of apples.\n";
11    return 0;
12 }
```



Program Output

```
Today we sold 20 bushels of apples.
```

Identifiers

- An identifier is a programmer-defined name for some part of a program: variables, functions, etc.

Variable Names

- A variable name should represent the purpose of the variable. For example:

  int x;

- `int itemsOrdered;`

  The purpose of this variable is to hold the number of items ordered.

Identifier Rules

- The first character of an identifier must be an alphabetic character or an underscore (_),
- After the first character you may use alphabetic characters, numbers, or underscore characters.
- Upper- and lowercase characters are distinct

Valid and Invalid Identifiers

IDENTIFIER	VALID?	REASON IF INVALID
`totalSales`	Yes	
`total_Sales`	Yes	
`total.Sales`	No	Cannot contain .
`4thQtrSales`	No	Cannot begin with digit
`totalSale$`	No	Cannot contain $

Integer Data Types

- Integer variables can hold whole numbers such as 12, 7, and -99.

Table 2-6 Integer Data Types, Sizes, and Ranges

Data Type	Size	Range
`short`	2 bytes	–32,768 to +32,767
`unsigned short`	2 bytes	0 to +65,535
`int`	4 bytes	–2,147,483,648 to +2,147,483,647
`unsigned int`	4 bytes	0 to 4,294,967,295
`long`	4 bytes	–2,147,483,648 to +2,147,483,647
`unsigned long`	4 bytes	0 to 4,294,967,295

Defining Variables

- Variables of the same type can be defined
  - On separate lines:

```
int length;
int width;
unsigned int area;
```

  - On the same line:

```
int length, width;
unsigned int area;
```

- Variables of different types must be in different definitions
  - int length;
  - unsigned int area;

Integer Types in Program 2-10

Program 2-10

```
1  // This program has variables of several of the integer types.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int checking;
8     unsigned int miles;
9     long days;
10
11    checking = -20;
12    miles = 4276;
13    days = 189000;
14    cout << "We have made a long journey of " << miles;
15    cout << " miles.\n";
16    cout << "Our checking account balance is " << checking;
17    cout << "\nAbout " << days << " days ago Columbus ";
18    cout << "stood on this spot.\n";
19    return 0;
20 }
```

This program has three variables: checking, miles, and days

Integer Literals

- An integer literal is an integer value that is typed into a program's code. For example:

```
itemsOrdered = 15;
```

In this code, 15 is an integer literal.

Integer Literals in Program 2-10

Program 2-10

```
1  // This program has variables of several of the integer types.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int checking;
8     unsigned int miles;
9     long days;
10
11    checking = -20;
12    miles = 4276;
13    days = 189000;
14    cout << "We have made a long journey of " << miles;
15    cout << " miles.\n";
16    cout << "Our checking account balance is " << checking;
17    cout << "\nAbout " << days << " days ago Columbus ";
18    cout << "stood on this spot.\n";
19    return 0;
20 }
```

Integer Literals

Integer Literals

- Integer literals are stored in memory as `int`s by default
- To store an integer constant in a long memory location, put '`L`' at the end of the number: `1234L`
- Constants that begin with '`0`' (zero) are base 8: `075 (octal)`
- Constants that begin with '`0x`' are base 16: `0x75A (hexadecimal)`

The `char` Data Type

- Used to hold characters or very small integer values
- Usually 1 byte of memory
- Numeric value of character from the character set is stored in memory:

CODE:

```
char letter;
letter = 'C';
```

MEMORY:

letter

Character Literals

- Character literals must be enclosed in single quote marks. Example:

```
'A'
```

Character Literals in Program 2-13

Program 2-13

```
1  // This program uses character literals.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     char letter;
8
9     letter = 'A';
10    cout << letter << endl;
11    letter = 'B';
12    cout << letter << endl;
13    return 0;
14 }
```

Program Output

```
A
B
```

Exercise

- Write a program that asks the users to perform the following processes:
  - Display : 1. Obtained grades in Math:
  - Input : mathGrade;
  - Display : 2. Obtained grade in Programing:
  - Input : programmingGrade;
  - Display : 1. Grade in Math is : B
  - Display : 2. Grade in Progmramming is : A

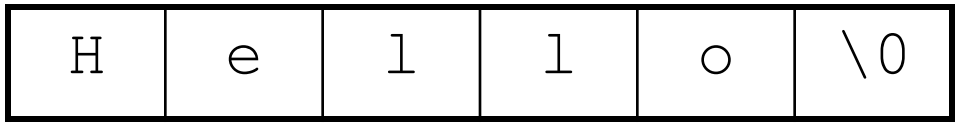
Character Strings

- A series of characters in consecutive memory locations:

  `"Hello"`

- Stored with the null terminator, `\0`, at the end:

- Comprised of the characters between the " "



The C++ `string` Class

- Special data type supports working with strings

```
#include <string>
```

- Can define `string` variables in programs:

```
string firstName, lastName;
```

- Can receive values with assignment operator:

```
firstName = "George";
lastName  = "Washington";
```

- Can be displayed via `cout`

```
cout << firstName << " " << lastName;
```

The `string` class in Program 2-15

```
#include <iostream>
#include <string> // required for the string class
using namespace std;
int main()
{
string movieTitle;
movieTitle = "wheels of Fury";
cout << "My favourite movie is " << movieTitle<<endl;
return 0;
}
```

Floating-Point Data Types

- Can hold real numbers i.e., fractional values
- The floating-point data types are:
  ```
  float
  double
  long double
  ```
- They can hold real numbers such as:
  ```
  12.45      -3.8
  ```
- Stored in a form similar to scientific notation
- All floating-point numbers are signed

Floating-Point Data Types

Table 2-8 Floating Point Data Types on PCs

Data Type	Key Word	Description
Single precision	`float`	4 bytes. Numbers between ±3.4E-38 and ±3.4E38
Double precision	`double`	8 bytes. Numbers between ±1.7E-308 and ±1.7E308
Long double precision	`long double*`	8 bytes. Numbers between ±1.7E-308 and ±1.7E308

Floating-Point Literals

- Can be represented in
  - Fixed point (decimal) notation:

    `31.4159` `0.0000625`
  - E notation:

    `3.14159E1` `6.25e-5`
- Are `double` by default
- Can be forced to be float (`3.14159f`) or long double (`0.0000625L`)

Floating-Point Data Types in Program 2-16

Program 2-16

```
1  // This program uses floating point data types.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     float distance;
8     double mass;
9
10    distance = 1.495979E11;
11    mass = 1.989E30;
12    cout << "The Sun is " << distance << " meters away.\n";
13    cout << "The Sun\'s mass is " << mass << " kilograms.\n";
14    return 0;
15 }
```

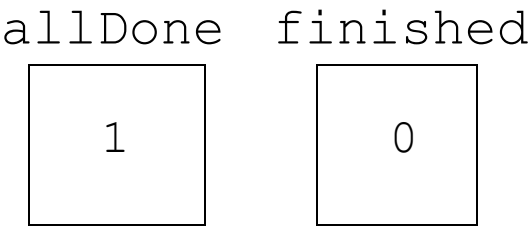
Program Output

```
The Sun is 1.49598e+011 meters away.
The Sun's mass is 1.989e+030 kilograms.
```

The `bool` Data Type

- Represents values that are `true` or `false`
- `bool` variables are stored as small integers
- `false` is represented by 0, `true` by 1:

```
bool allDone = true;
bool finished = false;
```



Boolean Variables in Program 2-17

Program 2-17

```
1   // This program demonstrates boolean variables.
2   #include <iostream>
3   using namespace std;
4
5   int main()
6   {
7      bool boolValue;
8
9      boolValue = true;
10     cout << boolValue << endl;
11     boolValue = false;
12     cout << boolValue << endl;
13     return 0;
14  }
```

Program Output

```
1
0
```

Determining the Size of a Data Type

The `sizeof` operator gives the size of any data type or variable:

```
      double amount;
cout << "A double is stored in "
       << sizeof(double) <<
"bytes\n";
cout << "Variable amount is stored
in "      << sizeof(amount)
     << "bytes\n";
```

Variable Assignments and Initialization

- An assignment statement uses the = operator to store a value in a variable.

```
item = 12;
```

- This statement assigns the value 12 to the `item` variable.

Assignment

- The variable receiving the value must appear on the left side of the = operator.
- This will NOT work:

```
// ERROR!
12 = item;
```

Variable Initialization

- To initialize a variable means to assign it a value when it is defined:

```
int length = 12;
```

- Can initialize some or all variables:

```
int length = 12, width = 5, area;
```

Variable Initialization in Program 2-19

Program 2-19

```
1  // This program shows variable initialization.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int month = 2, days = 28;
8
9     cout << "Month " << month << " has " << days << " days.\n";
10    return 0;
11 }
```

Program Output

```
Month 2 has 28 days.
```

Scope

- The scope of a variable: the part of the program in which the variable can be accessed
- A variable cannot be used before it is defined

Variable Out of Scope in Program 2-20

Program 2-20

```
1  // This program can't find its variable.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     cout << value; // ERROR! value not defined yet!
8
9     int value = 100;
10    return 0;
11 }
```

Arithmetic Operators

- Used for performing numeric calculations
- C++ has unary, binary, and ternary operators:
  - unary (1 operand)
    - ex. int value = 12;
  - binary (2 operands)
    - `ex. value = val1 + val2`
    - `Ex. if (a == b)`
      - ternary (3 operands) `exp1 ? exp2 : exp3`

        `Ex. a < b : true`

Binary Arithmetic Operators

SYMBOL	OPERATION	EXAMPLE	VALUE OF `ans`
+	addition	`ans = 7 + 3;`	10
-	subtraction	`ans = 7 - 3;`	4
*	multiplication	`ans = 7 * 3;`	21
/	division	`ans = 7 / 3;`	2
%	modulus	`ans = 7 % 3;`	1

Arithmetic Operators in Program 2-21

Program 2-21

```
1  // This program calculates hourly wages, including overtime.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     double regularWages,              // To hold regular wages
8            basePayRate = 18.25,      // Base pay rate
9            regularHours = 40.0,      // Hours worked less overtime
10           overtimeWages,            // To hold overtime wages
11           overtimePayRate = 27.78,  // Overtime pay rate
12           overtimeHours = 10,       // Overtime hours worked
13           totalWages;               // To hold total wages
14
15    // Calculate the regular wages.
16    regularWages = basePayRate * regularHours;
17
18    // Calculate the overtime wages.
19    overtimeWages = overtimePayRate * overtimeHours;
20
21    // Calculate the total wages.
22    totalWages = regularWages + overtimeWages;
23
24    // Display the total wages.
25    cout << "Wages for this week are $" << totalWages << endl;
26    return 0;
27 }
```

Program Output

```
Wages for this week are $1007.8
```

A Closer Look at the / Operator

- / (division) operator performs integer division if both operands are integers

```
cout << 13 / 5;     // displays 2
cout << 91 / 7;     // displays 13
```

- If either operand is floating point, the result is floating point

```
cout << 13 / 5.0;   // displays 2.6
cout << 91.0 / 7;   // displays 13.0
```

A Closer Look at the % Operator

- % (modulus) operator computes the remainder resulting from integer division

```
cout << 13 % 5;    // displays 3
```

- % requires integers for both operands

```
cout << 13 % 5.0; // error
```

Comments

- Used to document parts of the program
- Intended for persons reading the source code of the program:
  - Indicate the purpose of the program
  - Describe the use of variables
  - Explain complex sections of code
- Are ignored by the compiler

Single-Line Comments

Begin with // through to the end of line:

```
int length = 12; // length in inches
int width = 15;  // width in inches
int area;        // calculated area

// calculate rectangle area
area = length * width;
```

Multi-Line Comments

- Begin with `/*`, end with `*/`
- Can span multiple lines:

```
/* this is a multi-line
   comment
*/
```

- Can begin and end on the same line:

```
int area;   /* calculated area */
```

Named Constants

- Named constant (constant variable): variable whose content cannot be changed during program execution

- Used for representing constant values with descriptive names:

```
const double TAX_RATE = 0.0675;
const int NUM_STATES = 50;
```

- Often named in uppercase letters

Named Constants in Program 2-28

Program 2-28

```
1  // This program calculates the circumference of a circle.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     // Constants
8     const double PI = 3.14159;
9     const double DIAMETER = 10.0;
10
11    // Variable to hold the circumference
12    double circumference;
13
14    // Calculate the circumference.
15    circumference = PI * DIAMETER;
16
17    // Display the circumference.
18    cout << "The circumference is: " << circumference << endl;
19    return 0;
20 }
```

Program Output

```
The circumference is: 31.4159
```

Programming Style

- The visual organization of the source code
- Includes the use of spaces, tabs, and blank lines
- Does not affect the syntax of the program
- Affects the readability of the source code

Programming Style

Common elements to improve readability:

- Braces { } aligned vertically
- Indentation of statements within a set of braces
- Blank lines between declaration and other statements
- Long statements wrapped over multiple lines with aligned operators

Standard and Prestandard C++

Older-style C++ programs:

- Use `.h` at end of header files:
- `#include <iostream.h>`
- Use `#define` preprocessor directive instead of `const` definitions
- Do not use `using namespace` convention
- May not compile with a standard C++ compiler

`#define` directive in Program 2-31

Program 2-31

```
1  // This program calculates the circumference of a circle.
2  #include <iostream>
3  using namespace std;
4
5  #define PI 3.14159
6  #define DIAMETER 10.0
7
8  int main()
9  {
10    // Variable to hold the circumference
11    double circumference;
12
13    // Calculate the circumference.
14    circumference = PI * DIAMETER;
15
16    // Display the circumference.
17    cout << "The circumference is: " << circumference << endl;
18    return 0;
19 }
```

Program Output

```
The circumference is: 31.4159
```

Decimal-Octal

- 601 decimal to octal
  Divide 600 by 8 = 75 and the remainder is 1
  Divide 75 by 8 = 9 and the remainder is 3
  Divide 9 by 8 =1 and the remainder is 1
  Divide 1 by 8 = 0 and the remainder is 1
  Reading from bottom to top 300 decimal is 1131 octal.
  So keep dividing the number by the base number, until you only have a remainder, then use last remainder as the most significant number..

  To change 1131 octal to decimal:
- Right most number 1
  Multiply 3 x 8 = 24
  Multiply 1 x 8 x 8 = 64
  Multiply 1 x 8 x 8 x 8 = 512.
  Add your numbers up and you get 601.

MRSL 1163
SCIENTIFIC COMPUTING FOR SYSTEM ENGINEER

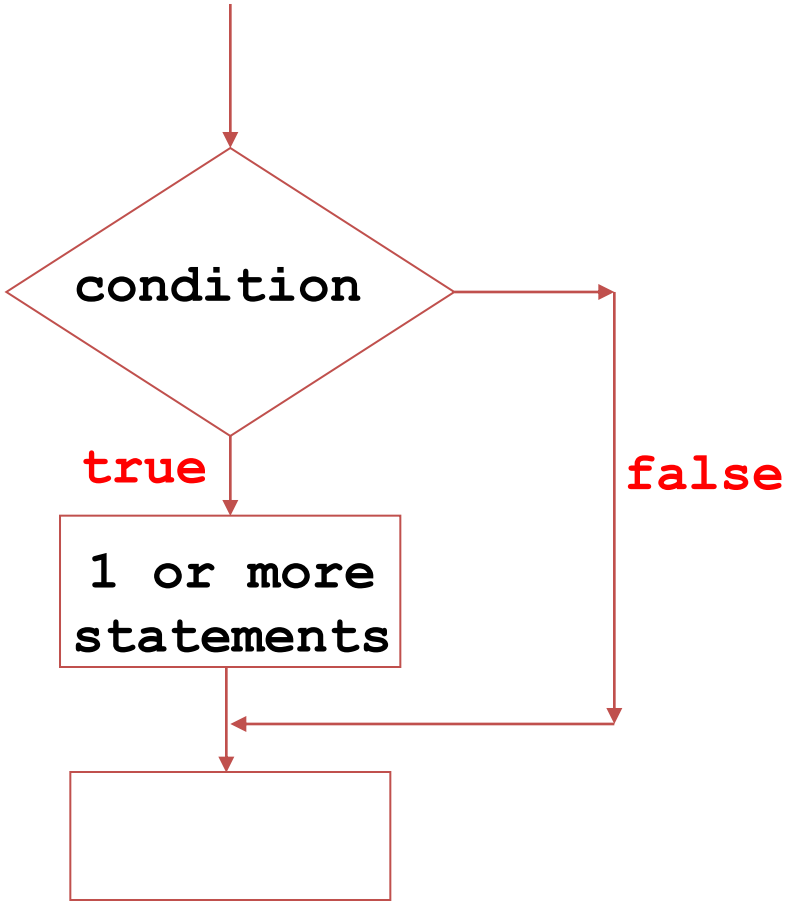
Lecture 3 (Conditional Statement)

Dr. Nelidya Md Yusoff

Razak Faculty of Technology and Informatics

Topics (Decision Making)

The `if` Statement
The `if/else` Statement
The `if/else if` Statement



Relational Operators

- Allow to compare numeric and char values and determine relative order
- Operators:

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

Relational Expressions

- Relational expressions are Boolean (*i.e.*, the value can be `true` or `false`)
- Examples:

  `12 > 5` is `true`

  `7 <= 5` is `false`

  if `x` is 10, then

  `x == 10` is `true`,

  `x != 8` is `true`, and

  `x == 8` is `false`

Relational Expressions

- Can be assigned to a variable

```
bool result = (x <= y);
```

- Assigns `0` for `false`, `1` for `true`
- Do not confuse `=` (assignment) and `==` (equal to)

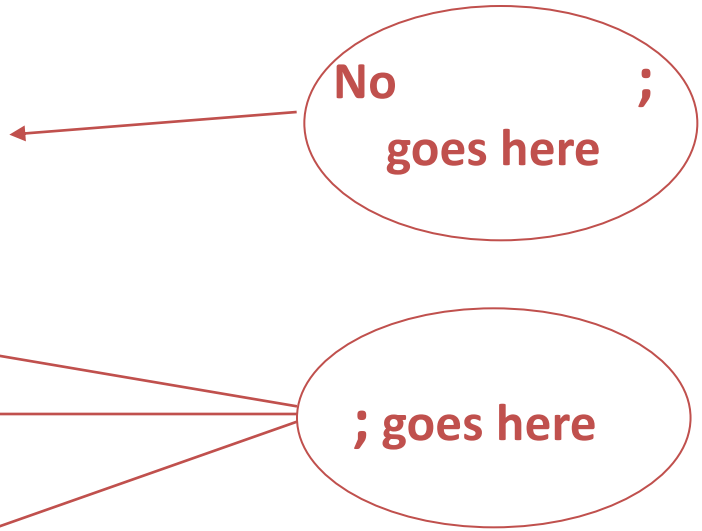
The `if` Statement

- Allows statements to be conditionally executed or skipped over
- Models the way we mentally evaluate situations

1. if (it is cold outside)
   wear a coat and wear a hat;
2. if (it is cold outside)
   wear a coat;
   wear a hat;

Format of the `if` Statement

```
if (condition){

   statement1;
   statement2;
        …
   statementn;
}
```



The block inside the braces is called the body of the `if` statement.

If there is only 1 statement in the body, the `{ }` may be omitted.

How the `if` Statement Works

- If (*condition*) is `true`, then the *statement(s)* in the body are executed.
- If (*condition*) is `false`, then the *statement(s)* are skipped.

`if` Statement in Program 4-2

Program 4-2

```
1  // This program averages three test scores
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  int main()
7  {
8     int score1, score2, score3;  // To hold three test scores
9     double average;              // To hold the average score
10
```

Continued…

`if` Statement in Program 4-2

Program 4-2 *(continued)*

```
11      // Get the three test scores.
12      cout << "Enter 3 test scores and I will average them: ";
13      cin >> score1 >> score2 >> score3;
14
15      // Calculate and display the average score.
16      average = (score1 + score2 + score3) / 3.0;
17      cout << fixed << showpoint << setprecision(1);
18      cout << "Your average is " << average << endl;
19
20      // If the average is greater than 95, congratulate the user.
21      if (average > 95)
22         cout << "Congratulations! That's a high score!\n";
23      return 0;
24  }
```

Program Output with Example Input Shown in Bold

```
Enter 3 test scores and I will average them: 80 90 70 [Enter]
Your average is 80.0
```

Program Output with Other Example Input Shown in Bold

```
Enter 3 test scores and I will average them: 100 100 100 [Enter]
Your average is 100.0
Congratulations! That's a high score!
```

Example `if` Statements

```
int score;
char grade;
if (score >= 60){
    cout << "You passed.\n";
}
if (score >= 90){
    grade = 'A';
    cout << "Wonderful job!\n";
}
```

`if` Statement Notes

- Do not place `;` after `(condition)`
- Don't forget the { } around a multi-statement body
- Place each `statement;` on a separate line after `(condition)`, indented
- 0 is `false`; any other value is `true`

- Relational expressions are not only the conditions that may be tested.
- Ex.
  - if (value){
    Cout << “It is true!”;
    }

    bool value = true;
  - if (x + y){
    Cout << “It is true!”;
    }

    int x = 10, y = 0;
  - if (pow(a, b)){
    Cout << “It is true!”;
    }

    double a; int b;

Exercise

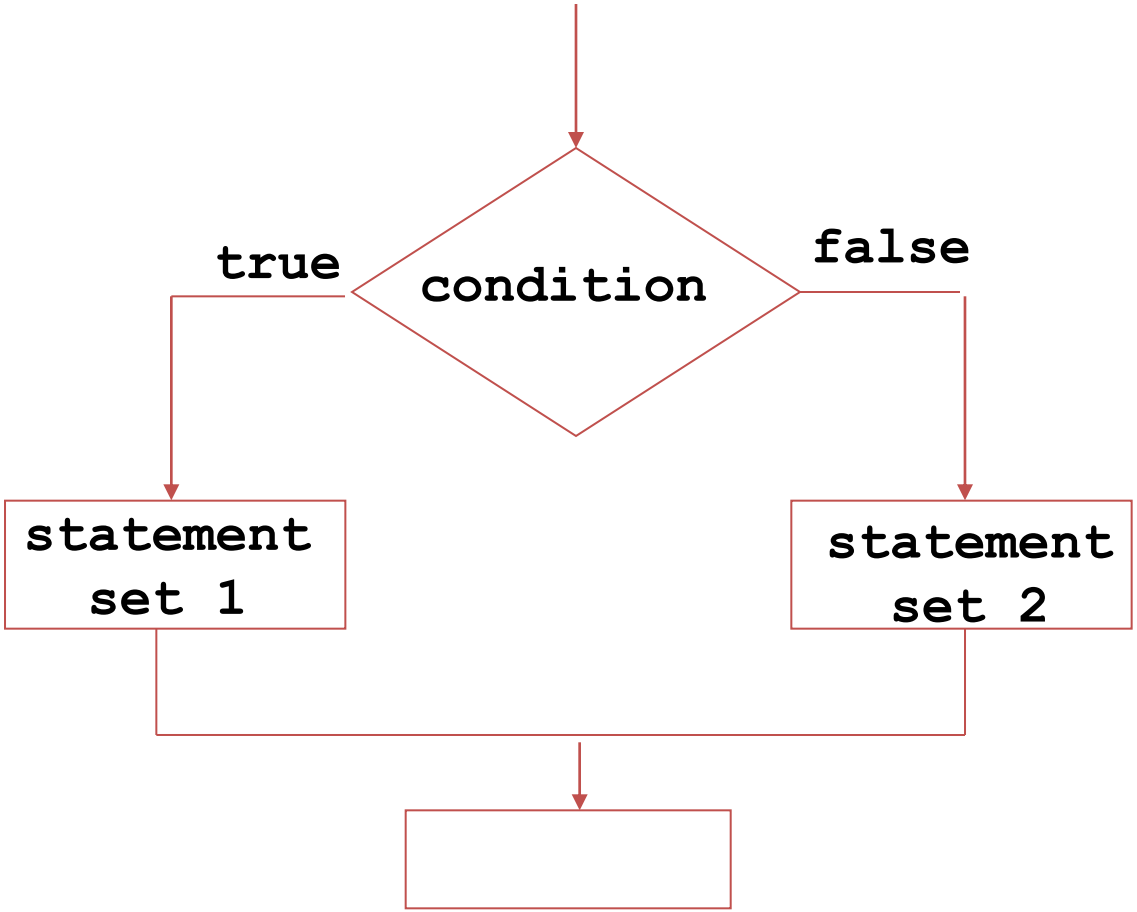
- if it is cold outside is true
  - Display “Please wear a coat”;
  - Display “Please wear a hat”;
  - Display “Please wear gloves”;

The `if/else` Statement

- Allows a choice between statements depending on whether `(`*`condition`*`)` is `true` or `false`
- Format:

```
if (condition){
    statement set 1;
}
else{
    statement set 2;
}
```

`if/else` Flow of Control



How the `if/else` Works

- If (*condition*) is `true`, *statement set 1* is executed and *statement set 2* is skipped.
- If (*condition*) is `false`, *statement set 1* is skipped and *statement set 2* is executed.

The `if/else` statement and Modulus Operator in Program 4-8

Program 4-8

```
1  // This program uses the modulus operator to determine
2  // if a number is odd or even. If the number is evenly divisible
3  // by 2, it is an even number. A remainder indicates it is odd.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     int number;
10
11    cout << "Enter an integer and I will tell you if it\n";
12    cout << "is odd or even. ";
13    cin >> number;
14    if (number % 2 == 0)
15       cout << number << " is even.\n";
16    else
17       cout << number << " is odd.\n";
18    return 0;
19 }
```

Program Output with Example Input Shown in Bold

```
Enter an integer and I will tell you if it
is odd or even. 17 [Enter]
17 is odd.
```

Example `if/else` Statements

```
if (score >= 60){
   cout << "You passed.\n";
}
else{
   cout << "You did not pass.\n";
}
if (intRate > 0){
     interest = loanAmt * intRate;
    cout << interest;
}
else{
    cout << "You owe no interest.\n";
}
```

Exercise

- *if it is cold outside is true*
  - Display “Please wear a coat”;
  - Display “Please wear a hat”;
  - Display “Please wear gloves”;
- *else*
  - Display “It’s hot today”;

The `if/else if` Statement

- Chain of `if` statements that test in order until one is found to be true
- Also models thought processes

"If it is raining, take an umbrella,
else, if it is windy, take a hat,
else, if it is sunny, take sunglasses."

if/else if Format

```
if (condition 1){
  statement set 1;
}
else if (condition 2){
  statement set 2;
}
        …
else if (condition n){
  statement set n;
}
```

Using a Trailing `else`

- Chain of if statements. They perform their tests one after the other until one of them is found to be true.
- Used with `if/else if` statement when all of the conditions are false
- Provides a default statement or action
- Can be used to catch invalid values or handle other exceptional situations

Example `if/else if` with Trailing `else`

```
if (age >= 21){
   cout << "Adult";
}
else if (age >= 13){
   cout << "Teen";
}
else if (age >= 2){
   cout << "Child";
}
else{
   cout << "Baby";
}
```

Nested `if` Statements

- An `if` statement that is part of the `if` or `else` part of another `if` statement
- Can be used to evaluate > 1 data item or condition

```
if (score < 100){
    if (score > 90){
        grade = "A+";
     }
     else if (score > 80){
         grade = "A";
     }
}
```

Notes on Coding Nested `if`s

- An `else` matches the nearest `if` that does not have an `else`

```
if (score < 100){
    if (score > 90){
       grade = 'A';
     }
    else ...  // goes with second if,
              // not first one
 }
```

- Proper indentation aids comprehension

Exercise

- if it is cold outside is true
  - Display “Please wear a coat”;
  - Display “Please wear a hat”;
  - Display “Please wear gloves”;
- Else if it is raining today
  - Display “Please wear rain coat”;
- Else
  - Display “It’s hot today”;

Logical Operators

Connect two or more relational expressions into one or reverse the logic of an expression.

Operators, Meaning, and Explanation

&&	AND	New relational expression is true if both expressions are true
||	OR	New relational expression is true if either expression is true
!	NOT	Reverses the value of an expression; true expression becomes false, false expression becomes true

Logical Operator Examples

```
int x = 12, y = 5, z = -4;
```

`(x > y) && (y > z)`	`true`
`(x > y) && (z > y)`	`false`
`(x <= z) || (y == z)`	`false`
`(x <= z) || (y != z)`	`true`
`!(x >= z)`	`false`

Logical Precedence

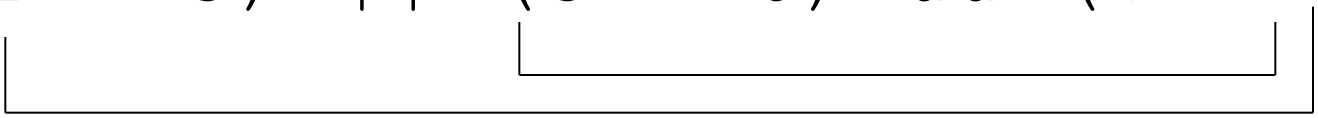
Highest `!` (!(x + y), (!x + y)

`&&`

Lowest `||`

Example:

```
(2 < 3) || (5 > 6) && (7 > 8)
```

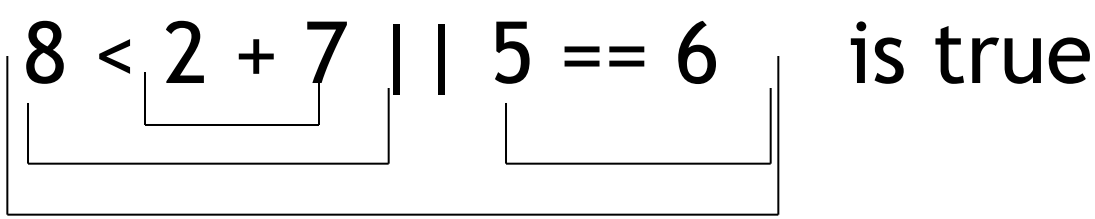


is true because AND is evaluated before OR

More on Precedence

Highest	arithmetic operators
↓	relational operators
Lowest	logical operators

Example:



is true

Checking Numeric Ranges with Logical Operators

- Used to test if a value is within a range

```
if (grade >= 0 && grade <= 100)
   cout << "Valid grade";
```

- Can also test if a value lies outside a range

```
if (grade <= 0 || grade >= 100)
   cout << "Invalid grade";
```

- Cannot use mathematical notation

```
if (0 <= grade <= 100) //Doesn't
                       //work!
```

Example

```
Please input grade.
if (grade >= 0 && grade <= 100) {
     cout << "Valid grade";
     if (grade > 90){
          cout << “You have obtained A+”;
     }
     else if (grade > 80){
          cout << “You have obtained A”;
     }
     else {
          cout << “You have to work hard”;
     }
}
else if (grade <= 0 || grade >= 100)
     cout << "Invalid grade";
```

Exercise.1

- Write a C++ program that calculates the gross pay of hari kurban month:
  - If the salary is over RM5000 then the bonus to be paid is RM500
  - Otherwise, if the salary is over RM2000 then the bonus to be paid is RM300
  - Otherwise, the salary is over RM1000 then the bonus to be paid is RM200
  - Else, the bonus to be paid is RM100

Exercise.2

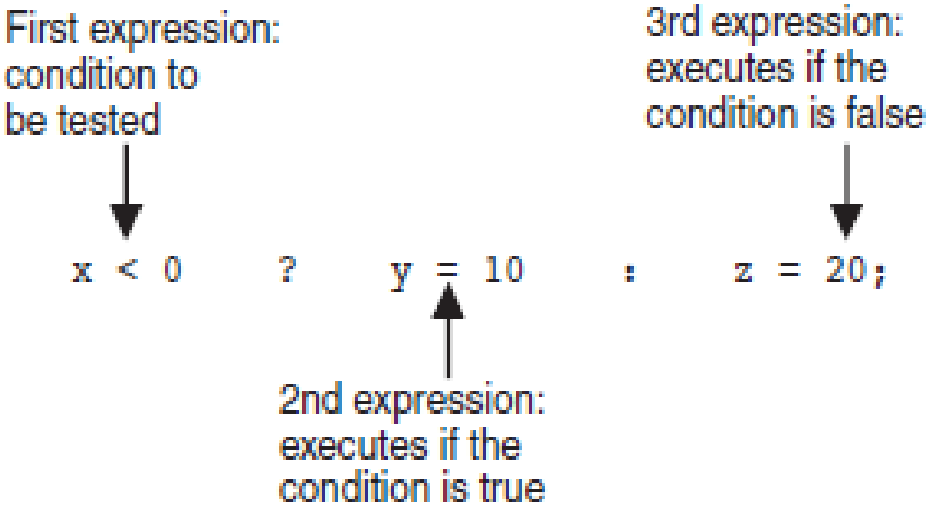
- Write a C++ program that asks the user to enter two numbers. The program should use the conditional operator to determine which number is smaller and which is the larger.

Exercise.3

- Write a C++ program that asks the user to enter a number within the range of 1 through 10.
  - Display the Roman numeral version of that number.

The Conditional Operator

- Can use to create short `if/else` statements
- Format: `expr ? expr : expr;`



The `switch` Statement

- Used to select among statements from several alternatives
- May sometimes be used instead of `if/else if` statements

switch Statement Format

```
switch (IntExpression)
{
  case exp1:
                statement set 1;
  case exp2:
                statement set 2;
  ...
  case expn:
                statement set n;
  default:
                statement set
  n+1;
}
```

`switch` Statement Requirements

1) *`IntExpression`* must be a `char` or an integer variable or an expression that evaluates to an integer value
2) *`exp1`* through *`expn`* must be constant integer type expressions and must be unique in the `switch` statement
3) `default` is optional but recommended

How the `switch` Statement Works

1) *`IntExpression`* is evaluated
2) The value of *`intExpression`* is compared against *`exp1`* through *`expn`*.
3) If *`IntExpression`* matches value *`expi`*, the program branches to the statement(s) following *`expi`* and continues to the end of the `switch`
4) If no matching value is found, the program branches to the statement after `default:`

The `break` Statement

- Used to stop execution in the current block
- Also used to exit a `switch` statement
- Useful to execute a single `case` statement without executing statements following it

Example `switch` Statement

```
switch (gender){
   case 'f':
              cout << "female";
                 break;
   case 'm':
              cout << "male";
                 break;
   default :
              cout << "invalid
gender";
}
```

The `switch` Statement in Program 4-23

Program 4-23

```
1  // The switch statement in this program tells the user something
2  // he or she already knows: the data just entered!
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     char choice;
9
10    cout << "Enter A, B, or C: ";
11    cin >> choice;
12    switch (choice)
13    {
14       case 'A': cout << "You entered A.\n";
15                 break;
16       case 'B': cout << "You entered B.\n";
17                 break;
18       case 'C': cout << "You entered C.\n";
19                 break;
20       default:  cout << "You did not enter A, B, or C!\n";
21    }
22    return 0;
23 }
```

Program Output with Example Input Shown in Bold

```
Enter A, B, or C: B [Enter]
You entered B.
```

Program Output with Example Input Shown in Bold

```
Enter A, B, or C: F [Enter]
You did not enter A, B, or C!
```

Program 4-25

```
1  // This program is carefully constructed to use the "fall through"
2  // feature of the switch statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int modelNum;  // Model number
9
10    // Get a model number from the user.
11    cout << "Our TVs come in three models:\n";
12    cout << "The 100, 200, and 300. Which do you want? ";
13    cin >> modelNum;
14
15    // Display the model's features.
16    cout << "That model has the following features:\n";
17    switch (modelNum)
18    {
19       case 300: cout << "\tPicture-in-a-picture.\n";
20       case 200: cout << "\tStereo sound.\n";
21       case 100: cout << "\tRemote control.\n";
22                 break;
23       default:  cout << "You can only choose the 100,";
24                 cout << "200, or 300.\n";
25    }
26    return 0;
27 }
```

Continued…

Program Output with Example Input Shown in Bold

```
Our TVs come in three models:
The 100, 200, and 300. Which do you want? 100 [Enter]
That model has the following features:
    Remote control.
```

Program Output with Example Input Shown in Bold

```
Our TVs come in three models:
The 100, 200, and 300. Which do you want? 200 [Enter]
That model has the following features:
    Stereo sound.
    Remote control.
```

Program Output with Example Input Shown in Bold

```
Our TVs come in three models:
The 100, 200, and 300. Which do you want? 300 [Enter]
That model has the following features:
    Picture-in-a-picture.
    Stereo sound.
    Remote control.
```

Program Output with Example Input Shown in Bold

```
Our TVs come in three models:
The 100, 200, and 300. Which do you want? 500 [Enter]
That model has the following features:
You can only choose the 100, 200, or 300.
```

Ladder

```
#include <iostream>
using namespace std;
int main()
{
int x;
for(x=0; x<6; x++) {
if(x==1) cout << "x is one\n";
else if(x==2) cout << "x is two\n";
else if(x==3) cout << "x is three\n";
else if(x==4) cout << "x is four\n";
else cout << "x is not between 1 and 4\n";
}
return 0;
}
```

Write a program that asks the user to enter a number within the range of 1 through 100 (only multiple of 10). Use a switch statement to display the Roman numeral version of that number.

MRSL 1163
SCIENTIFIC COMPUTING FOR SYSTEM ENGINEER

Lecture 4 (Loop)

Dr Nelidya Md Yusoff

Razak Faculty of Technology and Informatics

Topics

- The Increment and Decrement Operators
- Introduction to Loops: The `while` Loop
- Using the `while` loop for Input Validation
- Counters
- The `do-while` loop
- The `for` loop

The Increment and Decrement Operators

- ++ is the increment operator.

  It adds one to a variable.

  `val++;` is the same as `val = val + 1;`

- ++ can be used before (prefix) or after (postfix) a variable:

  ```
  ++val;      val++;
  ```

The Increment and Decrement Operators

- `--` is the decrement operator.

  It subtracts one from a variable.

  `val--;` is the same as `val = val - 1;`

- `--` can be also used before (prefix) or after (postfix) a variable:

  `--val;` `val--;`

Increment and Decrement Operators in Program 5-1

Program 5-1

```
1  // This program demonstrates the ++ and -- operators.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int num = 4;    // num starts out with 4.
8
9     // Display the value in num.
10    cout << "The variable num is " << num << endl;
11    cout << "I will now increment num.\n\n";
12
13    // Use postfix ++ to increment num.
14    num++;
15    cout << "Now the variable num is " << num << endl;
16    cout << "I will increment num again.\n\n";
17
18    // Use prefix ++ to increment num.
19    ++num;
20    cout << "Now the variable num is " << num << endl;
21    cout << "I will now decrement num.\n\n";
22
23    // Use postfix -- to decrement num.
24    num--;
25    cout << "Now the variable num is " << num << endl;
26    cout << "I will decrement num again.\n\n";
27
```

Continued…

Increment and Decrement Operators in Program 5-1

Program 5-1 *(continued)*

```
28      // Use prefix -- to increment num.
29      --num;
30      cout << "Now the variable num is " << num << endl;
31      return 0;
32  }
```

Program Output

```
The variable num is 4
I will now increment num.

Now the variable num is 5
I will increment num again.

Now the variable num is 6
I will now decrement num.

Now the variable num is 5
I will decrement num again.

Now the variable num is 4
```

Prefix vs. Postfix

- `++` and `--` operators can be used in complex statements and expressions
- In prefix mode (`++val`, `--val`) the operator increments or decrements, *then* returns the value of the variable
- In postfix mode (`val++`, `val--`) the operator returns the value of the variable, *then* increments or decrements

Prefix vs. Postfix - Examples

```
int num, val = 12;
cout << val++; // displays 12,
               // val is now 13;
cout << ++val; // sets val to 14,
                 // then displays it
num = --val;   // sets val to 13,
               // stores 13 in num
num = val--;   // stores 13 in num,
               // sets val to 12
```

Notes on Increment and Decrement

- Can be used in expressions:

```
result = num1++ + --num2;
```

- Must be applied to something that has a location in memory. Cannot have:

```
result = (num1 + num2)++;
```

- Can be used in relational expressions:

```
if (++num > limit)
```

pre- and post-operations will cause different comparisons

Loops

- A loop is a part of programs that repeats
- A loop is a control structure that causes a statement or group of statements to repeat.
- C++ has three looping control structures:
  - foor loop
  - while loop
  - do-while loop

The `for` Loop

- The foor loop is a pretest loop that combines the initialization, testing, and updating of a loop control variable in a single loop header

- Foor loop is a count-controlled loop. A loop that repeats a specific number of times is known as count-controlled loop.

- General Format:

```
for(initialization; test; update){
    statement;
}
```

- No semicolon after the `update` expression or after the `)`

`for` Loop - Mechanics

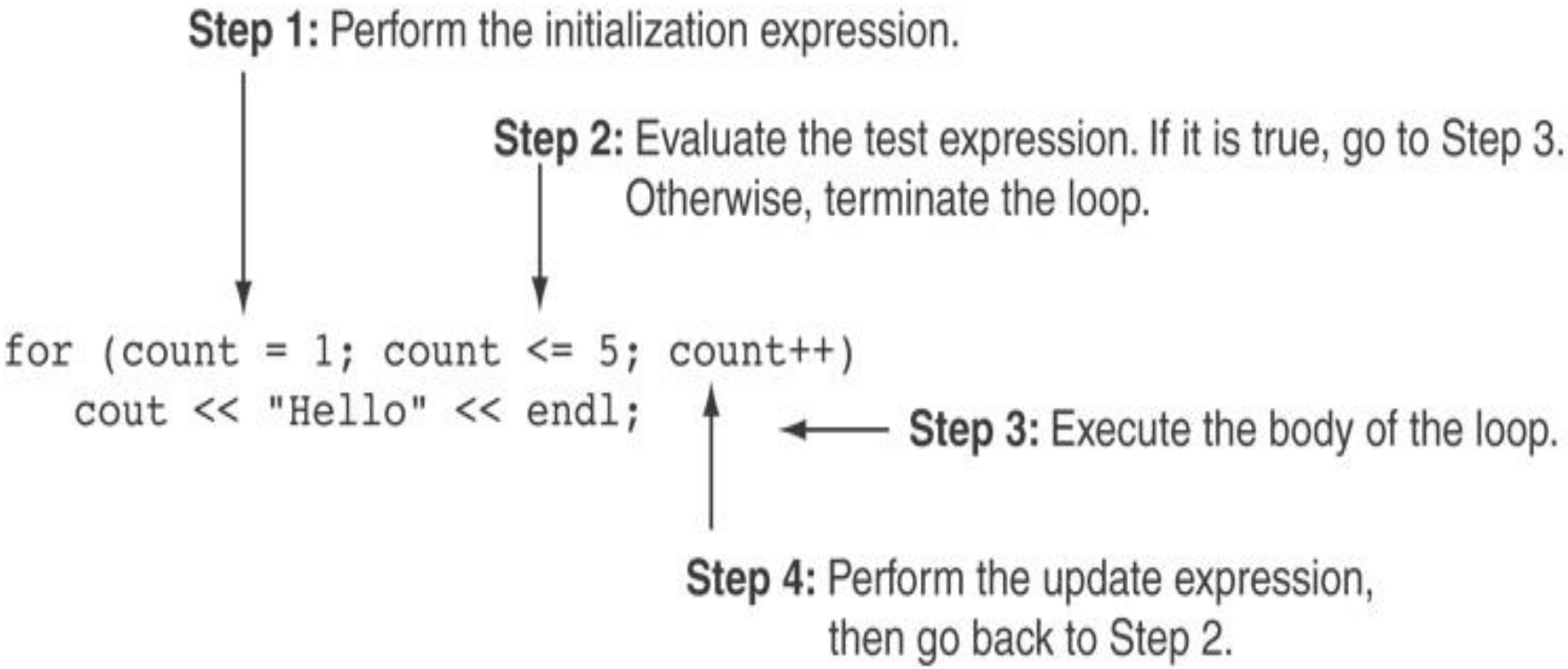
```
for(initialization; test; update)
        statement; // or block in { }
```

1) Perform *initialization*
2) Evaluate *test* expression
   - If `true`, execute *statement*
   - If `false`, terminate loop execution
3) Execute *update*, then re-evaluate *test* expression

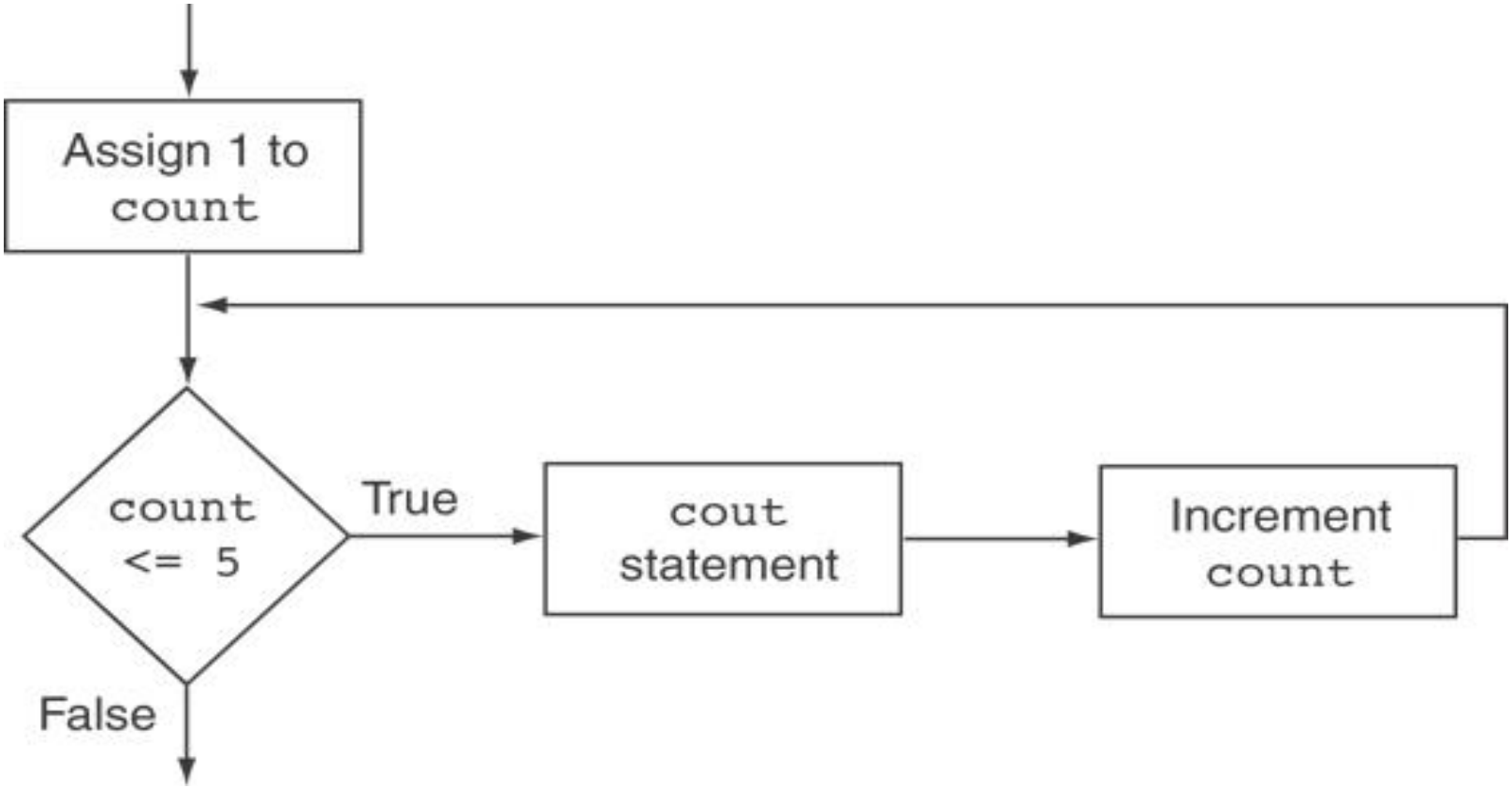
`for` Loop - Example

```
int count;

for (count = 0; count <= 10; count++){
   cout << "Hello MJIIT" << endl;
}
```

A Closer Look at the Previous Example



Flowchart for the Previous Example



A `for` Loop in Program 5-9

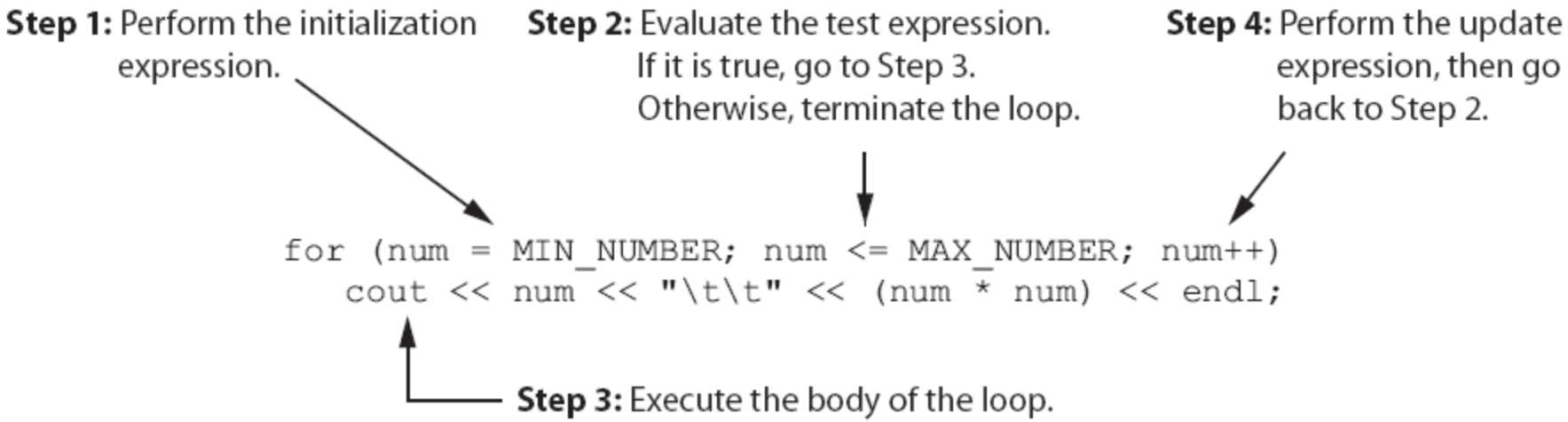
Program 5-9

```
1  // This program displays the numbers 1 through 10 and
2  // their squares.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int MIN_NUMBER = 1,    // Starting value
9               MAX_NUMBER = 10;   // Ending value
10    int num;
11
12    cout << "Number Number Squared\n";
13    cout << "-------------------------\n";
14
15    for (num = MIN_NUMBER; num <= MAX_NUMBER; num++)
16       cout << num << "\t\t" << (num * num) << endl;
17
18    return 0;
19 }
```

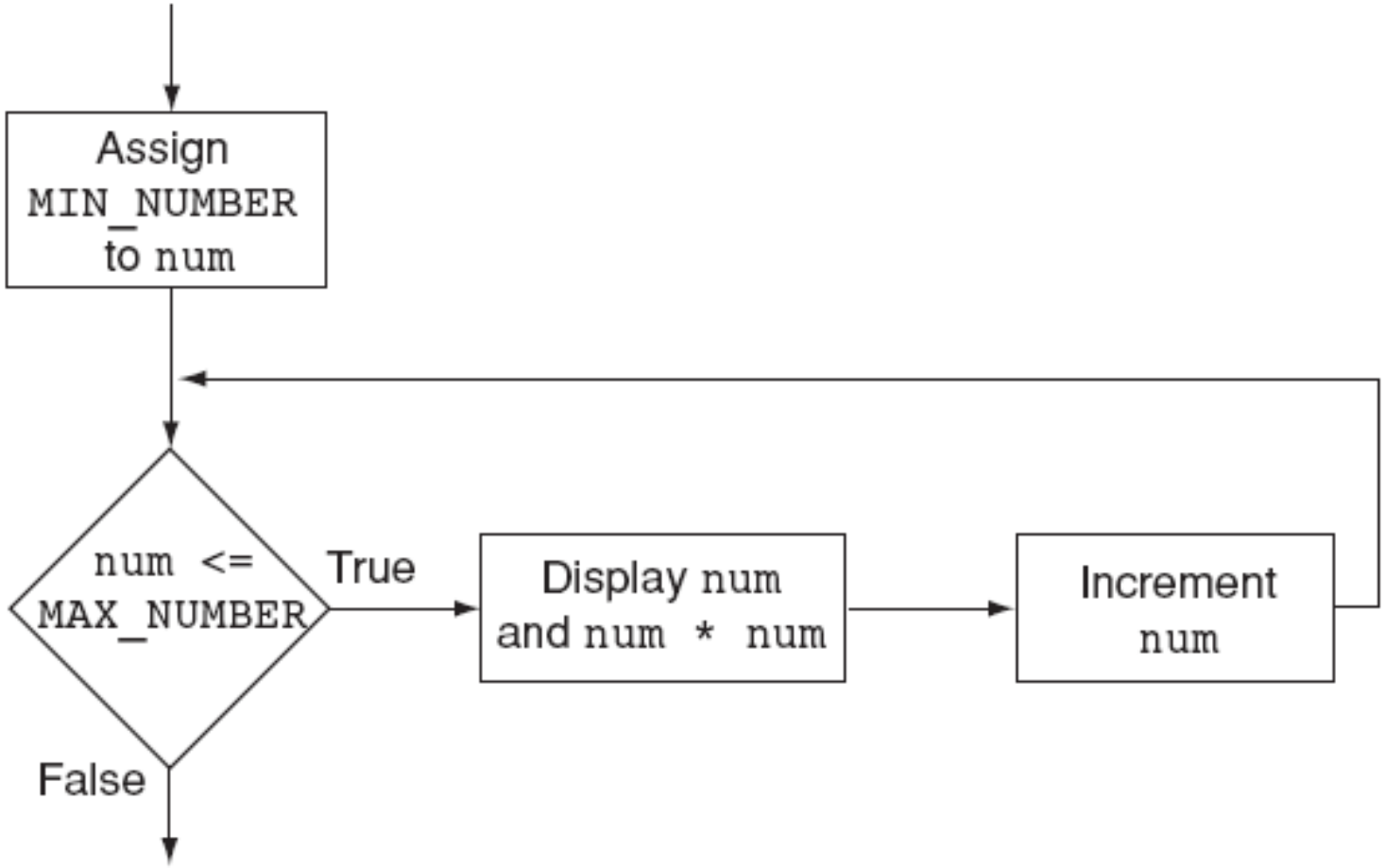
Continued…

Program Output

```
Number  Number Squared
----------------------
1               1
2               4
3               9
4               16
5               25
6               36
7               49
8               64
9               81
10              100
```


Step 1: Perform the initialization expression.
Step 2: Evaluate the test expression. If it is true, go to Step 3. Otherwise, terminate the loop.
Step 4: Perform the update expression, then go back to Step 2.
for (num = MIN_NUMBER; num <= MAX_NUMBER; num++)
cout << num << "\t\t" << (num * num) << endl;
Step 3: Execute the body of the loop.

Flowchart for Lines 15 through 16 in Program 5-9



When to Use the `for` Loop

- In any situation that clearly requires
  - an initialization
  - a false condition to stop the loop
  - an update to occur at the end of each iteration

The `for` Loop is a Pretest Loop

- The for loop tests its test expression before each iteration, so it is a pretest loop.
- Check the following loop (?):

```
for (count = 11; count <= 10; count++){
    cout << "Hello" << endl;
}
```

`for` Loop - Modifications

- You can have multiple statements in the *`initialization`* expression. Separate the statements with a comma:

```
int x, y;
for (x=1, y=1; x <= 5; x++){
    cout << x << " plus " << y
         << " equals " << (x+y)
         << endl;
}
```

Initialization Expression

`for` Loop - Modifications

- You can also have multiple statements in the *test* expression. Separate the statements with a comma:

Test Expression

```
int x, y, result;
for (x=1, y=5; x <= 5 || y <8; x++, y++)
{
    result = x + y;
    cout << x << " plus " << y
         << " equals " << (x+y)
         << endl;

  }
 cout << result;
```

`for` Loop - Modifications

- You can omit the *`initialization`* expression if it has already been done:

```
int sum = 0, num = 1;
for (; num <= 10; num++){
    sum += num;
// sum = sum + num;
}
```

`for` Loop - Modifications

- You can declare variables in the *`initialization`* expression:

```
int sum = 0;
for (int num = 0; num <= 10; num++){
      sum += num;
}
```

The scope of the variable `num` is the `for` loop.

Exercise-1, 2, 3

- Write a program that displays odd and even numbers between 1 and 10 (for/while/do-while Loop)
- Write a loop that displays the following set of numbers (for/while/do-while Loop) :

  0, 5, 15, 20, 25, 35, 40, …, 100

Exercise-4

- Write a program that displays prime numbers between 1 and 100.

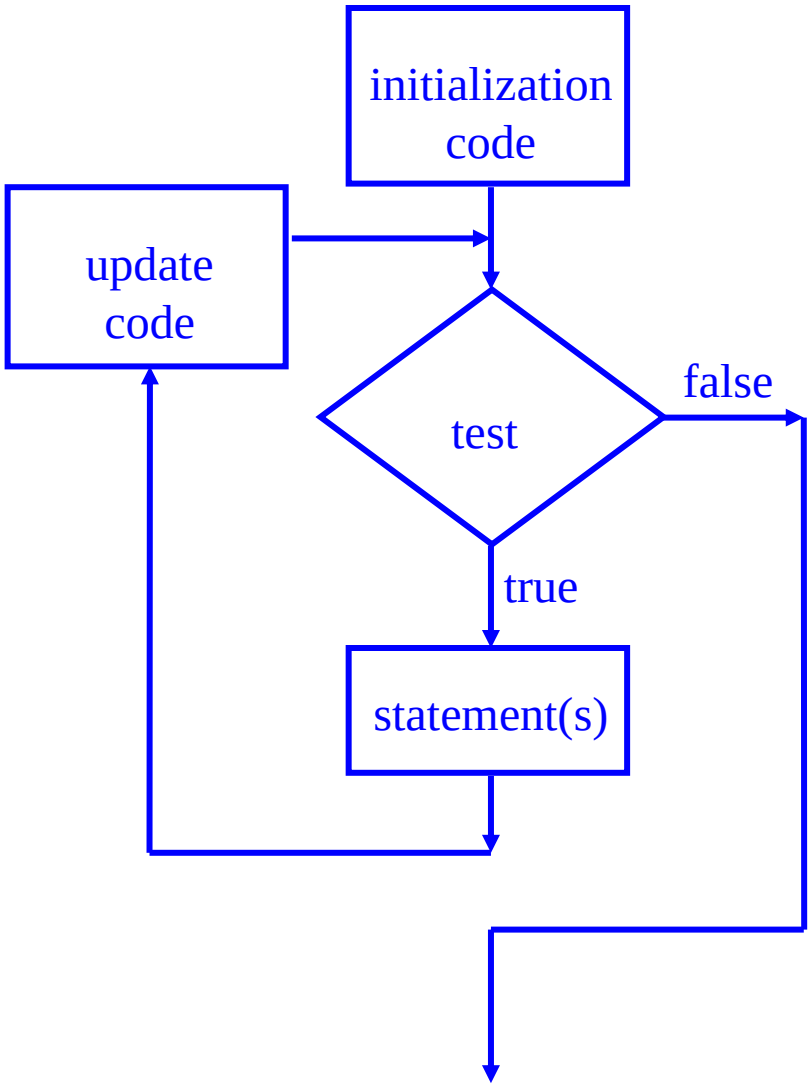
Nested Loops (Prime Number)

```
for (num=1; num<=10; num++){  //outer
  for (x=2; x < num; x++){//inner
    if ( num % x ==0){
        cout << num << ": is not a prime
  number"<<endl;
        break;
    }
  }
  if ( num == x){
    cout << num << ": IS a PRIME
  number"<<endl;
  }
}
```

Ex.

- Write a program that displays odd number numbers between 0 and 20.

`for` Loop Flow of Control

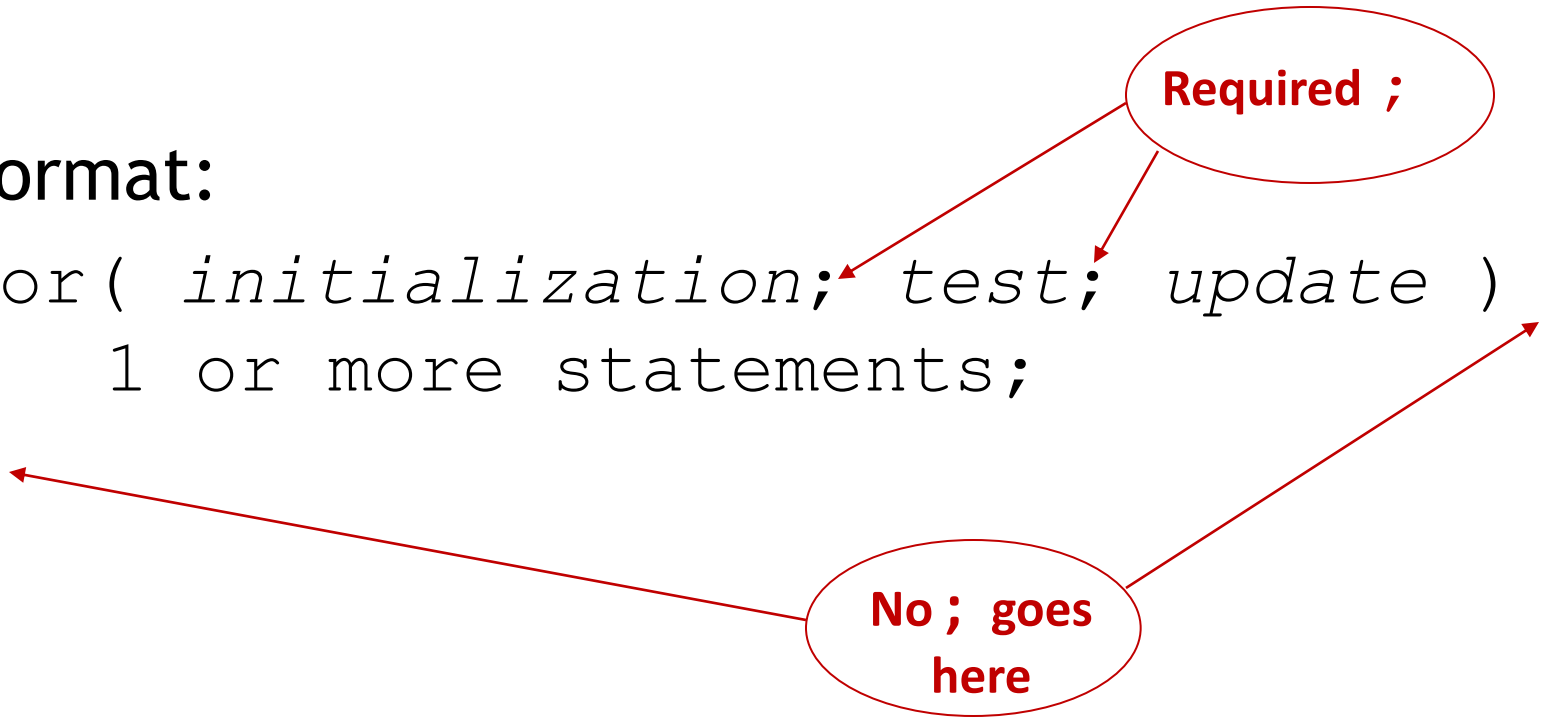


The `for` Loop

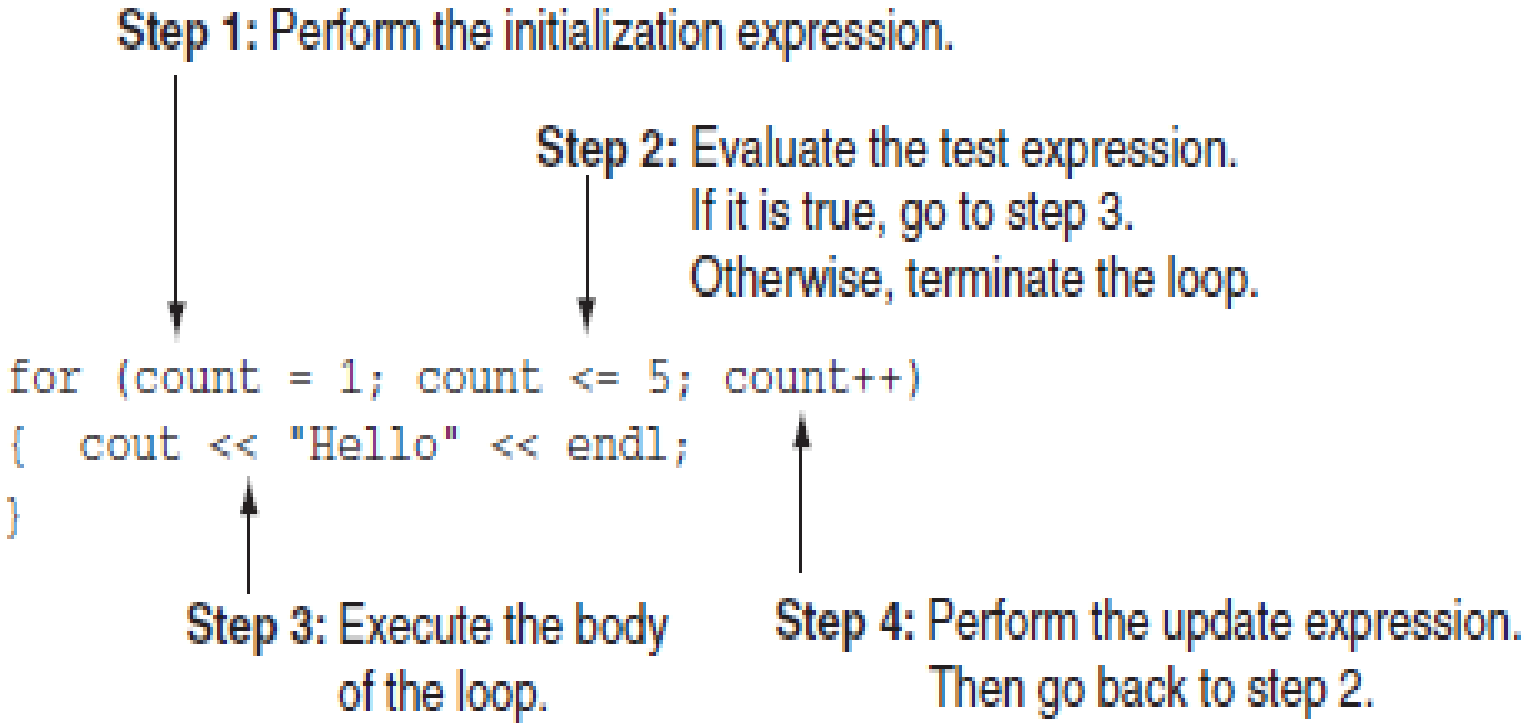
- Pretest loop that executes zero or more times
- Useful for counter-controlled loop

- Format:

```
for( initialization; test; update )
{    1 or more statements;
}
```



`for` Loop Mechanics



`for` Loop Example

```
int sum = 0, num;
for (num = 1; num <= 10; num++){
   sum += num;
}
cout << "Sum of numbers 1 - 10 is
  "
       << sum << endl;
```

`for` Loop Notes

- If *`test`* is false the first time it is evaluated, the body of the loop will not be executed
- The update expression can increment or decrement by any amount
- Variables used in the initialization section should not be modified in the body of the loop

`for` Loop Modifications

- Can define variables in initialization code
  - Their scope is the `for` loop
- Initialization and update code can contain more than one statement
  - Separate statements with commas
- Example:

```
for (int sum = 0, num = 1; num <= 10; num++)
    sum += num;
```

More `for` Loop Modifications
(These are NOT Recommended)

- Can omit *`initialization`* if already done

```
int sum = 0, num = 1;
for (; num <= 10; num++)
    sum += num;
```

- Can omit *`update`* if done in loop

```
for (sum = 0, num = 1; num <= 10;)
    sum += num++;
```

- Can omit *`test`* - may cause an infinite loop

```
for (sum = 0, num = 1; ; num++)
    sum += num;
```

- Can omit loop body if all work is done in header

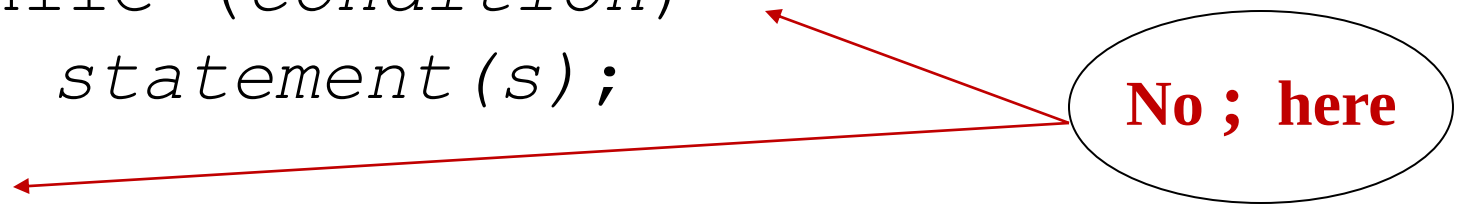
`while` Loop is a Pretest Loop

- `while` is a pretest loop (*`condition`* is evaluated before the loop executes)
- If the condition is initially false, the statement(s) in the body of the loop are never executed
- If the condition is initially true, the statement(s) in the body continue to be executed until the condition becomes false

The `while` Loop

- Loop: part of program that may execute > 1 time (*i.e.*, it repeats)
- `while` loop format:

```
while (condition)
{   statement(s);
}
```



- The `{}` can be omitted if there is only one statement in the body of the loop

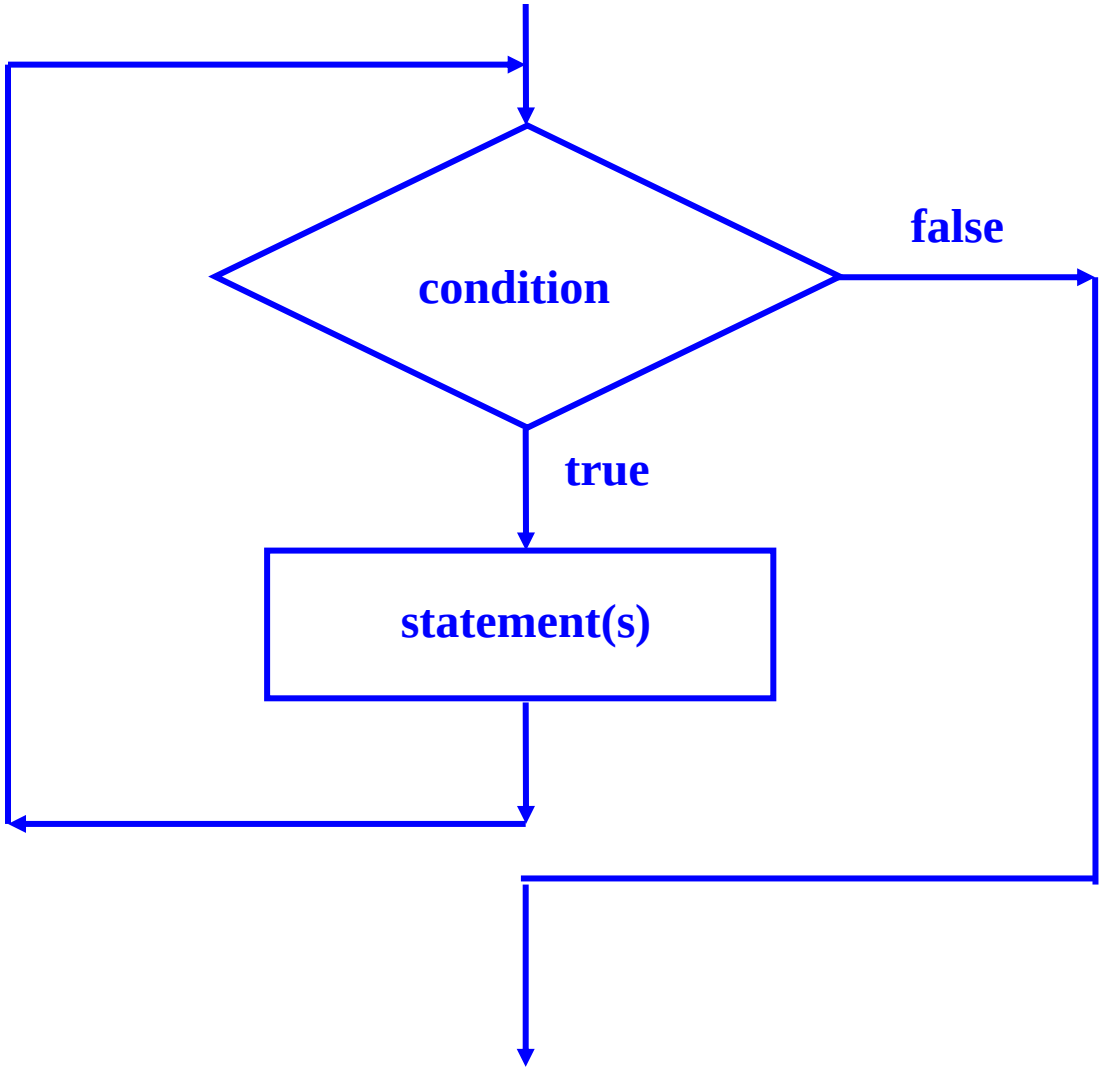
How the `while` Loop Works

```
while (condition)
{   statement(s);
}
```

condition is evaluated

- if it is true, the *statement(s)* are executed, and then *condition* is evaluated again
- if it is false, the loop is exited

`while` Loop Flow of Control



`while` Loop Example

```
int val = 5;
while (val >= 0)
{    cout << val << "   ";
     val--;
}
```

- produces output:

```
  5   4   3   2   1   0
```

Exiting the Loop

- The loop must contain code to allow *condition* to eventually become `false` so the loop can be exited
- Otherwise, you have an infinite loop (*i.e.*, a loop that does not stop)
- Example infinite loop:

```
x = 5;
while (x > 0)          // infinite loop
 because
    cout << x;         // x is always > 0
```

Common Loop Errors

- Don't forget the { } :

```
int numEntries = 1;
while (numEntries <=3)
    cout << "Still working … ";
    numEntries++; // not in the loop body
```

- Don't use = when you mean to use ==

```
while (numEntries = 3)  // always true
{
    cout << "Still working … ";
    numEntries++;
}
```

Using the `while` Loop for Input Validation

Loops are an appropriate structure for validating user input data

1. Prompt for and read in the data.
2. Use a `while` loop to test if data is valid.
3. Enter the loop only if data is not valid.
4. Inside the loop, display error message and prompt the user to re-enter the data.
5. The loop will not be exited until the user enters valid data.

Input Validation Loop Example

```
cout << "Enter a number (1-100) and"
     << " I will guess it. ";
cin  >> number;

while (number < 1 || number > 100){

    cout << "Number must be between 1 and 100."
         << " Re-enter your number. ";
    cin  >> number;
}
// Code to use the valid number goes here.
```

Letting the User Control the Loop

- Program can be written so that user input determines loop repetition
- Can be used when program processes a list of items, and user knows the number of items
- User is prompted before loop. Their input is used to control number of repetitions

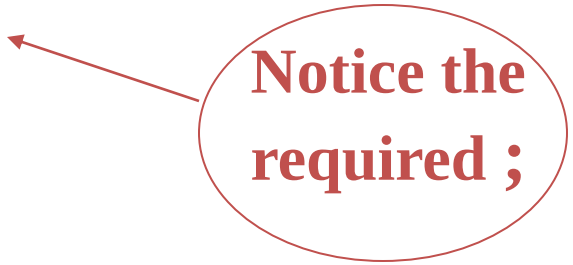
User Controls the Loop Example

```
int num, limit;
cout << "Table of squares\n";
cout << "How high to go? ";
cin  >> limit;
cout << "\n\nnumber square\n";
num = 1;
while (num <= limit)
{  cout << setw(5) << num << setw(6)
        << num*num << endl;
   num++;
}
```

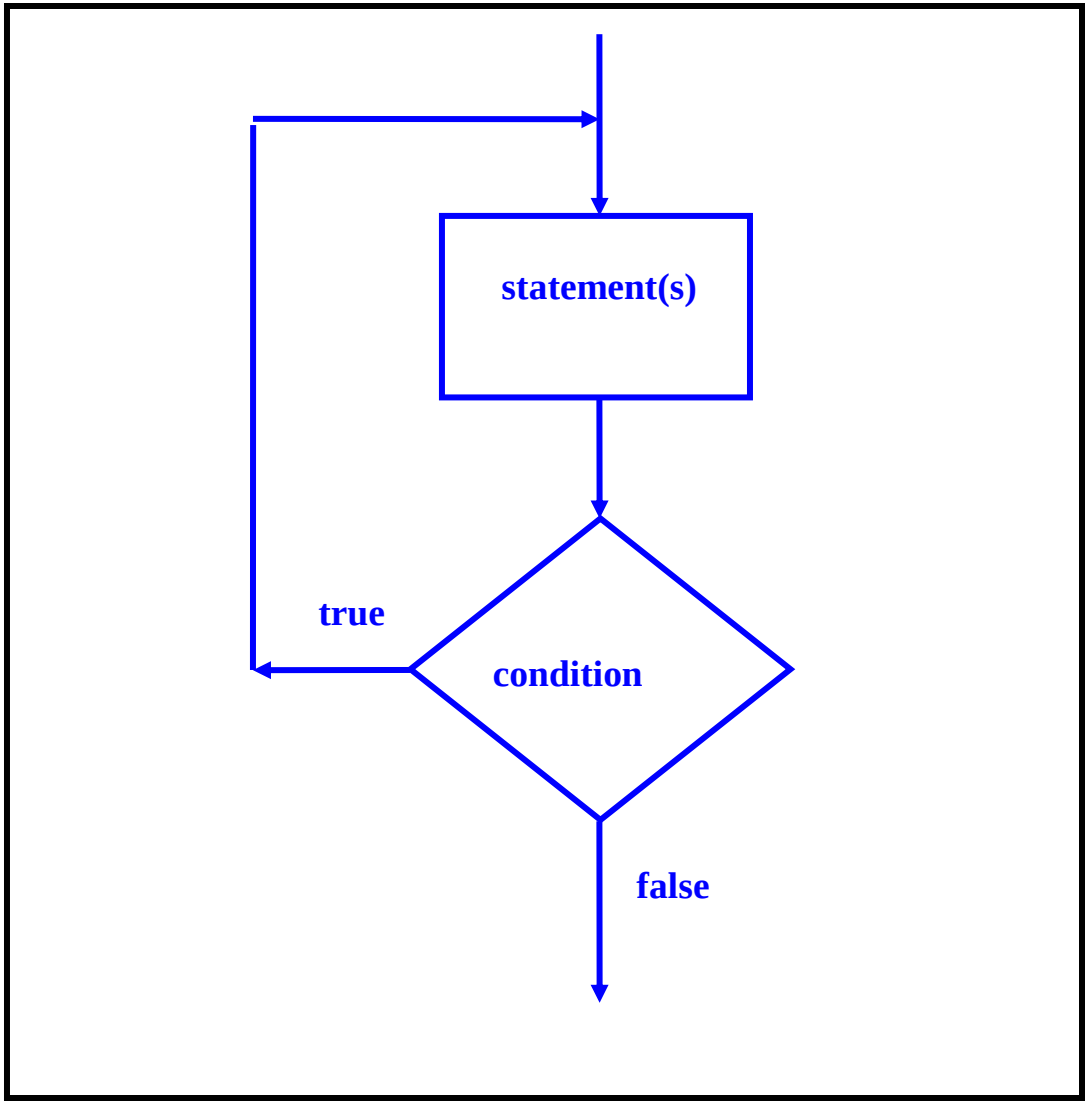
The `do-while` Loop

- `do-while`: a post test loop (*condition* is evaluated after the loop executes)
- Format:

```
do {
        1 or more statements;
} while (condition);
```



`do-while` Flow of Control



do-while Loop Notes

- Loop always executes at least once
- Execution continues as long as *condition* is `true`; the loop is exited when *condition* becomes `false`
- Useful in menu-driven programs to bring user back to menu to make another choice

Example.

```
int sum = 0, num = 1; // sum is the
char again;
do {
        sum += num;
        num++;

        cin >> again;
} while (again == 'Y' || again == 'y');
```

Exercise-1, 2, 3

- Write a program that displays odd and even numbers between 1 and 10 (for/while/do-while Loop)
- Write a loop that displays the following set of numbers (for/while/do-while Loop) :

  0, 5, 15, 20, 25, 35, 40, …, 100

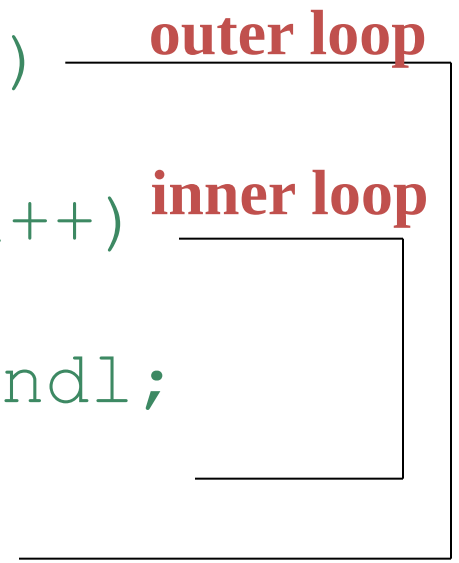
Deciding Which Loop to Use

- `while`: pretest loop (loop body may not be executed at all)
- `do-while`: post test loop (loop body will always be executed at least once)
- `for`: pretest loop (loop body may not be executed at all); has initialization and update code; is useful with counters or if precise number of repetitions is known

Nested Loops

- A **nested loop** is a loop inside the body of another loop
- Example:

```
for (row = 1; row <= 3; row++)
{
   for (col = 1; col <= 3; col++)
   {
      cout << row * col << endl;
   }
}
```



Notes on Nested Loops

- Inner loop goes through all its repetitions for each repetition of outer loop
- Inner loop repetitions complete sooner than outer loop
- Total number of repetitions for inner loop is product of number of repetitions of the two loops. In previous example, inner loop repeats 9 times

Nested Loops (Prime Number)

```
for (num=1; num<=10; num++){  //outer
  for (x=2; x < num; x++){//inner
    if ( num % x ==0){
        cout << num << ": is not a prime
  number"<<endl;
        break;
    }
  }
  if ( num == x){
    cout << num << ": IS a PRIME
  number"<<endl;
  }
}
```

Exercise 4

- Write a program by using a nested loops that displays prime numbers between 1 and 100.

Exercise 4 - answer

```
using namespace std;

int main()
{
        int x;
        for (int num=1; num<=100; num++){          //outer loop
                for (x=2; x < num; x++){           //inner loop
                        if ( num % x ==0){
                        cout << num <<": is not a prime number"<<endl;
                        break;
                }
        }
        if ( num == x){
        cout << num <<": IS a PRIME number"<<endl;
        }
}

return 0;
}
```

Breaking Out of a Loop

- Can use `break` to terminate execution of a loop
- Use sparingly if at all – makes code harder to understand
- When used in an inner loop, terminates that loop only and returns to the outer loop

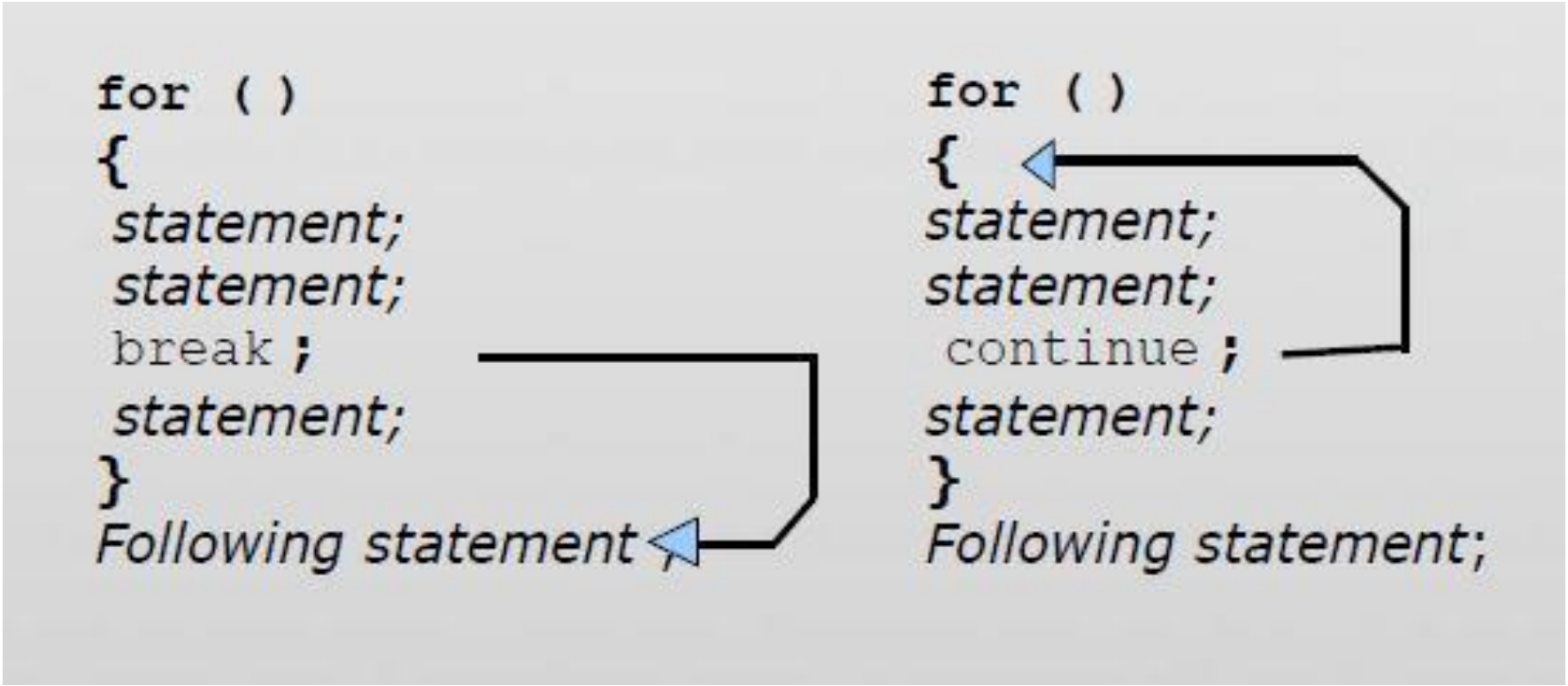
The `continue` Statement

- Can use `continue` to go to end of loop and prepare for next repetition
  - `while` and `do-while` loops go to test and repeat the loop if test condition is true
  - `for` loop goes to update step, then tests, and repeats loop if test condition is true
- Use sparingly - like `break`, can make program logic hard to follow

The `break` and `continue` Statements

- A statement `break` and `continue` are **important in loop usage**.
- `break` will stop the loop i.e. the program goes straight to the following statement.
- `continue` stop the current iteration and continue to the next iteration.

The `break` and `continue` Statements



The `break` and `continue` Statements - Example

```
{
  int count,total,totalS,num;
  count=1;
  total=0;
  while (count <=10)
  {
      cout<<"Please enter number "<<count<<" :";
      cin>> num;
      if (num<=0)
      {
         cout<<"You enter a wrong number"<<endl;
         continue;
      }
```

The break and continue Statements - Example

```
        count=count+1;
        totalT=total        //totalT=Temperory total
        total=total+num;
        if (total>20)
        {
              cout<<"\nNumber is enough." <<endl;
              cout<<"Thank you.\n"<<endl;
              total=totalT
              break;
        }
   }
cout<<"Total number is "<< total <<"\n"<<"."<<endl;
return 0;
}
```

The `break` and `continue` Statements - Example

Output:

Please enter number 1: **-12**

You enter a wrong number

Please enter number 1: **2**

Please enter number 2: **12**

Please enter number 3: **10**

Number is enough.

Thank you.

Total number is **14**.

Counters

- Counter: variable that is incremented or decremented each time a loop repeats
- Can be used to control execution of the loop (loop control variable)
- Must be initialized before entering loop
- May be incremented/decremented either inside the loop or in the loop test

MRSL 1163
SCIENTIFIC COMPUTING FOR SYSTEM ENGINEER

DR. NELIDYA MD. YUSOFF

Razak Faculty of Technology and Informatics

Defining and Calling Functions

- Function definition: A collection of statements that perform a specific task

- Function call: statement causes a function to execute

Writing codes inside main function

```
int main (){
    //Odd number
       -------
       -------
    //Even Number
       -------
       -------
    //MaxMin number
       -------
       -------
    //Even Number
       -------
       -------
    //odd Number
       -------
       return 0;
}
```

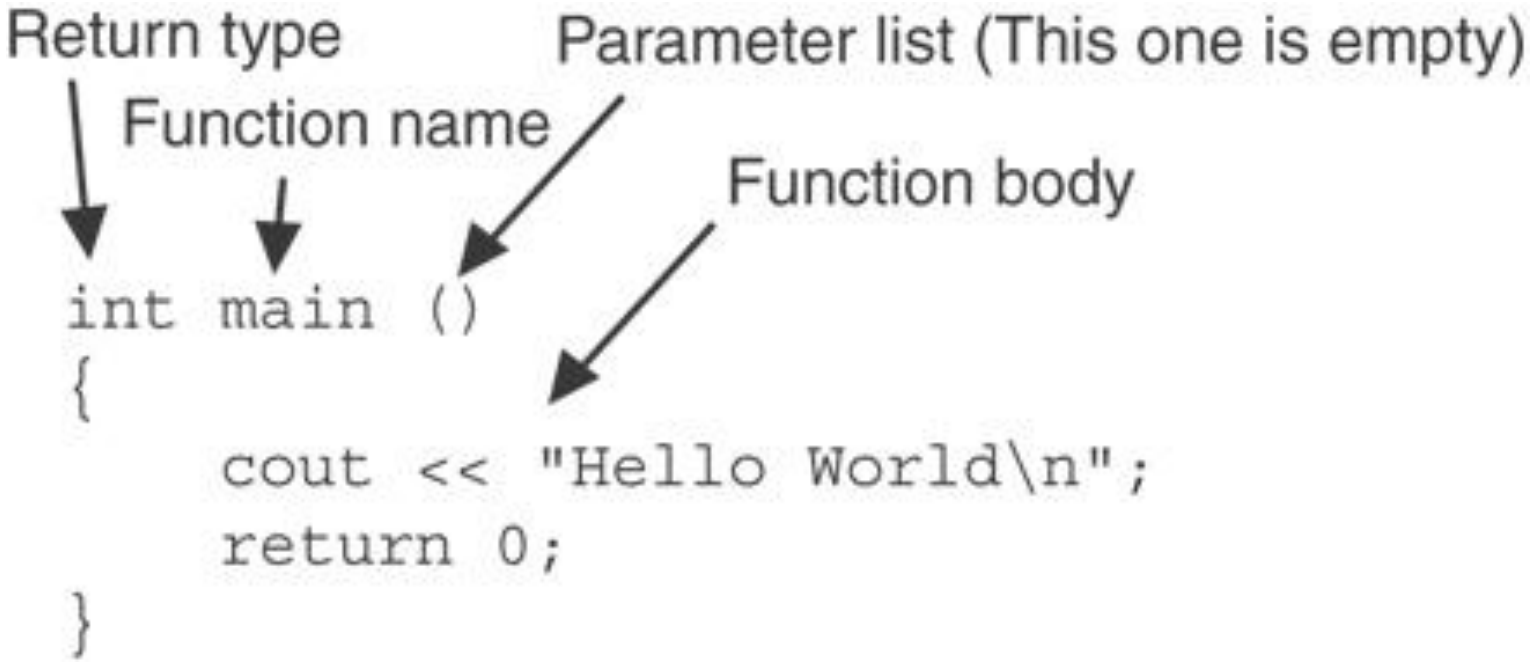
```
int primeNumberFunction(){
    -------
    retun val;
}
int oddEvenFunction(){
    -------
    retun val;
}
string romanNumberFunction(){
    -------
    retun "X";
}
int maxMinFunction(){
    -------
    retun val;
}
```

```
int main(){
  primeNumberFunction();
  oddEvenFunction();
  romanNumberFunction();
  fibanocciNumberFunction();
  ----------------------
  primeNumberFunction();
  maxMinFuction();
  retrun 0;
}
```

Function Definition

- Definition includes:
  - return type: data type of the value that function returns to the part of the program that called it
    - int
  - name: name of the function. Function names follow same rules as variables
    - main
  - parameter list: variables containing values passed to the function
    - ( ) - EMPTY
  - body: statements that perform the function's task, enclosed in { }

Function Definition



Note: The line that reads `int main()` is the *function header*.

Function Return Type

- If a function returns a value, the type of the value must be indicated:

```
void main()
```

- If a function does not return a value, its return type is `void`:

```
void printHeading(int num){
        cout << "My Monthly Sales is :" << num;
  }

int printHeading(){
        cout << "My Monthly Sales not sure :";
        return 1;
 }
```

Calling a Function

- A function is executed when it is called
- Need to define a function before it is called.
- To call a function, use the function name followed by `()` and `;`

```
 printHeading();
myFunction(5, 6);
int myFunction(int x, int y){
 ---
return 1;
}
```

- When called, program executes the body of the called function
- After the function terminates, execution resumes in the calling function at point of call.

Functions in Program 6-1

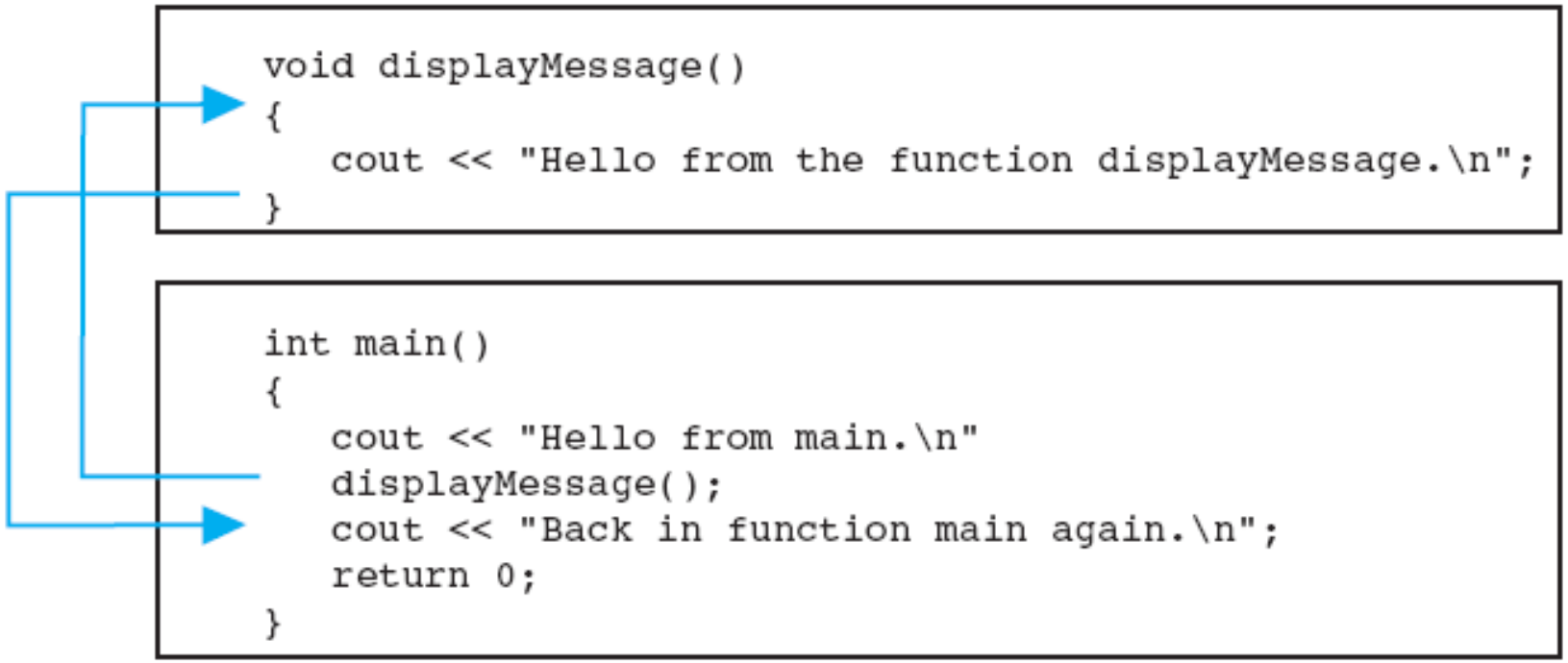
Program 6-1

```
1  // This program has two functions: main and displayMessage
2  #include <iostream>
3  using namespace std;
4
5  //*****************************************
6  // Definition of function displayMessage  *
7  // This function displays a greeting.     *
8  //*****************************************
9
10 void displayMessage()
11 {
12    cout << "Hello from the function displayMessage.\n";
13 }
14
15 //*****************************************
16 // Function main                          *
17 //*****************************************
18
19 int main()
20 {
21    cout << "Hello from main.\n";
22    displayMessage();
23    cout << "Back in function main again.\n";
24    return 0;
25 }
```

Program Output

```
Hello from main.
Hello from the function displayMessage.
Back in function main again.
```

Flow of Control in Program 6-1



```
void displayMessage()
{
   cout << "Hello from the function displayMessage.\n";
}
```

```
int main()
{
   cout << "Hello from main.\n"
   displayMessage();
   cout << "Back in function main again.\n";
   return 0;
}
```

Calling Functions

- `main` can call any number of functions
- Functions can call other functions
- Compiler must know the following about a function before it is called:
  - name
  - return type
  - number of parameters
  - data type of each parameter

Function Prototypes

- Eliminates the need to place a function definition before it is called

```
void first ();
```

- Ways to notify the compiler about a function before a call to the function:

1. Place function definition before calling function's definition

2. Use a function prototype (function declaration) – like the function definition without the body
   - Header: `void printHeading()`
   - Prototype: `void printHeading();`

Program 6-5

```
1  // This program has three functions: main, First, and Second.
2  #include <iostream>
3  using namespace std;
4
5  // Function Prototypes
6  void first();
7  void second();
8
9  int main()
10 {
11    cout << "I am starting in function main.\n";
12    first();    // Call function first
13    second();   // Call function second
14    cout << "Back in function main again.\n";
15    return 0;
16 }
17
```

(Program Continues)

Program 6-5 (Continued)

```
18  //*****************************************
19  // Definition of function first.          *
20  // This function displays a message.      *
21  //*****************************************
22
23  void first()
24  {
25     cout << "I am now inside the function first.\n";
26  }
27
28  //*****************************************
29  // Definition of function second.         *
30  // This function displays a message.      *
31  //*****************************************
32
33  void second()
34  {
35     cout << "I am now inside the function second.\n";
36  }
```

Prototype Notes

- Place prototypes near top of program
- Program must include either prototype or full function definition before any call to the function – compiler error otherwise
- When using prototypes, can place function definitions in any order in source file

Sending Data into a Function

- Can pass values into a function at time of call:
- int a, b, c;
  - cin >> a >> b;

```
c = pow(a, b);
```

- Values passed to function are arguments
- Variables in a function that hold the values passed as arguments are parameters

A Function with a Parameter Variable

```
void displayValue(int num){
    cout << "The value is " << num << endl;
}
void displayValue(){
    cout << "The value is " << num << endl;
}
displayValue(num);
```

The integer variable `num` is a parameter. It accepts any integer value passed to the function.

Program 6-6

```
1  // This program demonstrates a function with a parameter.
2  #include <iostream>
3  using namespace std;
4
5  // Function Prototype
6  void displayValue(int);
7
8  int main()
9  {
10    cout << "I am passing 5 to displayValue.\n";
11    displayValue(5);  // Call displayValue with argument 5
12    cout << "Now I am back in main.\n";
13    return 0;
14 }
15
```

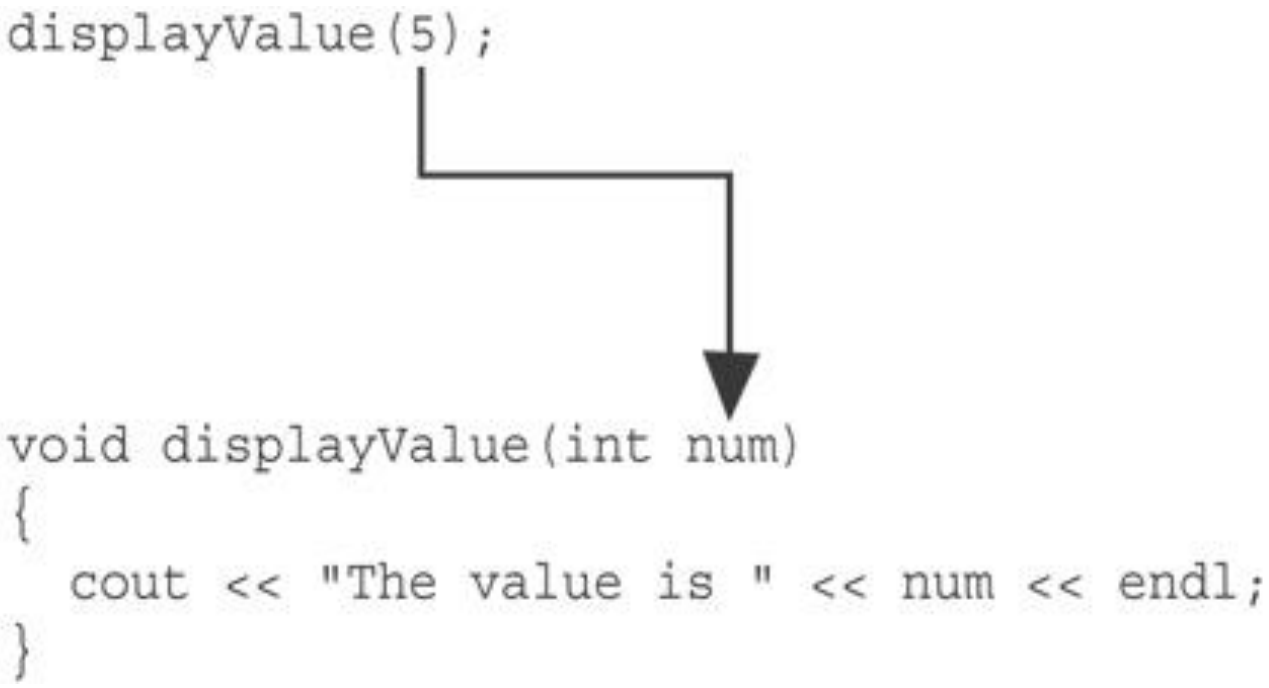
(Program Continues)

Program 6-6 *(continued)*

```
16  //*************************************************
17  // Definition of function displayValue.           *
18  // It uses an integer parameter whose value is displayed. *
19  //*************************************************
20
21  void displayValue(int num)
22  {
23     cout << "The value is " << num << endl;
24  }
```

Program Output

```
I am passing 5 to displayValue.
The value is 5
Now I am back in main.
```



The function call in line 11 passes the value 5 as an argument to the function.

Parameters, Prototypes, and Function Headers

- For each function argument,
  - the prototype must include the data type of each parameter inside its parentheses
  - the header must include a declaration for each parameter in its ()

```
void evenOrOdd(int);   //prototype
void evenOrOdd(int num) //header
evenOrOdd(val);         //call
```

Passing Multiple Arguments

When calling a function and passing multiple arguments:

- the number of arguments in the call must match the prototype and definition
- the first argument will be used to initialize the first parameter, the second argument to initialize the second parameter, etc.

Program 6-8

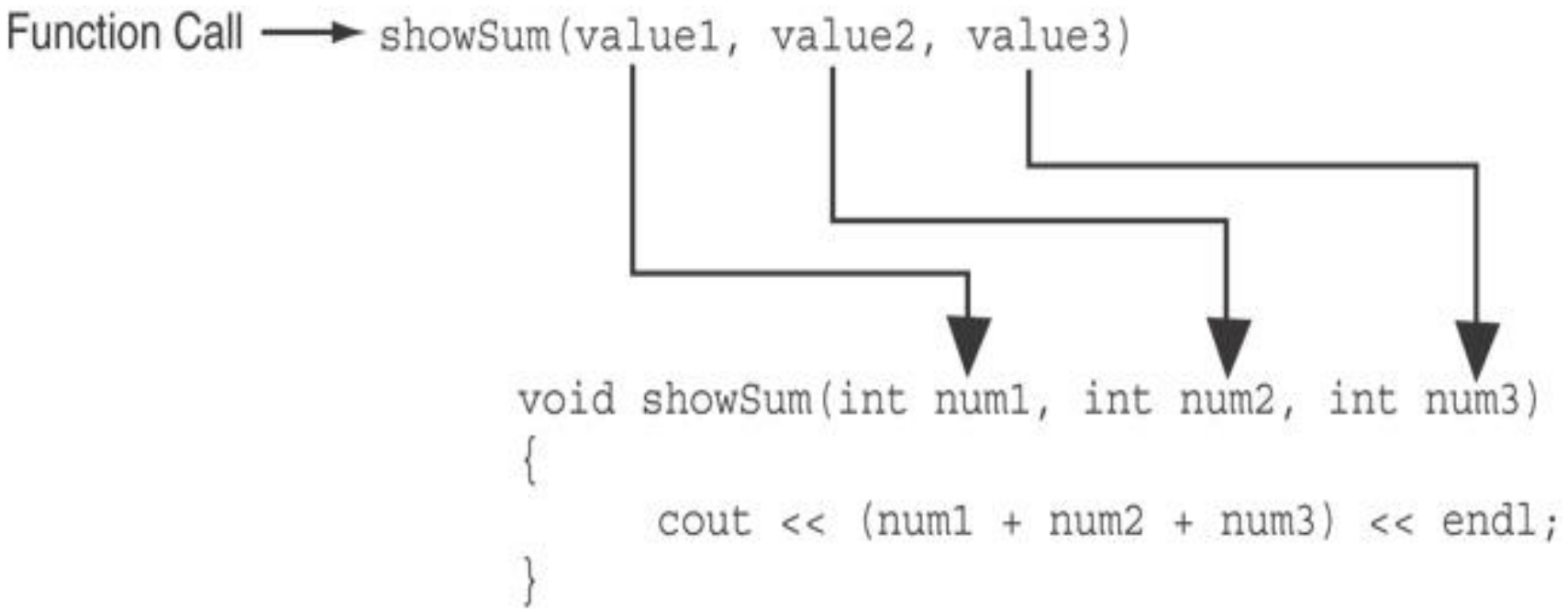
```
1  // This program demonstrates a function with three parameters.
2  #include <iostream>
3  using namespace std;
4
5  // Function Prototype
6  void showSum(int, int, int);
7
8  int main()
9  {
10    int value1, value2, value3;
11
12    // Get three integers.
13    cout << "Enter three integers and I will display ";
14    cout << "their sum: ";
15    cin >> value1 >> value2 >> value3;
16
17    // Call showSum passing three arguments.
18    showSum(value1, value2, value3);
19    return 0;
20 }
21
```

(Program Continues)

```
22  //*****************************************************
23  // Definition of function showSum.                    *
24  // It uses three integer parameters. Their sum is displayed. *
25  //*****************************************************
26
27  void showSum(int num1, int num2, int num3)
28  {
29     cout << (num1 + num2 + num3) << endl;
30  }
```

Program Output with Example Input Shown in Bold

```
Enter three integers and I will display their sum: 4 8 7 [Enter]
19
```



The function call in line 18 passes value1, value2, and value3 as a arguments to the function.

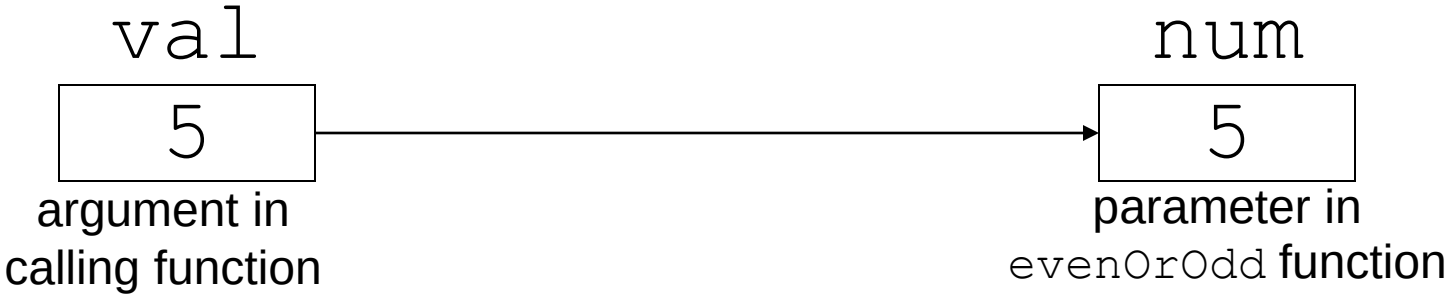
Passing Data by Value

- Pass by value: when an argument is passed to a function, its value is copied into the parameter.

- Changes to the parameter in the function do not affect the value of the argument

Passing Information to Parameters by Value

- **Example:** `int val=5;`
  `evenOrOdd(val);`



- `evenOrOdd` can change variable `num`, but it will have no effect on variable `val`

The `return` Statement

- Used to end execution of a function
- Can be placed anywhere in a function
  - Statements that follow the `return` statement will not be executed
- Can be used to prevent abnormal termination of program
- In a `void` function without a `return` statement, the function ends at its last `}`

Program 6-11

```
1  // This program uses a function to perform division. If division
2  // by zero is detected, the function returns.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype.
7  void divide(double, double);
8
9  int main()
10 {
11    double num1, num2;
12
13    cout << "Enter two numbers and I will divide the first\n";
14    cout << "number by the second number: ";
15    cin >> num1 >> num2;
16    divide(num1, num2);
17    return 0;
18 }
```

(Program Continues)

Program 6-11(Continued)

```
20  //*************************************************************
21  // Definition of function divide.                             *
22  // Uses two parameters: arg1 and arg2. The function divides arg1*
23  // by arg2 and shows the result. If arg2 is zero, however, the  *
24  // function returns.                                          *
25  //*************************************************************
26
27  void divide(double arg1, double arg2)
28  {
29     if (arg2 == 0.0)
30     {
31        cout << "Sorry, I cannot divide by zero.\n";
32        return;
33     }
34     cout << "The quotient is " << (arg1 / arg2) << endl;
35  }
```

Program Output with Example Input Shown in Bold

```
Enter two numbers and I will divide the first
number by the second number: 12 0 [Enter]
Sorry, I cannot divide by zero.
```

Returning a Value From a Function

- A function can return a value back to the statement that called the function.
- You've already seen the `pow` function, which returns a value:

```
double x;
x = pow(2.0, 10.0);
```

Returning a Value From a Function

- In a value-returning function, the `return` statement can be used to return a value from function to the point of call. Example:

```
int sum(int num1, int num2)
{
   double result;
   result = num1 + num2;
   return result;
}
```

A Value-Returning Function

Return Type

```
int sum(int num1, int num2)
{
    double result;
    result = num1 + num2;
    return result;
}
```

Value Being Returned

A Value-Returning Function

```
int sum(int num1, int num2)
{
    return num1 + num2;
}
```

Functions can return the values of expressions, such as `num1 + num2`

Program 6-12

```
1  // This program uses a function that returns a value.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype
6  int sum(int, int);
7
8  int main()
9  {
10    int value1 = 20,   // The first value
11        value2 = 40,   // The second value
12        total;         // To hold the total
13
14    // Call the sum function, passing the contents of
15    // value1 and value2 as arguments. Assign the return
16    // value to the total variable.
17    total = sum(value1, value2);
18
19    // Display the sum of the values.
20    cout << "The sum of " << value1 << " and "
21         << value2 << " is " << total << endl;
22    return 0;
23 }
```

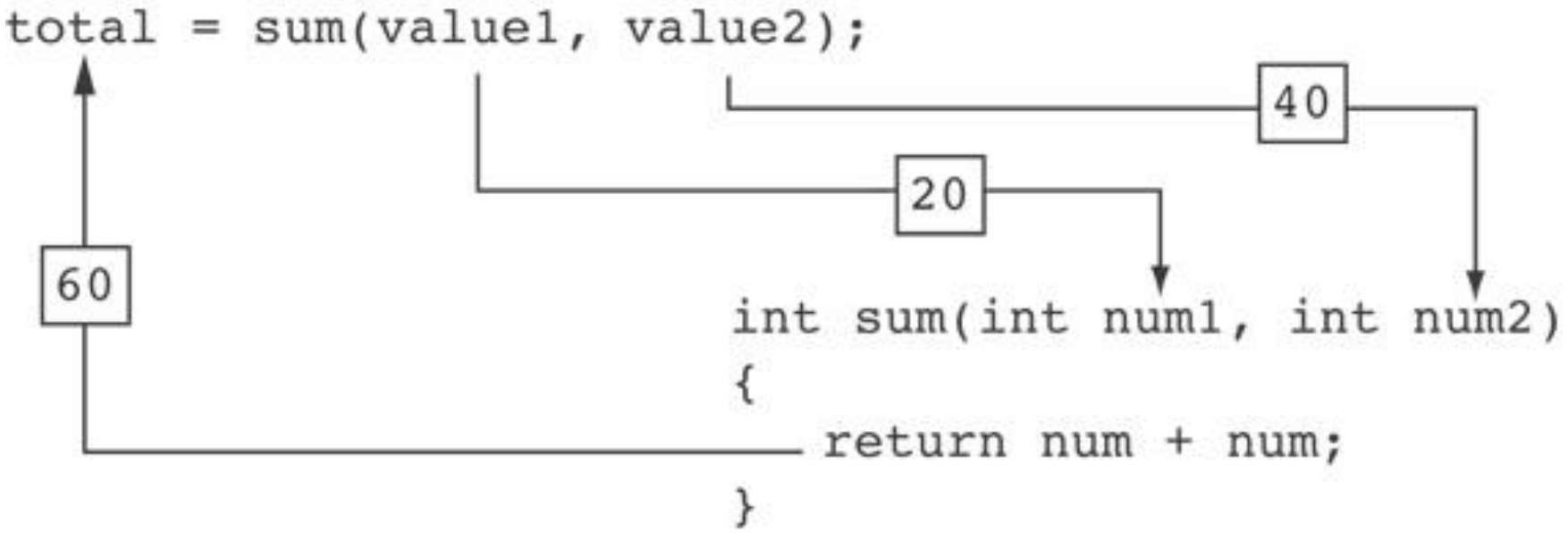
(Program Continues)

Program 6-12 (Continued)

```
24
25  //*************************************************
26  // Definition of function sum. This function returns  *
27  // the sum of its two parameters.                     *
28  //*************************************************
29
30  int sum(int num1, int num2)
31  {
32     return num1 + num2;
33  }
```

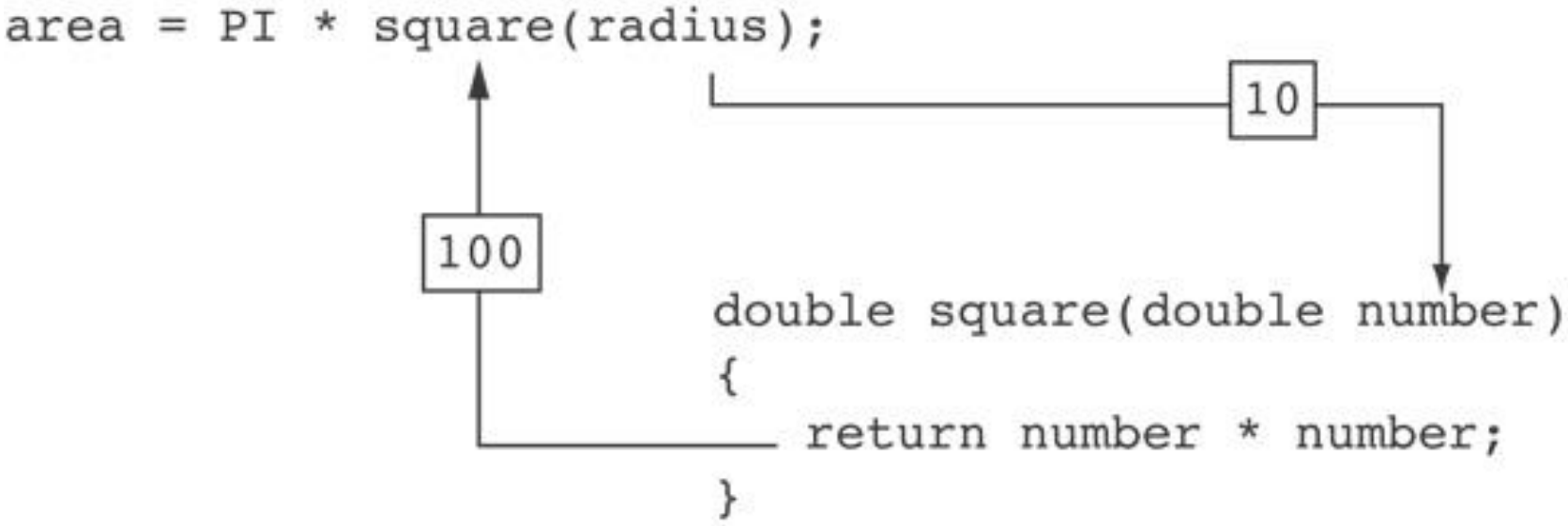
Program Output

```
The sum of 20 and 40 is 60
```



The statement in line 17 calls the sum function, passing `value1` and `value2` as arguments. The return value is assigned to the `total` variable.

Another Example, from Program 6-13



Returning a Value From a Function

- The prototype and the definition must indicate the data type of return value (not `void`)

- Calling function should use return value:
  - assign it to a variable
  - send it to `cout`
  - use it in an expression

Returning a Boolean Value

- Function can return `true` or `false`
- Declare return type in function prototype and heading as `bool`
- Function body must contain `return` statement(s) that return `true` or `false`
- Calling function can use return value in a relational expression

Program 6-15

```
1  // This program uses a function that returns true or false.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype
6  bool isEven(int);
7
8  int main()
9  {
10    int val;
11
12    // Get a number from the user.
13    cout << "Enter an integer and I will tell you ";
14    cout << "if it is even or odd: ";
15    cin >> val;
16
17    // Indicate whether it is even or odd.
18    if (isEven(val))
19       cout << val << " is even.\n";
20    else
21       cout << val << " is odd.\n";
22    return 0;
23 }
24
```

(Program Continues)

```
25  //*************************************************************
26  // Definition of function isEven. This function accepts an    *
27  // integer argument and tests it to be even or odd. The function  *
28  // returns true if the argument is even or false if the argument  *
29  // is odd. The return value is a bool.                        *
30  //*************************************************************
31
32  bool isEven(int number)
33  {
34     bool status;
35
36     if (number % 2 == 0)
37        status = true;  // The number is even if there is no remainder.
38     else
39        status = false; // Otherwise, the number is odd.
40     return status;
41  }
```

Program Output with Example Input Shown in Bold

```
Enter an integer and I will tell you if it is even or odd: 5 [Enter]
5 is odd.
```

Local and Global Variables

- Variables defined inside a function are *local* to that function. They are hidden from the statements in other functions, which normally cannot access them.
- Because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.

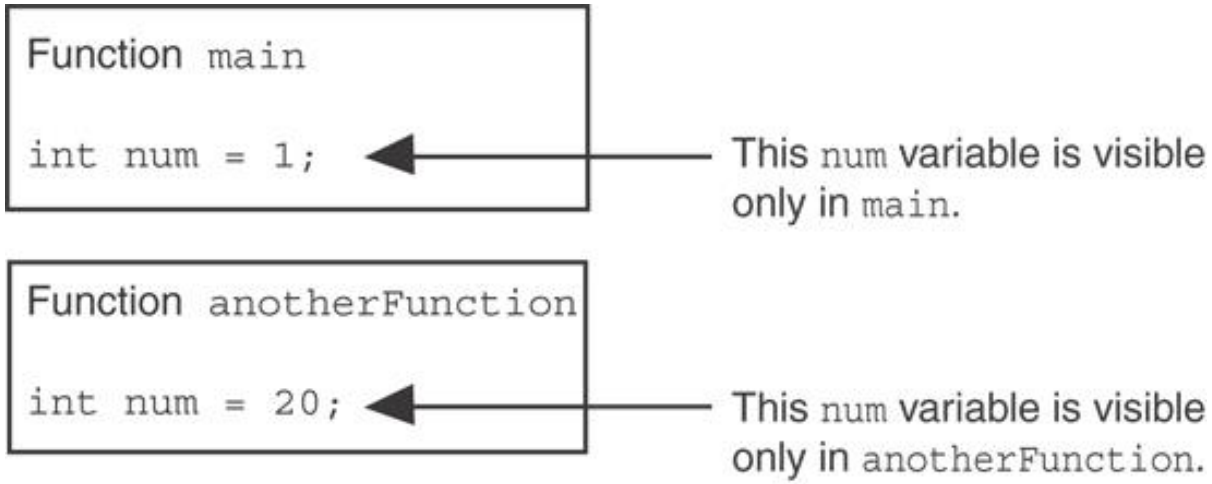
Program 6-16

```
1  // This program shows that variables defined in a function
2  // are hidden from other functions.
3  #include <iostream>
4  using namespace std;
5
6  void anotherFunction(); // Function prototype
7
8  int main()
9  {
10    int num = 1;   // Local variable
11
12    cout << "In main, num is " << num << endl;
13    anotherFunction();
14    cout << "Back in main, num is " << num << endl;
15    return 0;
16 }
17
18 //*************************************************
19 // Definition of anotherFunction                  *
20 // It has a local variable, num, whose initial value  *
21 // is displayed.                                  *
22 //*************************************************
23
24 void anotherFunction()
25 {
26    int num = 20;  // Local variable
27
28    cout << "In anotherFunction, num is " << num << endl;
29 }
```

Program Output

```
In main, num is 1
In anotherFunction, num is 20
Back in main, num is 1
```

When the program is executing in `main`, the `num` variable defined in `main` is visible. When `anotherFunction` is called, however, only variables defined inside it are visible, so the `num` variable in `main` is hidden.



Local Variable Lifetime

- A function's local variables exist only while the function is executing. This is known as the *lifetime* of a local variable.

- When the function begins, its local variables and its parameter variables are created in memory, and when the function ends, the local variables and parameter variables are destroyed.

- This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.

Global Variables and Global Constants

- A global variable is any variable defined outside all the functions in a program.
- The scope of a global variable is the portion of the program from the variable definition to the end.
- This means that a global variable can be accessed by *all* functions that are defined after the global variable is defined.
- You should avoid using global variables because they make programs difficult to

Program 6-19

```
1  // This program calculates gross pay.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  // Global constants
7  const double PAY_RATE = 22.55;     // Hourly pay rate
8  const double BASE_HOURS = 40.0;    // Max non-overtime hours
9  const double OT_MULTIPLIER = 1.5; // Overtime multiplier
10
11 // Function prototypes
12 double getBasePay(double);
13 double getOvertimePay(double);
14
15 int main()
16 {
17    double hours,              // Hours worked
18           basePay,            // Base pay
19           overtime = 0.0,     // Overtime pay
20           totalPay;           // Total pay
```

Global constants defined for values that do not change throughout the program's execution.

The constants are then used for those values throughout the program.

```
29      // Get overtime pay, if any.
30      if (hours > BASE_HOURS)
31         overtime = getOvertimePay(hours);

56      // Determine base pay.
57      if (hoursWorked > BASE_HOURS)
58         basePay = BASE_HOURS * PAY_RATE;
59      else
60         basePay = hoursWorked * PAY_RATE;

75      // Determine overtime pay.
76      if (hoursWorked > BASE_HOURS)
77      {
78         overtimePay = (hoursWorked - BASE_HOURS) *
79                       PAY_RATE * OT_MULTIPLIER;
```

Initializing Local and Global Variables

- Local variables are not automatically initialized. They must be initialized by programmer.

- Global variables (not constants) are automatically initialized to `0` (numeric) or `NULL` (character) when the variable is defined.

Static Local Variables

- Local variables only exist while the function is executing. When the function terminates, the contents of local variables are lost.

- `static` local variables retain their contents between function calls.

- `static` local variables are defined and initialized only the first time the function is executed. `0` is the default initialization value.

Program 6-21

```
1  // This program shows that local variables do not retain
2  // their values between function calls.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype
7  void showLocal();
8
9  int main()
10 {
11    showLocal();
12    showLocal();
13    return 0;
14 }
15
```

(Program Continues)

Program 6-21 *(continued)*

```
16  //*********************************************************
17  // Definition of function showLocal.                      *
18  // The initial value of localNum, which is 5, is displayed. *
19  // The value of localNum is then changed to 99 before the   *
20  // function returns.                                      *
21  //*********************************************************
22
23  void showLocal()
24  {
25     int localNum = 5; // Local variable
26
27     cout << "localNum is " << localNum << endl;
28     localNum = 99;
29  }
```

Program Output

```
localNum is 5
localNum is 5
```

In this program, each time `showLocal` is called, the `localNum` variable is re-created and initialized with the value 5.

A Different Approach, Using a Static Variable

Program 6-22

```
1  // This program uses a static local variable.
2  #include <iostream>
3  using namespace std;
4
5  void showStatic(); // Function prototype
6
7  int main()
8  {
9     // Call the showStatic function five times.
10    for (int count = 0; count < 5; count++)
11       showStatic();
12    return 0;
13 }
14
```

(Program Continues)

Program 6-22 *(continued)*

```
15  //*************************************************************
16  // Definition of function showStatic.                         *
17  // statNum is a static local variable. Its value is displayed *
18  // and then incremented just before the function returns.     *
19  //*************************************************************
20
21  void showStatic()
22  {
23     static int statNum;
24
25     cout << "statNum is " << statNum << endl;
26     statNum++;
27  }
```

Program Output

```
statNum is 0
statNum is 1
statNum is 2
statNum is 3
statNum is 4
```

`statNum` is automatically initialized to 0. Notice that it retains its value between function calls.

If you do initialize a local static variable, the initialization only happens once. See Program 6-23.

```
16  //*********************************************************
17  // Definition of function showStatic.                     *
18  // statNum is a static local variable. Its value is displayed *
19  // and then incremented just before the function returns.  *
20  //*********************************************************
21
22  void showStatic()
23  {
24     static int statNum = 5;
25
26     cout << "statNum is " << statNum << endl;
27     statNum++;
28  }
```

Program Output

```
statNum is 5
statNum is 6
statNum is 7
statNum is 8
statNum is 9
```

Default Arguments

A Default argument is an argument that is passed automatically to a parameter if the argument is missing on the function call.

- Must be a constant declared in prototype:

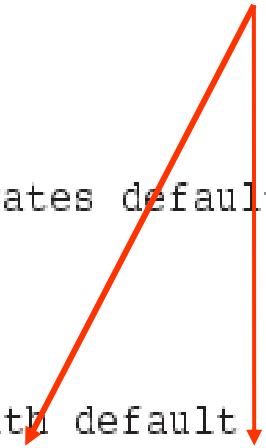
```
void evenOrOdd(int = 0);
```

- Can be declared in header if no prototype

- Multi-parameter functions may have default arguments for some or all of them:

```
int getSum(int, int=0, int=0);
```

Default arguments specified in the prototype



Program 6-24

```
1  // This program demonstrates default function arguments.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype with default arguments
6  void displayStars(int = 10, int = 1);
7
8  int main()
9  {
10    displayStars();      // Use default values for cols and rows.
11    cout << endl;
12    displayStars(5);     // Use default value for rows.
13    cout << endl;
14    displayStars(7, 3);  // Use 7 for cols and 3 for rows.
15    return 0;
16 }
```

(Program Continues)

Program 6-23 (Continued)

```
18  //*********************************************************
19  // Definition of function displayStars.                   *
20  // The default argument for cols is 10 and for rows is 1.*
21  // This function displays a square made of asterisks.     *
22  //*********************************************************
23
24  void displayStars(int cols, int rows)
25  {
26     // Nested loop. The outer loop controls the rows
27     // and the inner loop controls the columns.
28     for (int down = 0; down < rows; down++)
29     {
30        for (int across = 0; across < cols; across++)
31           cout << "*";
32        cout << endl;
33     }
34  }
```

Program Output

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

*****

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

Using Reference Variables as Parameters

- A mechanism that allows a function to work with the original argument from the function call, not a copy of the argument
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to ‘return’ more than one value

Passing by Reference

- A reference variable is an alias for another variable
- Defined with an ampersand (`&`)

```
void getDimensions(int&, int&);
```

- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters *by reference*

Program 6-25

The & here in the prototype indicates that the parameter is a reference variable.

```
1  // This program uses a reference variable as a function
2  // parameter.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype. The parameter is a reference variable.
7  void doubleNum(int &);
8
9  int main()
10 {
11    int value = 4;
12
13    cout << "In main, value is " << value << endl;
14    cout << "Now calling doubleNum..." << endl;
15    doubleNum(value);
16    cout << "Now back in main. value is " << value << endl;
17    return 0;
18 }
19
```

Here we are passing value by reference.

(Program Continues)

Program 6-25 (Continued)

```
20  //*************************************************************
21  // Definition of doubleNum.                                   *
22  // The parameter refVar is a reference variable. The value    *
23  // in refVar is doubled.                                      *
24  //*************************************************************
25
26  void doubleNum (int &refVar)
27  {
28     refVar *= 2;
29  }
```

The & also appears here in the function header.

Program Output

```
In main, value is 4
Now calling doubleNum...
Now back in main. value is 8
```

Reference Variable Notes

- Each reference parameter must contain `&`
- Space between type and `&` is unimportant
- Must use `&` in both prototype and header
- Argument passed to reference parameter must be a variable – cannot be an expression or constant
- Use when appropriate – don't use when argument should not be changed by function, or if function needs to return only 1 value

Overloading Functions

- Overloaded functions have the same name but different parameter lists
- Can be used to create functions that perform the same task but take different parameter types or different number of parameters
- Compiler will determine which version of function to call by argument and parameter lists

Function Overloading Examples

Using these overloaded functions,

```
void getDimensions(int);              // 1
void getDimensions(int, int);         // 2
void getDimensions(int, double);      // 3
void getDimensions(double, double);// 4
```

the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);              // 1
getDimensions(length, width);       // 2
getDimensions(length, height);      // 3
getDimensions(height, base);        // 4
```

Program 6-27

```
1  // This program uses overloaded functions.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  // Function prototypes
7  int square(int);
8  double square(double);
9
10 int main()
11 {
12    int userInt;
13    double userFloat;
14
15    // Get an int and a double.
16    cout << fixed << showpoint << setprecision(2);
17    cout << "Enter an integer and a floating-point value: ";
18    cin >> userInt >> userFloat;
19
20    // Display their squares.
21    cout << "Here are their squares: ";
22    cout << square(userInt) << " and " << square(userFloat);
23    return 0;
24 }
```

The overloaded functions have different parameter lists

Passing a double

Passing an int

(Program Continues)

Program 6-27 (Continued)

```
26  //*************************************************************
27  // Definition of overloaded function square.                  *
28  // This function uses an int parameter, number. It returns the *
29  // square of number as an int.                                *
30  //*************************************************************
31
32  int square(int number)
33  {
34     return number * number;
35  }
36
37  //*************************************************************
38  // Definition of overloaded function square.                  *
39  // This function uses a double parameter, number. It returns  *
40  // the square of number as a double.                          *
41  //*************************************************************
42
43  double square(double number)
44  {
45     return number * number;
46  }
```

Program Output with Example Input Shown in Bold

```
Enter an integer and a floating-point value: 12 4.2 [Enter]
Here are their squares: 144 and 17.64
```

The `exit()` Function

- Terminates the execution of a program
- Can be called from any function
- Can pass an `int` value to operating system to indicate status of program termination
- Usually used for abnormal termination of program
- Requires `cstdlib` header file

The `exit()` Function

- Example:

```
exit(0);
```

- The `cstdlib` header defines two constants that are commonly passed, to indicate success or failure:

```
exit(EXIT_SUCCESS);
exit(EXIT_FAILURE);
```

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

MRSL 1163
SCIENTIFIC COMPUTING FOR SYSTEM ENGINEER

Lecture 6 (Arrays)

Dr Nelidya Md Yusoff
Razak Faculty of Technology and Informatics

ARRAY

- A form of variable from usual data type such as `int, double` and `char`
- Able to store a group of data of the same data type under one name.
- Example (in 1D-ARRAY) :

  for variable `mark[100]`:
  - 100 memory cell are provided
- Array allows to store and work with multiple values of the same data type
- Values are stored in adjacent memory locations

- Array
  - Consecutive group of memory locations
  - Same name and type (**int**, **char**, etc.)
- To refer to an element
  - Specify array name and position number (index)
  - *Format: arrayname [position number]*
  - First element at position 0
- N-element array c

  c[0], c[1] … c[n - 1]
  - Nth element as position N-1

Introduction to Arrays

- **Array:** **variable** that can **store multiple values or a group of the same data type under one name**.
- Usual data type such as `int`, `double` and `char`.
- Values are stored in adjacent memory locations.
- Declared using `[ ]` operator:

```
int tests[5];
```

Introduction to Arrays

- Variable declaration so far can hold only one value at a time

```
int val = 5;
```
// enough memory for 1 int (4 Bytes)

```
char letter = 'A';
```
// enough memory for 1 char (1 Byte)

```
Double price = 56.981;
```
// enough memory for 1 double (8 Bytes)

Array Terminology

In the definition `int tests[5];`

- `int` is the **data type** of the array elements
- `tests` is the **name** of the array
- `5`, in `[5]`, is the **size declarator**. It shows the number of elements in the array.
- The **size** of an array is (number of elements) * (size of each element)

Array Terminology

- The **size** of an array is:
  - the total number of bytes allocated for it
  - (number of elements) * (number of bytes for each element)
- Examples:

  `int tests[5]` is an array of 20 bytes, assuming 4 bytes for an `int`

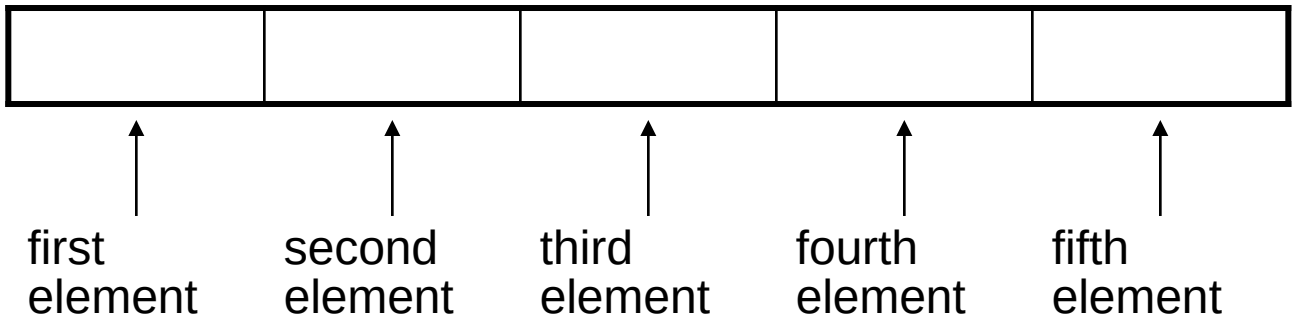
  `long double measures[10]` is an array of 80 bytes, assuming 8 bytes for a `long double`

Array Definition

- The definition:

```
int tests[5]; // size is 5
```

allocates the following memory:



- `tests` - name of the array, read as tests sub 5
- `5` - the number of elements or values that tests can hold

Size Declarators

- Named constants are commonly used as size declarators.

```
const int SIZE = 5;
int tests[SIZE];
```

- This eases program maintenance when the size of the array needs to be changed.

Array Declaration

- **Syntax:**

```
DataType variable1 [n1],
variable2 [n2];
```

- n1 and n2 are called **size declaration** for array.
- **Example:**

```
int mark[100];
long stress[15];
float quiz[60];
```

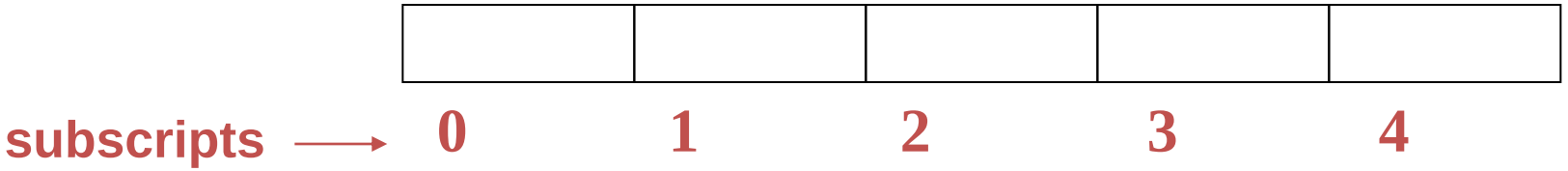
Array Declaration

	element	subscript
mark[0]	98	0
mark[1]	76	1
mark[2]	87	2
...	...	.
...	...	.
...	...	.
mark[99]	68	99

- With this declaration, in RAM
- In C++,
  - element starts with 0 subscript
  - Ends with n-i subscript where n is the size declaration.
- In n=100 in mark[100], then we have
- mark[0],mark[1],...,mark[99]

Accessing Array Elements

- An array has only one name, the elements may be accessed and used as individual variables.
- Each array element has a unique subscript, used to access the element.
- Subscripts start at 0
- The last element's subscript is *n*-1 where *n* is the number of elements in the array.



Accessing Array Elements

- Array elements can be used as regular variables:

```
tests[0] = 79;
cout << tests[0];
cin >> tests[1];
tests[4] = tests[0] + tests[1];
```

- Arrays must be accessed by array name and subscript:

```
cout << tests; // illegal due to
missing subscript
```

Array Subscripts

- Array subscript can be
  - integer **constant**,
  - integer **variable**, or
  - integer **expression**
- Examples:

	Subscript is
`cin >> tests[3];`	int constant
`cout << tests[i];`	int variable
`cout << tests[i+j];`	int expression

Inputting and Displaying Array Contents

- `cout` and `cin` can be used to **display values** from and **store values** into an array

```
const int ISIZE = 5;

int tests[ISIZE]; // Define array with 5
cout << "Enter first test score ";
cin  >>  tests[0];// NOT cin >> tests
cout << tests[0];// NOT cout << tests
```

Program 7-1

```
1  // This program asks for the number of hours worked
2  // by six employees. It stores the values in an array.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int NUM_EMPLOYEES = 6;
9     int hours[NUM_EMPLOYEES];
10
11    // Get the hours worked by each employee.
12    cout << "Enter the hours worked by "
13         << NUM_EMPLOYEES << " employees: ";
14    cin >> hours[0];
15    cin >> hours[1];
16    cin >> hours[2];
17    cin >> hours[3];
18    cin >> hours[4];
19    cin >> hours[5];
20
```

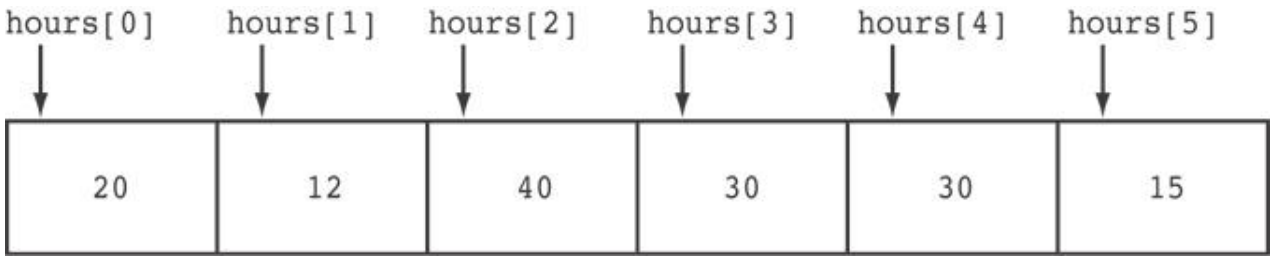
(Program Continues)

```
21    // Display the values in the array.
22    cout << "The hours you entered are:";
23    cout << " " << hours[0];
24    cout << " " << hours[1];
25    cout << " " << hours[2];
26    cout << " " << hours[3];
27    cout << " " << hours[4];
28    cout << " " << hours[5] << endl;
29    return 0;
30 }
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked by 6 employees: 20 12 40 30 30 15 [Enter]
The hours you entered are: 20 12 40 30 30 15
```

Here are the contents of the `hours` array, with the values entered by the user in the example output:



Inputting and Displaying Array Contents using Loops

- To access each element of an array
  - Use a loop
  - Let the loop control variable be the array subscript
  - A different array element will be referenced each time through the loop

```
for (i = 0; i < 5; i++){
   cout << tests[i] << endl;
}
```

Exercise 1 - Display the inputs using `for` loop

Program 7-1

```
1  // This program asks for the number of hours worked
2  // by six employees. It stores the values in an array.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int NUM_EMPLOYEES = 6;
9     int hours[NUM_EMPLOYEES];
10
11    // Get the hours worked by each employee.
12    cout << "Enter the hours worked by "
13         << NUM_EMPLOYEES << " employees: ";
14    cin >> hours[0];
15    cin >> hours[1];
16    cin >> hours[2];
17    cin >> hours[3];
18    cin >> hours[4];
19    cin >> hours[5];
20
```

(Program Continues)

```
21     // Display the values in the array.
22     cout << "The hours you entered are:";
23     cout << " " << hours[0];
24     cout << " " << hours[1];
25     cout << " " << hours[2];
26     cout << " " << hours[3];
27     cout << " " << hours[4];
28     cout << " " << hours[5] << endl;
29     return 0;
30 }
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked by 6 employees: 20 12 40 30 30 15 [Enter]
The hours you entered are: 20 12 40 30 30 15
```

Exercise 1 - Answer

```
6   int main()
7   {
8        const int NUM_EMPLOYEES = 6; // Number of employees
9        int hours[NUM_EMPLOYEES];    // Each employee's hours
10       int count;                   // Loop counter
11
12       // Input the hours worked.
13       for (count = 0; count < NUM_EMPLOYEES; count++)
14       {
15            cout << "Enter the hours worked by employee "
16                 << (count + 1) << ": ";
17            cin >> hours[count];
18       }
19
20       // Display the contents of the array.
21       cout << "The hours you entered are:";
22       for (count = 0; count < NUM_EMPLOYEES; count++)
23          cout << " " << hours[count];
24       cout << endl;
25       return 0;
26  }
```

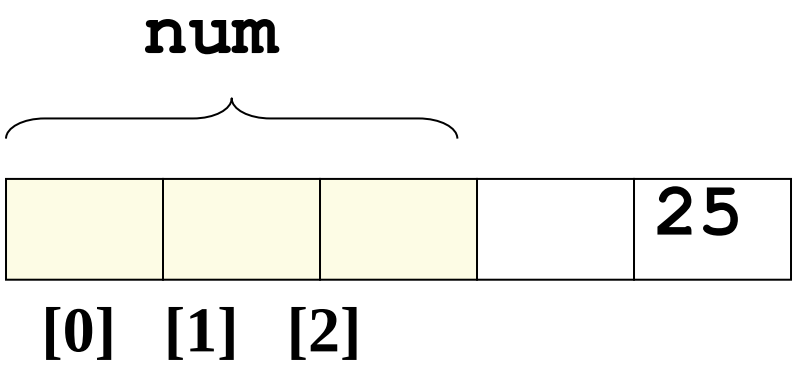
Program Output with Example Input Shown in Bold

```
Enter the hours worked by employee 1: 20 [Enter]
Enter the hours worked by employee 2: 12 [Enter]
Enter the hours worked by employee 3: 40 [Enter]
Enter the hours worked by employee 4: 30 [Enter]
Enter the hours worked by employee 5: 30 [Enter]
Enter the hours worked by employee 6: 15 [Enter]
The hours you entered are: 20 12 40 30 30 15
```

No Bounds Checking

- There are no checks in C++ that an array subscript is in range
- An invalid array subscript can cause program to overwrite other memory
- Example:

```
const int ISIZE = 3;
int i = 4;
int num[ISIZE];
num[i] = 25;
```

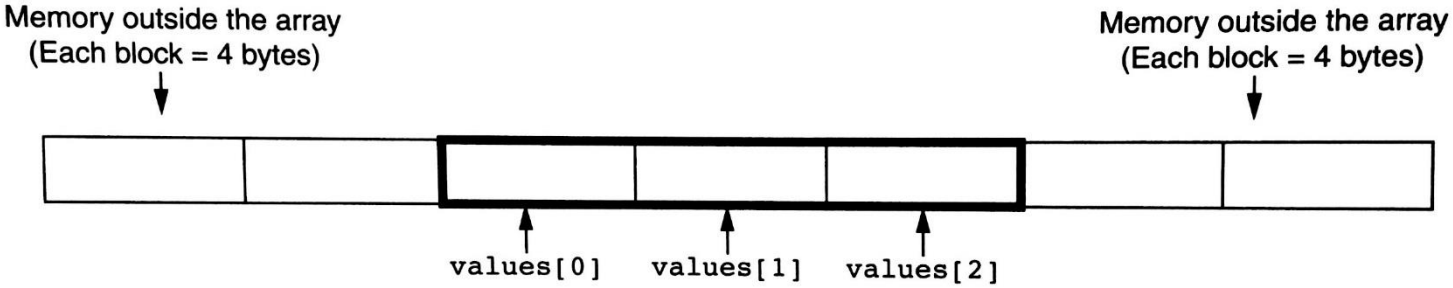


Program 7-5

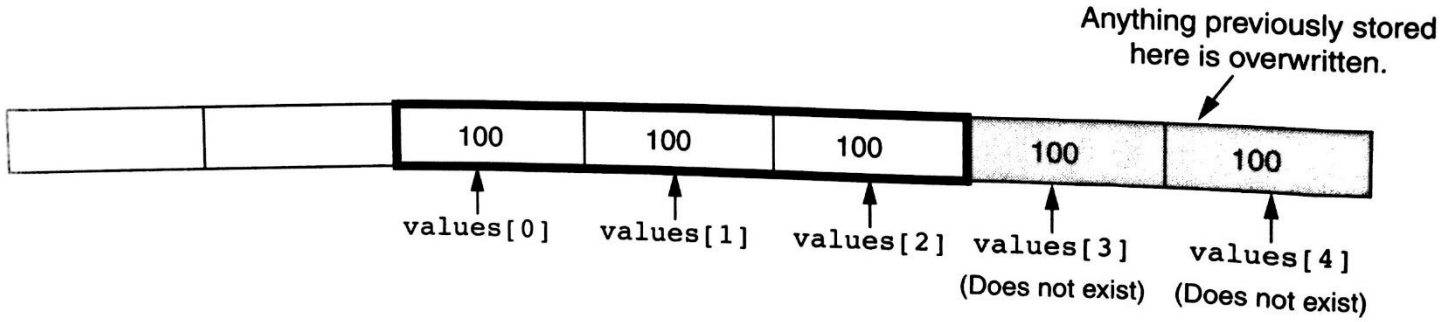
```
1  // This program unsafely accesses an area of memory by writing
2  // values beyond an array's boundary.
3  // WARNING: If you compile and run this program, it could crash.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9       const int SIZE = 3;   // Constant for the array size
10      int values[SIZE];     // An array of 3 integers
11      int count;            // Loop counter variable
12
13      // Attempt to store five numbers in the three-element array.
14      cout << "I will store 5 numbers in a 3-element array!\n";
15      for (count = 0; count < 5; count++)
16           values[count] = 100;
17
18      // If the program is still running, display the numbers.
19      cout << "If you see this message, it means the program\n";
20      cout << "has not crashed! Here are the numbers:\n";
21      for (count = 0; count < 5; count++)
22           cout << values[count] << endl;
23      return 0;
24 }
```

Figure 7-9

The way the `values` array is set up in memory.
The outlined area represents the array.



How the numbers assigned to the array overflow the array's boundaries.
The shaded area is the section of memory illegally written to.



Off-By-One Errors

- An off-by-one error happens when you use array subscripts that are off by one.
- This can happen when you **start the initialization at 1** rather than 0.

```
// This code has an off-by-one error.
const int SIZE = 100;
int numbers[SIZE];
for (int count = 1; count <= SIZE;
count++){
   numbers[count] = 0;
}
```

Off-By-One Errors

- As a result the first element which is at subscript 0 is skipped.
- Loop attempts to use 100 as a subscript during the last iteration
- 100 is invalid subscript, the program will write data beyond the array's boundaries.

Array Initialization

- Can be initialized during program execution with assignment statements

```
tests[0] = 79;
tests[1] = 82; // etc.
```

- Can be initialized at array definition with an initialization list

```
const int ISIZE = 5;
int tests[ISIZE]={79,82,91,77,84};
```

Program 7-6

```
7     const int MONTHS = 12;
8     int days[MONTHS] = { 31, 28, 31, 30,
9                          31, 30, 31, 31,
10                         30, 31, 30, 31};
11
12    for (int count = 0; count < MONTHS; count++)
13    {
14       cout << "Month " << (count + 1) << " has ";
15       cout << days[count] << " days.\n";
16    }
```

Program Output

```
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

Partial Array Initialization

- If array is initialized at definition with fewer values than the size declarator of the array, remaining elements will be set to `0` or `NULL`

```
int tests[ISIZE] = {79, 82};
```

79	82	0	0	0

Implicit Array Sizing

- Can determine array size by the size of the initialization list

```
short quizzes[]={12,17,15,11};
```

12	17	15	11

- Must use either array size declarator or initialization list when array is defined

Exercise 2

- Write a program that defines array of 5 elements.
  - Display max of the elements.
  - Display min of the elements.

Exercise 2 - Answer

Array-Ex-2.cpp

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6   int num[]={5, 13, 0, 1, 7};
7   int numMax=0, numMin=0;
8   int valMax=num[0], valMin=num[0];
9
10  for(int i=0;i<5;i++)
11  {
12      if(num[i]>valMax)
13      {
14          numMax=i;
15          valMax=num[i];
16      }
17
18      else if(num[i]<valMin)
19      {
20          numMin=i;
21          valMin=num[i];
22      }
23  }
24
25  cout<<"Minimum element is from num["<<numMin<<"]   = "<<valMin<<endl;
26  cout<<"Maximum element is from num["<<numMax<<"]   = "<<valMax;
27
28      return 0;
29  }
```

Output:

```
Minimum element is from num[2]   = 0
Maximum element is from num[1]   = 13
```

Processing Array Contents

- Array elements can be
  - treated as ordinary variables of the same type as the array
  - used in arithmetic operations, in relational expressions, etc.
- Example:

```
if (principalAmt[3] >= 10000){
   interest = principalAmt[3] * intRate1;
}
else{
   interest = principalAmt[3] * intRate2;
}
```

Processing Array Contents

- Comparison with working with variables :

```
// principalAmt is a integer type variable
if (principalAmt >= 10000){
   interest = principalAmt * intRate1;
 }else{
   interest = principalAmt * intRate2;
 }
// principalAmt is an integer type ARRAY
if (principalAmt[3] >= 10000){
   interest = principalAmt[3] * intRate1;
 }else{
   interest = principalAmt[3] * intRate2;
 }
```

Using Increment and Decrement Operators with Array Elements

- When using ++ and -- operators, don't confuse the element with the subscript

```
tests[i]++;  // adds 1 to tests[i]
tests[i++];  // increments i, but has
             // no effect on tests
```

Example:

int tests [5]={2,4,6,8,10}; i=0;
tests [i]++; //adds 1 to data of test [0] ➔2+1
tests [i++]; // from tests[0] to tests[1]

Copying One Array to Another

- Arrays: tests and tests2
- Cannot copy with an assignment statement:

```
tests2 = tests;  //won't work
```

- Must instead use a loop to copy element-by-element:

```
for (int indx=0; indx < ISIZE;
 indx++){
   tests2[indx] = tests[indx];
}
```

Comparing Arrays

- Like copying, cannot compare in a single expression:

```
if (tests2 == tests)
```

- Use a while loop with a boolean variable:

```
bool areEqual=true;
int indx=0;
while (areEqual && indx < ISIZE){
   if(tests[indx] != tests2[indx]{
      areEqual = false;
      break;
   }
   indx++;
}
```

Comparing Arrays

- Cannot use the == operator with the names of two arrays to determine they are equal.

```
int firstArray[] = { 5, 10, 15, 20, 25 };
int secondArray[] = { 5, 10, 15, 20, 25 };
if (firstArray == secondArray)        // This is a mistake.
    cout << "The arrays are the same.\n";
else
    cout << "The arrays are not the same.\n";
```

- The operator compares the beginning memory addresses of the arrays not the contents of the arrays.
- The code reports that arrays are not the same.

Comparing Arrays

- To compare the contents of two arrays, must compare the elements of the arrays.

```
const int SIZE = 5;
int firstArray[SIZE] = { 5, 10, 15, 20, 25 };
int secondArray[SIZE] = { 5, 10, 15, 20, 25 };
bool arraysEqual = true; // Flag variable
int count = 0;           // Loop counter variable

// Determine whether the elements contain the same data.
while (arraysEqual && count < SIZE)
{
    if (firstArray[count] != secondArray[count])
        arraysEqual = false;
    count++;
}

if (arraysEqual)
    cout << "The arrays are equal.\n";
else
    cout << "The arrays are not equal.\n";
```

Example: Finding size of array

```
#include<iostream>
using namespace std;
int main() {
    int a[]={1,2,3,4,5,6};
    int size;
    //total size of array/size of array
data type
    size=sizeof (a)/sizeof(int);
    cout<<size;
    return 0;
}
```

Largest Array Element

- Use a loop to examine each element and find the largest element (*i.e.*, one with the largest value)

```
int largest = tests[0];
for (int tnum = 1; tnum < ISIZE; tnum++){
    if (tests[tnum] > largest){
      largest = tests[tnum];
    }
}
cout << "Highest score is " << largest;
```

- A similar algorithm exists to find the smallest element

Exercise 3

a) Write a program that sums all the elements of arrays with size 10. Once summed, compute their average.

b) Write a program that implements linear search algorithm

»x= 15

12	17	15	11

Exercise 3a - Answer

Array-Ex-3-1.cpp

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6   int x;
7   int num[x];
8   int tempo=0,aveNum;
9
10  cout<<"Enter array size = ";
11  cin>>x;
12  cout<<endl;
13
14  for(int i=0;i<x;i++)
15  {
16      cout<<"Enter a number "<<i+1<<":";
17      cin>>num[x];
18      tempo=tempo+num[x];
19      aveNum=(tempo/x);
20  }
21  cout<<"Total sums = "<<tempo<<"."<<endl;
22  cout<<"Average = "<<aveNum<<".";
23
24  return 0;
25  }
```

Output:

```
Enter array size = 4

Enter a number 1:2
Enter a number 2:4
Enter a number 3:6
Enter a number 4:8
Total sums = 20.
Average = 5.
```

Exercise 3b - Answer

Array-Ex-3-2.cpp

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6   int z=0,x,y=0,i;
7   int num[4]={12 ,17, 15, 11};
8
9
10  cout<<"Enter number to search value in array ";
11  cin>>x;
12
13  for(i=0;i<4;i++)
14  {
15      if(num[i]==x)
16      {
17          z=i;
18      }
19
20      else if(num[i]!=x)
21      {
22          y=y+1;
23      }
24  }
25
26  if(y<i)
27  {
28      cout<<"Value "<<x<<" is in array num["<<z<<"]";
29  }
30
31  else if(y=i)
32  {
33      cout<<"No such value.";
34  }
35
36      return 0;
37  }
```

Exercise 3b - Answer

Output:

```
Enter number to search value in array 15
Value 15 is in array num[2]
```

```
Enter number to search value in array 10
No such value.
```

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

MRSL 1163
SCIENTIFIC COMPUTING FOR SYSTEM ENGINEER

Lecture 6 (2D & 3D Arrays)

Dr Nelidya Md Yusoff
Razak Faculty of Technology and Informatics

Character Arrays

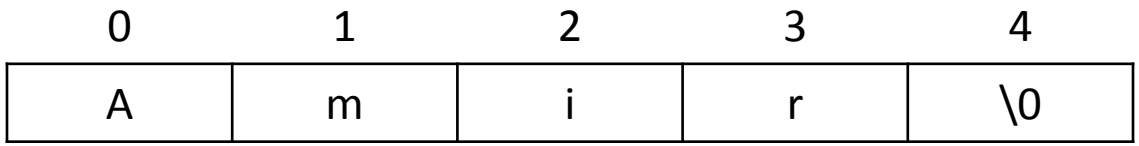
- Variable of 1 character:

  e.g. `char grade;`
- Variable that has more than 1 char known as 1D array.

  e.g. `char name[20];`
- Initialization **using string**

  `char name[]="Amir";`

```
name[0]='A'
name[1]='m'
name[2]='i'
name[3]='r'
```



Printing the Contents of an Array

- You can display the contents of a *character* array by sending its name to cout:

```
char fName[] = "Henry";
 cout << fName << endl;
```

But, this ONLY works with character arrays!

Printing the Contents of an Array

- For other types of arrays, you must print element-by-element:

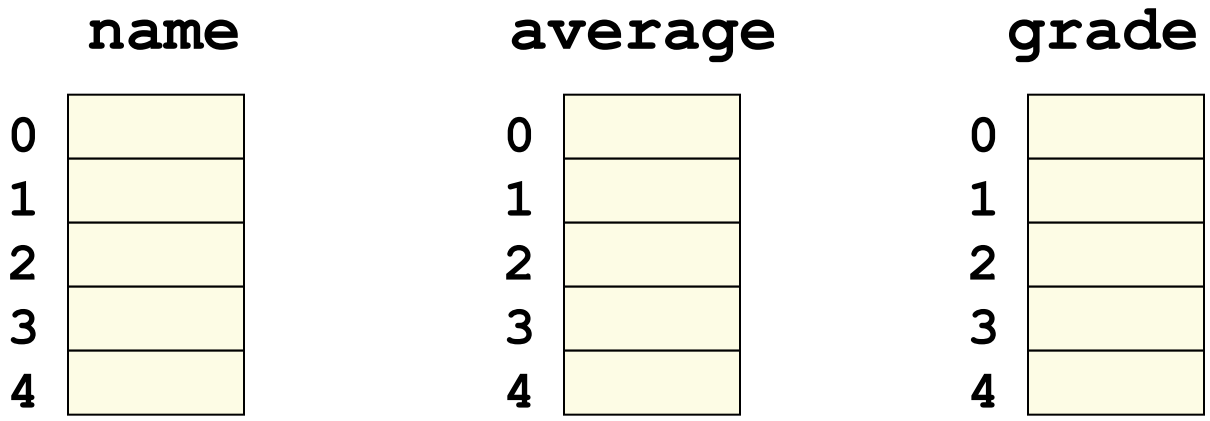
```
for (i = 0; i < ARRAY_SIZE; i++)
    cout << tests[i] << endl;
```

Using Parallel Arrays

- Parallel arrays: two or more arrays that contain related data by **using the same subscript**
- Subscript is used to relate arrays
  - elements at same subscript are related
- The arrays do not have to hold data of the same type

Parallel Array Example

```
const int ISIZE = 5;   // array size
 string name[ISIZE];    // student name
float average[ISIZE]; // course average
char grade[ISIZE];     // course grade
```



Parallel Array Processing

```
const int ISIZE = 5;
 string name[ISIZE];   // student name
float average[ISIZE]; // course average
char grade[ISIZE];    // course grade
...
for (int i = 0; i < ISIZE; i++)
    cout << " Student: " << name[i]
         << " Average: " << average[i]
         << " Grade: "   << grade[i]
         << endl;
```

Example Program 7-12

Program 7-12

```
1  // This program uses two parallel arrays: one for hours
2  // worked and one for pay rate.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     const int NUM_EMPLOYEES = 5;    // Number of employees
10    int hours[NUM_EMPLOYEES];       // Holds hours worked
11    double payRate[NUM_EMPLOYEES]; // Holds pay rates
12
13    // Input the hours worked and the hourly pay rate.
14    cout << "Enter the hours worked by " << NUM_EMPLOYEES
15         << " employees and their\n"
16         << "hourly pay rates.\n";
17    for (int index = 0; index < NUM_EMPLOYEES; index++)
18    {
19       cout << "Hours worked by employee #" << (index+1) << ": ";
20       cin >> hours[index];
21       cout << "Hourly pay rate for employee #" << (index+1) << ": ";
22       cin >> payRate[index];
23    }
24
```

(Program Continues)

```
25     // Display each employee's gross pay.
26     cout << "Here is the gross pay for each employee:\n";
27     cout << fixed << showpoint << setprecision(2);
28     for (int index = 0; index < NUM_EMPLOYEES; index++)
29     {
30        double grossPay = hours[index] * payRate[index];
31        cout << "Employee #" << (index + 1);
32        cout << ": $" << grossPay << endl;
33     }
34     return 0;
35  }
```

Program Output with Example Input Shown in Bold

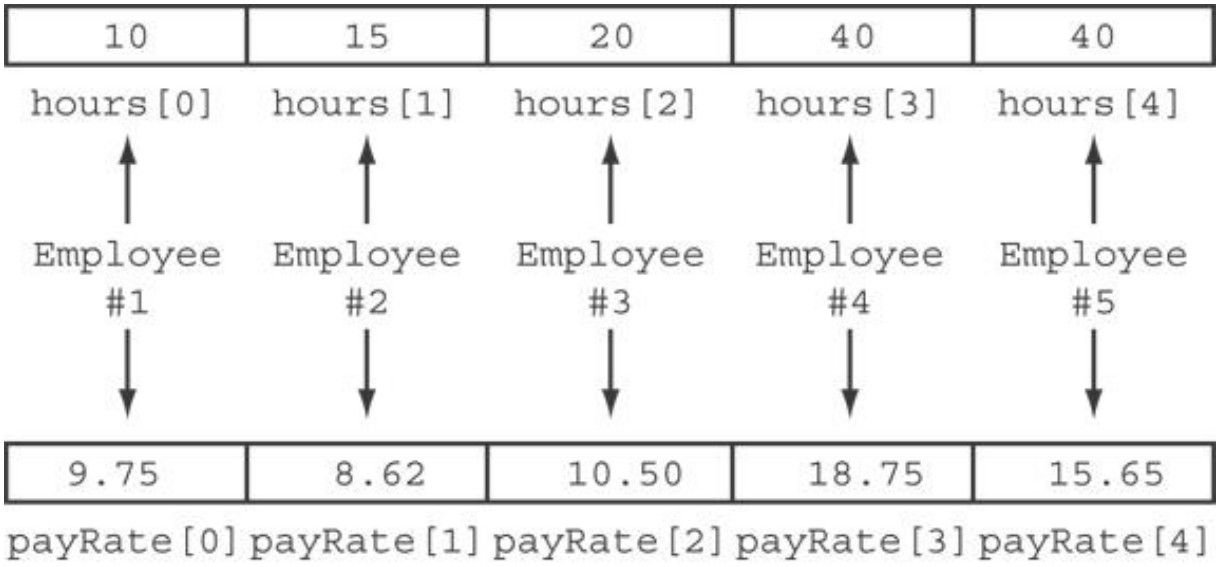
```
Enter the hours worked by 5 employees and their
hourly pay rates.
Hours worked by employee #1: 10 [Enter]
Hourly pay rate for employee #1: 9.75 [Enter]
Hours worked by employee #2: 15 [Enter]
Hourly pay rate for employee #2: 8.62 [Enter]
Hours worked by employee #3: 20 [Enter]
Hourly pay rate for employee #3: 10.50 [Enter]
Hours worked by employee #4: 40 [Enter]
Hourly pay rate for employee #4: 18.75 [Enter]
Hours worked by employee #5: 40 [Enter]
Hourly pay rate for employee #5: 15.65 [Enter]
```

(program output continues)

Program 7-12 *(continued)*

```
Here is the gross pay for each employee:
Employee #1: $97.50
Employee #2: $129.30
Employee #3: $210.00
Employee #4: $750.00
Employee #5: $626.00
```

The `hours` and `payRate` arrays are related through their subscripts:



Arrays as Function Arguments

- To **pass an array to a function**, just use the array name without any bracket:

```
showScores(tests);
```

- To define a function that takes an array parameter, use empty `[]` for array argument:

```
void showScores(int []);
                // function prototype
void showScores(int tests[])
                // function header
```

Arrays as Function Arguments

- When **passing an array** to a function, it is also common to **pass array size** so that function knows how many elements to process:

```
showScores(tests, ARRAY_SIZE);
```

- Array size must also be reflected in prototype, header:

```
void showScores(int [], int);
                // function prototype
void showScores(int tests[], int size)
                // function header
```

Passing an Array Element

- **Passing a single array element** to a function is no different than passing a regular variable of that data type
- Function does not need to know that the value it receives is coming from an array

```
displayValue(score[i]);          // call
void displayValue(int item){     // header
    cout << item << endl;
 }
```

Example Program 7-14

Program 7-14

```
1  // This program demonstrates an array being passed to a function.
2  #include <iostream>
3  using namespace std;
4
5  void showValues(int [], int); // Function prototype
6
7  int main()
8  {
9     const int ARRAY_SIZE = 8;
10    int numbers[ARRAY_SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
11
12    showValues(numbers, ARRAY_SIZE);
13    return 0;
14 }
15
```

(Program Continues)

```
16  //*********************************************
17  // Definition of function showValue.          *
18  // This function accepts an array of integers and  *
19  // the array's size as its arguments. The contents *
20  // of the array are displayed.                *
21  //*********************************************
22
23  void showValues(int nums[], int size)
24  {
25     for (int index = 0; index < size; index++)
26        cout << nums[index] << " ";
27     cout << endl;
28  }
```

Program Output

```
5 10 15 20 25 30 35 40
```

Modifying Arrays in Functions

- Array parameters in functions are similar to reference variables.
- They give the function direct access to the original array.
- Changes made with the array parameter in a function are actually made on the original array in the calling function.
- Must be careful that an array is not unintentionally changed by a function.

Example Program 7-19

Program 7-19

```
1  // This program uses a function to double the value of
2  // each element of an array.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototypes
7  void doubleArray(int [], int);
8  void showValues(int [], int);
9
10 int main()
11 {
12      const int ARRAY_SIZE = 7;
13      int set[ARRAY_SIZE] = {1, 2, 3, 4, 5, 6, 7};
14
15      // Display the initial values.
16      cout << "The array's values are:\n";
17      showValues(set, ARRAY_SIZE);
18
19      // Double the values in the array.
20      doubleArray(set, ARRAY_SIZE);
21
22      // Display the resulting values.
23      cout << "After calling doubleArray the values are:\n";
24      showValues(set, ARRAY_SIZE);
25
26      return 0;
27 }
28
```

(Program Continues)

Program 7-19 *(continued)*

```
29  //*****************************************************
30  // Definition of function doubleArray                 *
31  // This function doubles the value of each element    *
32  // in the array passed into nums. The value passed    *
33  // into size is the number of elements in the array.  *
34  //*****************************************************
35
36  void doubleArray(int nums[], int size)
37  {
38       for (int index = 0; index < size; index++)
39            nums[index] *= 2;
40  }
41
42  //*****************************************************
43  // Definition of function showValues.                 *
44  // This function accepts an array of integers and     *
45  // the array's size as its arguments. The contents    *
46  // of the array are displayed.                        *
47  //*****************************************************
48
49  void showValues(int nums[], int size)
50  {
51       for (int index = 0; index < size; index++)
52            cout << nums[index] << " ";
53       cout << endl;
54  }
```

Program Output

```
The array's values are:
1 2 3 4 5 6 7
After calling doubleArray the values are:
2 4 6 8 10 12 14
```

Two-Dimensional (2D) Arrays

- Can define one array for multiple sets of data
- Like a table in a spreadsheet
- Use two size declarators in definition:

```
const int ROWS = 4, COLS = 3;
int exams[ROWS][COLS];
```

- First declarator is number of rows; second is number of columns

2D Arrays Representation

```
const int ROWS = 4, COLS = 3; int
exams[ROWS][COLS];
```

columns

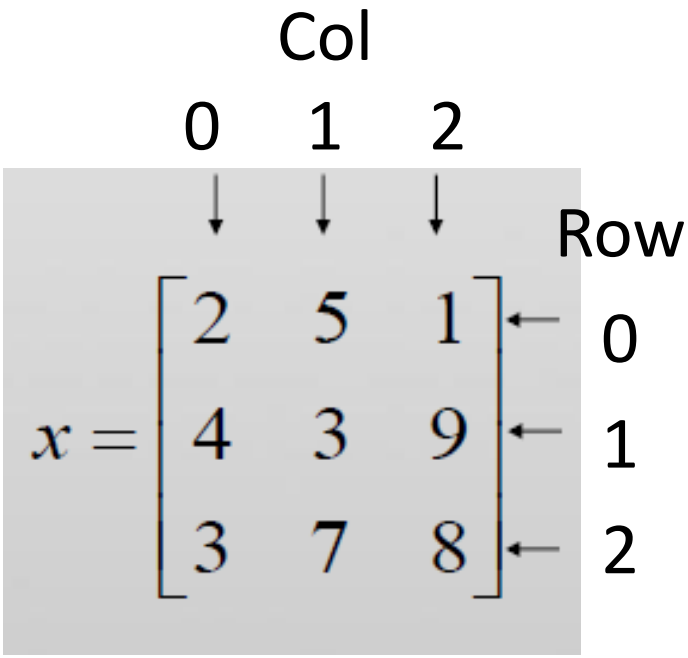
rows

exams[0][0]	exams[0][1]	exams[0][2]
exams[1][0]	exams[1][1]	exams[1][2]
exams[2][0]	exams[2][1]	exams[2][2]
exams[3][0]	exams[3][1]	exams[3][2]

- Use two subscripts to access element:

```
exams[2][2] = 86;
```

2D Arrays Declaration




```
int x [3][3];
```

- Declaration of 2D array
- **SYNTAX**

```
DatatypeArray[nr][nc];
```

- nr is the number of row
- nc is the number of column

- Example

```
double markah [4][7];
```

2D Arrays Declaration

- Data input for 2D array

```
for (int row=0;row<3;row++)
{
for (int column=0;column<3;column++)
{
cin>>x[row][column];
cout<<x[row][column]<<" ";
}
cout<<endl;
}
```

Example Program 7-18

Program 7-18

```
1  // This program demonstrates a two-dimensional array.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  int main()
7  {
8     const int NUM_DIVS = 3;           // Number of divisions
9     const int NUM_QTRS = 4;           // Number of quarters
10    double sales[NUM_DIVS][NUM_QTRS]; // Array with 3 rows and 4 columns.
11    double totalSales = 0;            // To hold the total sales.
12    int div, qtr;                     // Loop counters.
13
14    cout << "This program will calculate the total sales of\n";
15    cout << "all the company's divisions.\n";
16    cout << "Enter the following sales information:\n\n";
17
```

(program continues)

Program 7-18 *(continued)*

```
18      // Nested loops to fill the array with quarterly
19      // sales figures for each division.
20      for (div = 0; div < NUM_DIVS; div++)
21      {
22         for (qtr = 0; qtr < NUM_QTRS; qtr++)
23         {
24            cout << "Division " << (div + 1);
25            cout << ", Quarter " << (qtr + 1) << ": $";
26            cin >> sales[div][qtr];
27         }
28         cout << endl; // Print blank line.
29      }
30
31      // Nested loops used to add all the elements.
32      for (div = 0; div < NUM_DIVS; div++)
33      {
34         for (qtr = 0; qtr < NUM_QTRS; qtr++)
35            totalSales += sales[div][qtr];
36      }
37
38      cout << fixed << showpoint << setprecision(2);
39      cout << "The total sales for the company are: $";
40      cout << totalSales << endl;
41      return 0;
42   }
```

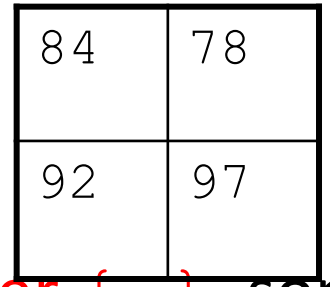
Program Output with Example Input Shown in Bold

```
This program will calculate the total sales of
all the company's divisions.
Enter the following sales data:

Division 1, Quarter 1: $31569.45 [Enter]
Division 1, Quarter 2: $29654.23 [Enter]
Division 1, Quarter 3: $32982.54 [Enter]
Division 1, Quarter 4: $39651.21 [Enter]

Division 2, Quarter 1: $56321.02 [Enter]
Division 2, Quarter 2: $54128.63 [Enter]
Division 2, Quarter 3: $41235.85 [Enter]
Division 2, Quarter 4: $54652.33 [Enter]

Division 3, Quarter 1: $29654.35 [Enter]
Division 3, Quarter 2: $28963.32 [Enter]
Division 3, Quarter 3: $25353.55 [Enter]
Division 3, Quarter 4: $32615.88 [Enter]

The total sales for the company are: $456782.34
```

2D Arrays Initialization

- Two-dimensional arrays are **initialized row-by-row**:

```
const int ROWS = 2, COLS = 2;
int exams[ROWS][COLS] = { {84,78},{92,97} };
```



- Can omit inner `{ }`, some initial values in a row – array elements without initial values will be set to `0` or `NULL`

2D Arrays Initialization

- Initialization of 2D array

a) `int x[3][3]={{2,5,1},{4,3,9},{3,7,8}};`

b) `int x[3][3]= {2,5,1,4,3,9,3,7,8};`

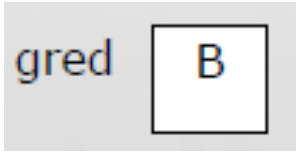
c) `int x[3][3]= {2,5,1,9,3,7,8};`

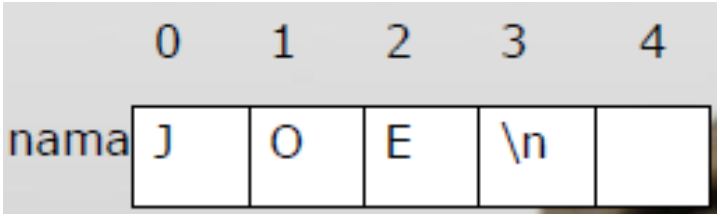
$$x=\begin{bmatrix}2 & 5 & 1\\ 4 & 3 & 9\\ 3 & 7 & 8\end{bmatrix} \quad x=\begin{bmatrix}2 & 5 & 1\\ 4 & 3 & 9\\ 3 & 7 & 8\end{bmatrix} \quad x=\begin{bmatrix}2 & 5 & 1\\ 9 & 3 & 7\\ 8 & 0 & 0\end{bmatrix}$$

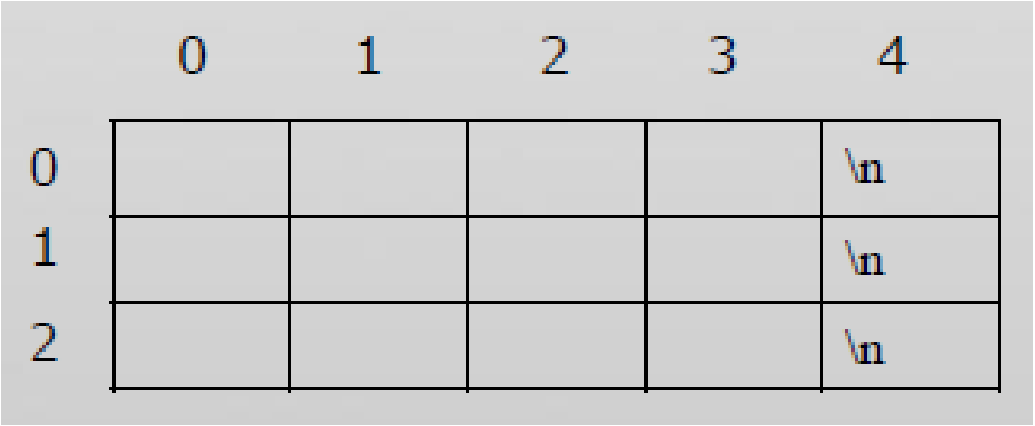
2D Arrays

- Arrangement of 2D Arrays.

```
char gred; gred='B';
char nama[5]="JOE";
char nama[3][5];
```







2D Arrays

- Initialization of 2D char array

```
fullName[4][10]={"Alex Jamy", "Edy
Gonza", "Adam Les", "Zack Yus"}
```

fullName[0]	A	l	e	x		J	a	m	y	\n
fullName[1]	E	d	y		G	o	n	z	a	\n
fullName[2]										
fullName[3]										

Passing a 2D Array to a Function

- Two-dimensional arrays can be **passed as parameters to a function**, and they are passed by reference.
- When declaring a 2D array as a formal parameter, we **can omit the size of the first dimension (row)**, but not the second; that is, we must **specify the number of columns.**
- For example in function header:

```
void print(int A[][3],int N, int M)
```

Passing a 2D Array to a Function

- Use array name as argument in **function call**

```
getExams(exams, 2);
```

- Use empty `[]` for row and a size declarator for col in the prototype and header

```
const int COLS = 2;
// Prototype
void getExams(int [][COLS], int);

// Header
void getExams(int exam[][COLS], int
rows)
```

Passing a 2D Array to a Function

- Prototype:

```
void display(int [ ][2]);
void grade (char [ ][20]);
```

- Function Call:

```
display(num, 2);
grade (name, 20);
```

- Header:

```
void display(int number[ ][2])
void grade(char names[ ][20])
```

Example Program 7-22

Program 7-22

```
1  // This program demonstrates accepting a 2D array argument.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  // Global constants
7  const int COLS = 4;       // Number of columns in each array
8  const int TBL1_ROWS = 3; // Number of rows in table1
9  const int TBL2_ROWS = 4; // Number of rows in table2
10
11 void showArray(const int [][COLS], int); // Function prototype
12
13 int main()
14 {
15     int table1[TBL1_ROWS][COLS] = {{1, 2, 3, 4},
16                                    {5, 6, 7, 8},
17                                    {9, 10, 11, 12}};
18     int table2[TBL2_ROWS][COLS] = {{10, 20, 30, 40},
19                                    {50, 60, 70, 80},
20                                    {90, 100, 110, 120},
21                                    {130, 140, 150, 160}};
```

(Program Continues)

```
23        cout << "The contents of table1 are:\n";
24        showArray(table1, TBL1_ROWS);
25        cout << "The contents of table2 are:\n";
26        showArray(table2, TBL2_ROWS);
27        return 0;
28  }
29
30  //*************************************************************
31  // Function Definition for showArray                          *
32  // The first argument is a two-dimensional int array with COLS *
33  // columns. The second argument, rows, specifies the number of *
34  // rows in the array. The function displays the array's contents. *
35  //*************************************************************
36
37  void showArray(const int numbers[][COLS], int rows)
38  {
39        for (int x = 0; x < rows; x++)
40        {
41              for (int y = 0; y < COLS; y++)
42              {
43                    cout << setw(4) << numbers[x][y] << " ";
44              }
45              cout << endl;
46        }
47  }
```

Respective row

* This function can accept any 2D integer array as long as it consists of four columns.

Program Output

```
The contents of table1 are:
   1    2    3    4
   5    6    7    8
   9   10   11   12
The contents of table2 are:
  10   20   30   40
  50   60   70   80
  90  100  110  120
 130  140  150  160
```

Summing All the Elements of a 2D Array

- Given the following definitions:

```
const int NUM_ROWS = 5; // Number of rows
const int NUM_COLS = 5; // Number of columns
int total = 0;          // Accumulator
int numbers[NUM_ROWS][NUM_COLS] =
   {{2, 7, 9, 6, 4},
    {6, 1, 8, 9, 4},
    {4, 3, 7, 2, 9},
    {9, 9, 0, 3, 1},
    {6, 2, 7, 4, 1}};
```

Summing All the Elements of a 2D Array

```
// Sum the array elements.
for (int row = 0; row < NUM_ROWS; row++)
{
   for (int col = 0; col < NUM_COLS;
col++)
      total += numbers[row][col];
}

// Display the sum.
cout << "The total is " << total << endl;
```

Summing All the Elements of a 2D Array

```
1   #include <iostream>
2   #include<iomanip>
3   using namespace std;
4
5   int main()
6   {
7       const int NUM_ROWS = 5; // Number of rows
8       const int NUM_COLS = 5; // Number of columns
9       int total = 0;          // Accumulator
10      int numbers[NUM_ROWS][NUM_COLS] =
11          {{2, 7, 9, 6, 4},
12          {6, 1, 8, 9, 4},
13          {4, 3, 7, 2, 9},
14          {9, 9, 0, 3, 1},
15          {6, 2, 7, 4, 1}};
16
17      // Sum the array elements.
18      for (int row = 0; row < NUM_ROWS; row++)
19      {
20          for (int col = 0; col < NUM_COLS; col++)
21          total += numbers[row][col];
22      }
23      // Display the sum.
24      cout << "The total is " << total << endl;
25
26
27  return 0;
28  }
```

Summing the Rows of a 2D Array

- Given the following definitions:

```
const int NUM_STUDENTS = 3;
const int NUM_SCORES = 5;
double total;   // Accumulator
double average; // To hold average scores
double scores[NUM_STUDENTS][NUM_SCORES] =
      {{88, 97, 79, 86, 94},
       {86, 91, 78, 79, 84},
       {82, 73, 77, 82, 89}};
```

Summing the Rows of a 2D Array

```
// Get each student's average score.
for (int row = 0; row < NUM_STUDENTS; row++)
{
   // Set the accumulator.
   total = 0;
   // Sum a row.
   for (int col = 0; col < NUM_SCORES; col++)
      total += scores[row][col];
   // Get the average
   average = total / NUM_SCORES;
   // Display the average.
   cout << "Score average for student "
        << (row + 1) << " is " << average <<endl;
}
```

Summing the Columns of a 2D Array

- Given the following definitions:

```
const int NUM_STUDENTS = 3;
const int NUM_SCORES = 5;
double total;   // Accumulator
double average; // To hold average scores
double scores[NUM_STUDENTS][NUM_SCORES] =
      {{88, 97, 79, 86, 94},
       {86, 91, 78, 79, 84},
       {82, 73, 77, 82, 89}};
```

Summing the Columns of a 2D Array

```
// Get the class average for each score
for (int col = 0; col < NUM_SCORES; col++)
{
   // Reset the accumulator
   total = 0;
   // Sum a column
   for (int row = 0; row < NUM_STUDENTS; row++)
      total += scores[row][col];
   // Get the average
   average = total / NUM_STUDENTS;
   // Display the class average
   cout << "Class average for test " << (col + 1)
        << " is " << average << endl;
}
```

Arrays with Three or More Dimensions

- C++ does **not limit the number of dimensions** that an array may have.
- It is possible to create arrays with multiple dimensions, to model data that occur in multiple sets.
- Can define arrays with any number of dimensions:

```
short rectSolid[2][3][5];
double timeGrid[3][4][3][4];
```

Arrays with Three or More Dimensions

- Arrays with **more than three dimensions are difficult to visualize,** but can be useful in some programming problems
- When used as parameter, specify all except the 1st dimension in function prototype and heading:

```
void getRectSolid(short [][3][5]);
```

3D Arrays

- Example of 3D array:

```
double seats[3][5][8]);
```

- Can be thought of as **three sets** of **five rows**, which each row containing **eight elements**.

Figure 7-18