Chapter 1:
An Overview of Computers and Programming Languages

Outlines

- In this chapter, you will study:
  - Processing a C++ Program
  - Programming with the Problem Analysis–Coding–Execution Cycle

Introduction

- Without software, the computer is useless
- Software is developed with programming languages
  - C++ is a programming language
- C++ suited for a wide variety of programming tasks

Processing a C++ Program

```
#include <iostream>
using namespace std;
int main()
{
     cout << "My first C++ program." << endl;
    return 0;
}
```

Sample Run:

```
My first C++ program.
```

Processing a C++ Program (cont'd.)

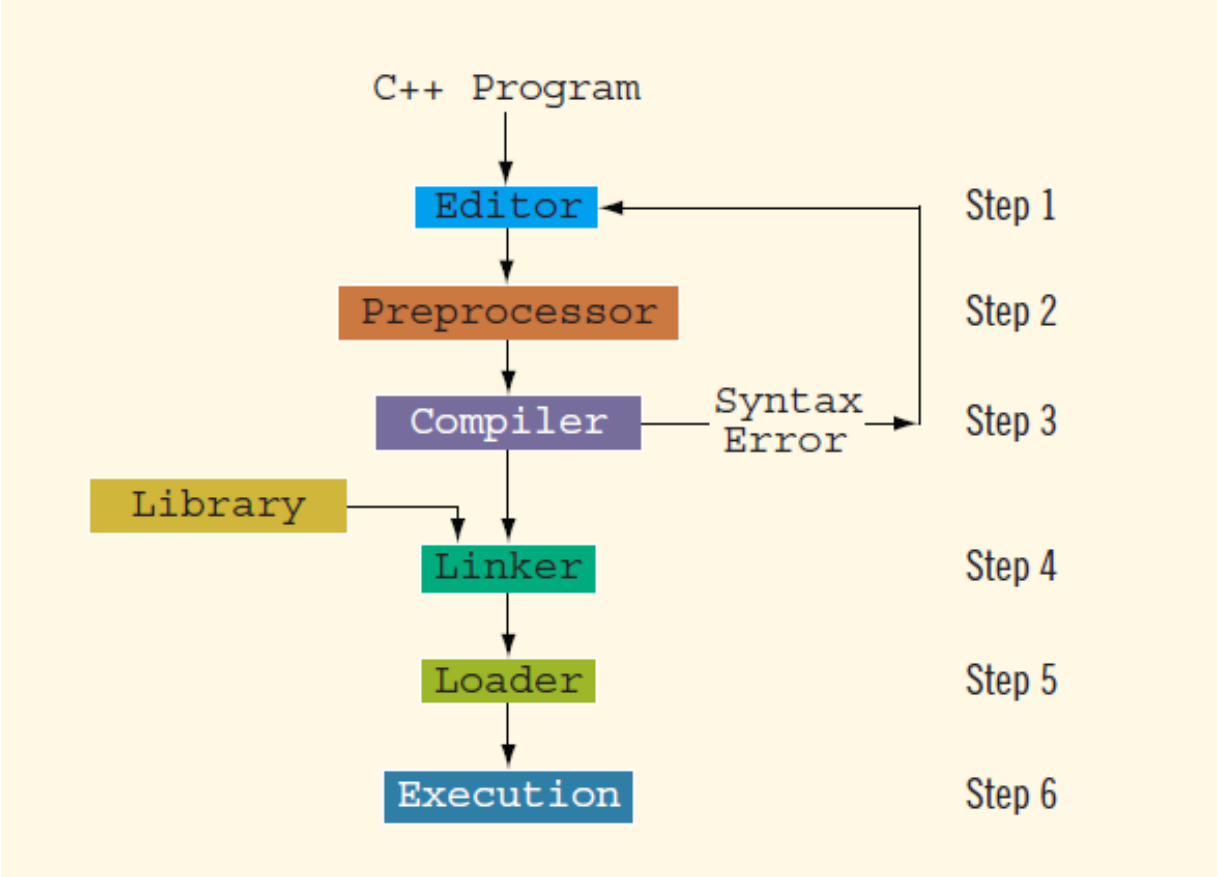


FIGURE 1-3 Processing a C++ program

Processing a C++ Program (cont'd.)

- To execute a C++ program:
  - Use an editor to create a source program in C++
  - Preprocessor directives begin with # and are processed by the preprocessor
  - Use the compiler to:
    - Check that the program obeys the language rules
    - Translate into machine language (object program)

Processing a C++ Program (cont’d.)

- To execute a C++ program (cont'd.):
  - Linker:
    - Combines object program with other programs provided by the SDK to create executable code
    - Library: contains prewritten code you can use
  - Loader:
    - Loads executable program into main memory
  - The last step is to execute the program
- Some IDEs do all this with a Build or Rebuild command

Programming with the Problem Analysis–Coding–Execution Cycle

- Algorithm:
  - Step-by-step problem-solving process
  - Solution achieved in finite amount of time
- Programming is a process of problem solving

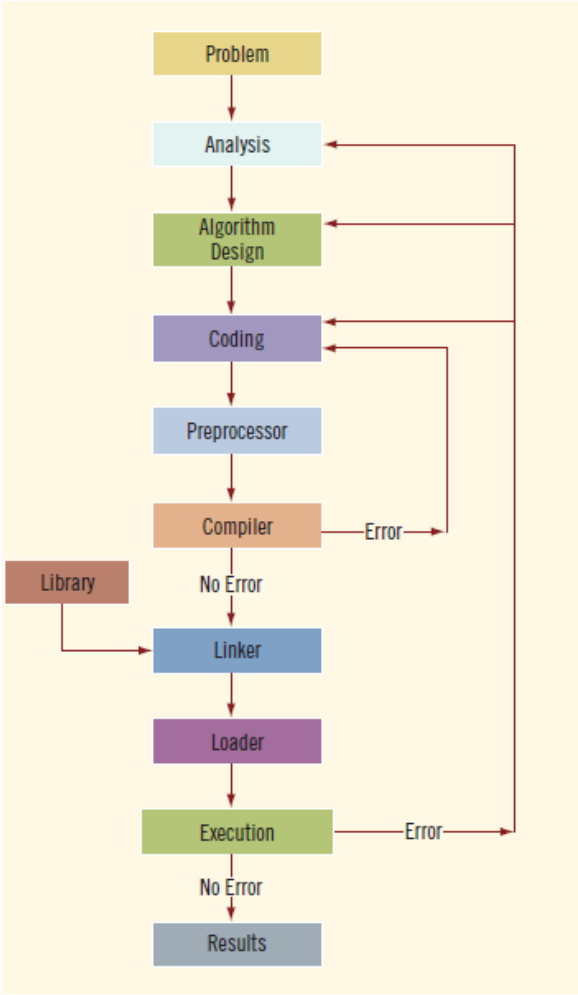


FIGURE 1-4 Problem analysis–coding–execution cycle

The Problem Analysis–Coding–Execution Cycle (cont'd.)

- Step 1: Analyze the problem
  - Thoroughly understand the problem and all requirements
    - Does program require user interaction?
    - Does program manipulate data?
    - What is the output?
  - If the problem is complex, divide it into subproblems
    - Analyze and design algorithms for each subproblem
  - Check the correctness of algorithm
    - Can test using sample data
    - Some mathematical analysis might be required

The Problem Analysis–Coding–Execution Cycle (cont'd.)

- Step 2: Implement the algorithm
  - Implement the algorithm in code
  - Verify that the algorithm works
  - Once the algorithm is designed and correctness verified, Write the equivalent code in high-level language
  - Enter the program using text editor

The Problem Analysis–Coding–Execution Cycle (cont'd.)

- Step 3: Maintenance
  - Use and modify the program if the problem domain changes
  - Run code through compiler
  - If compiler generates errors
    - Look at code and remove errors
    - Run code again through compiler
  - If there are no syntax errors
    - Compiler generates equivalent machine code
  - Linker links machine code with system resources

The Problem Analysis–Coding–Execution Cycle (cont'd.)

- Once compiled and linked, loader can place program into main memory for execution
- The final step is to execute the program
- Compiler guarantees that the program follows the rules of the language
  - Does not guarantee that the program will run correctly

Example 1-1

- Design an algorithm to find the perimeter and area of a rectangle
- The perimeter and area of the rectangle are given by the following formulas:

```
perimeter = 2 * (length + width)
area = length * width
```

Example 1-1 (cont'd.)

- Algorithm:
  - Get length of the rectangle
  - Get width of the rectangle
  - Find the perimeter using the following equation:

```
perimeter = 2 * (length + width)
```

  - Find the area using the following equation:

```
area = length * width
```

Example 1-3

- Design an algorithm to calculates the monthly paycheck of a salesperson at a local department store.

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

- Data:
  - base salary
  - The number of years that the salesperson has been with the company
  - The total sales made by the salesperson for that month

Example 1-3 (cont'd.)

- Suppose **noOfServiceYears** denotes the number of years that the salesperson has been with the store
- Suppose **bonus** denotes the bonus.

- Algorithm to calculate the bonus

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

Example 1-3 (cont'd.)

- Suppose **totalSales** denotes the total sales made by the salesperson for the month
- Suppose **additionalBonus** denotes the additional bonus.

- Algorithm to calculate additional bonus

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

Example 1-3 (cont'd.)

- The algorithm to calculate a salesperson's monthly paycheck:
  1. Get baseSalary.
  2. Get noOfServiceYears.
  3. Calculate bonus using calculate bonus algorithm
  4. Get totalSales.
  5. Calculate additionalBonus using the algorithm to calculate additional bonus
  6. Calculate payCheck using the equation:

  payCheck = baseSalary + bonus + additionalBonus

Example 1-5

- Calculate each student's grade
  - 10 students in a class; each student has taken five tests; each test is worth 100 points
- Design algorithms to:
  - Calculate the grade for each student and class average
  - Find the average test score
  - Determine the grade
- Data: students' names; test scores

Example 1-5 (cont'd.)

- Algorithm to determine the average test score:
  - Get the five test scores
  - Add the five test scores
    - Suppose `sum` stands for the sum of the test scores
  - Suppose `average` stands for the average test score:
    - `average = sum / 5;`

Example 1-5 (cont'd.)

- Algorithm to determine the grade:

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

Example 1-5 (cont'd.)

- Main algorithm is as follows:
  - `totalAverage` = 0;
  - Repeat the following for each student:
    - Get student's name
    - Use the algorithm to find the average test score
    - Use the algorithm to find the grade
    - Update `totalAverage` by adding current student's average test score
  - Determine the class average as follows:
    - `classAverage = totalAverage / 10`

Refer to text book and read examples 1-2, and 1-4

Compiling C++ Code

Self Study Slides

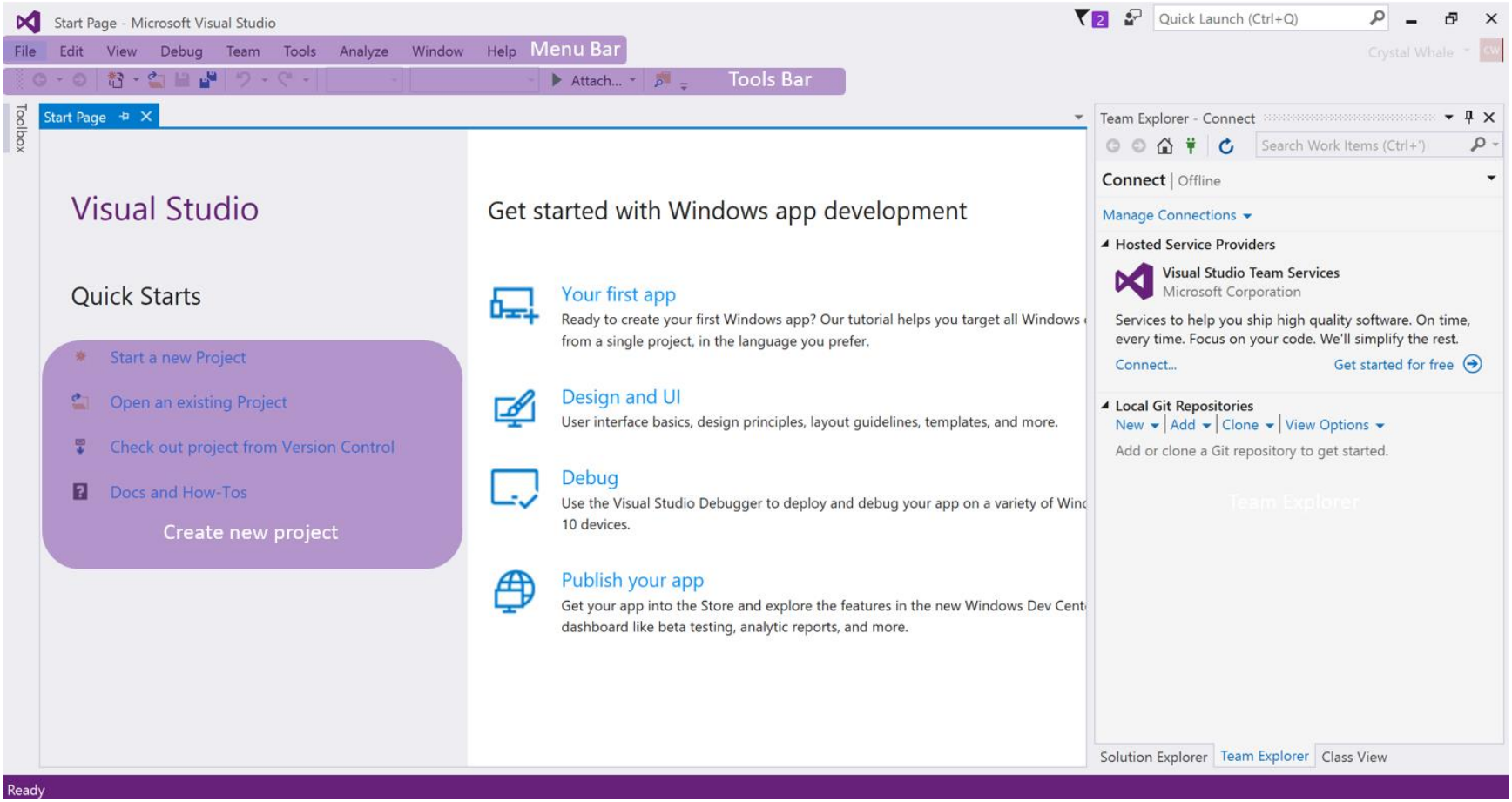
C++ PROGRAMMING:
From Problem Analysis to Program Design

Some examples on C++ IDEs

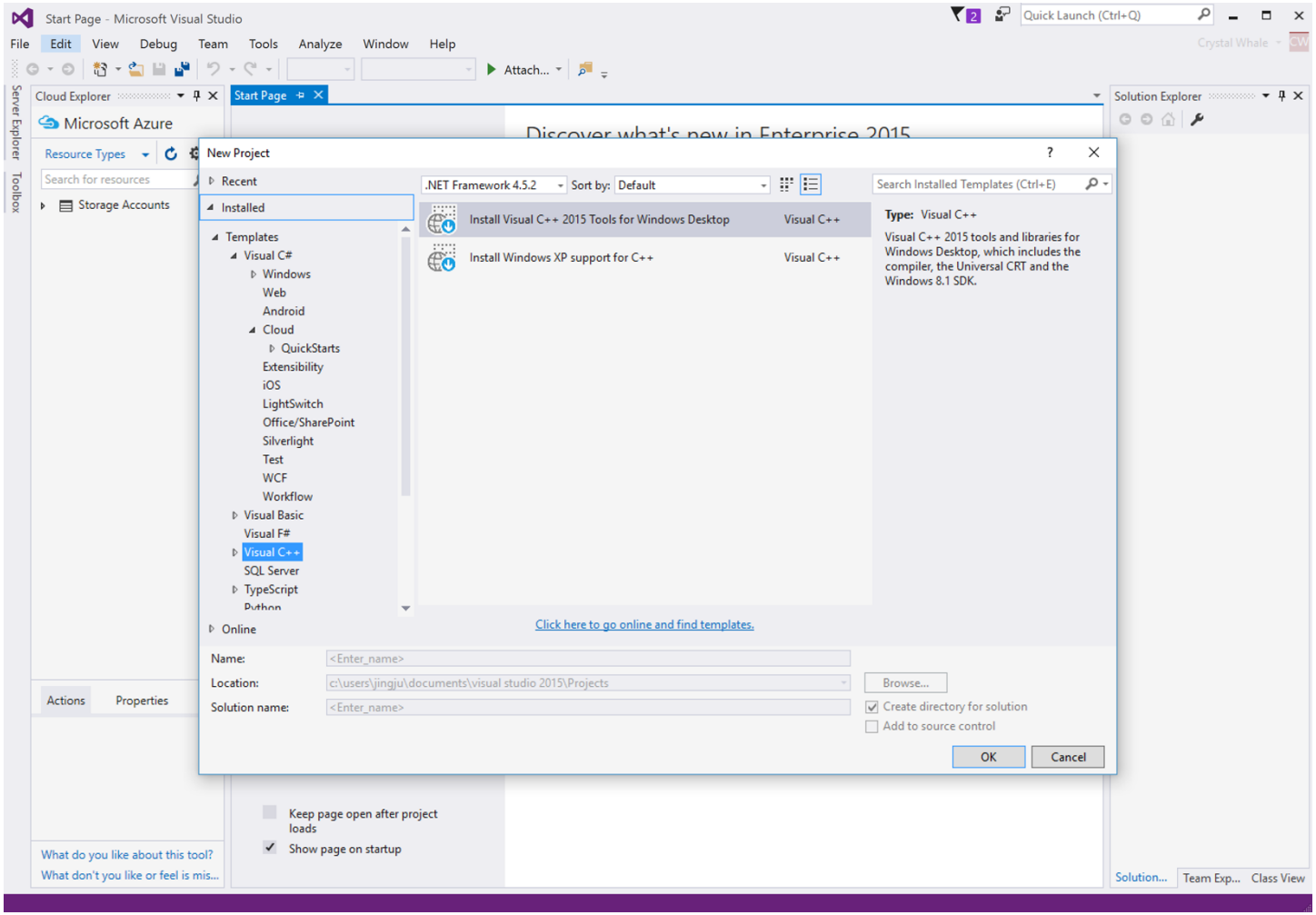
- Online IDEs
  - https://www.codechef.com/ide
  - https://www.onlinegdb.com/
  - https://www.jdoodle.com/online-compiler-c++

- Offline IDEs
  - Microsoft visual studio (will be explained in this slides)
  - Eclipse
  - Code::Blocks
  - CodeLite

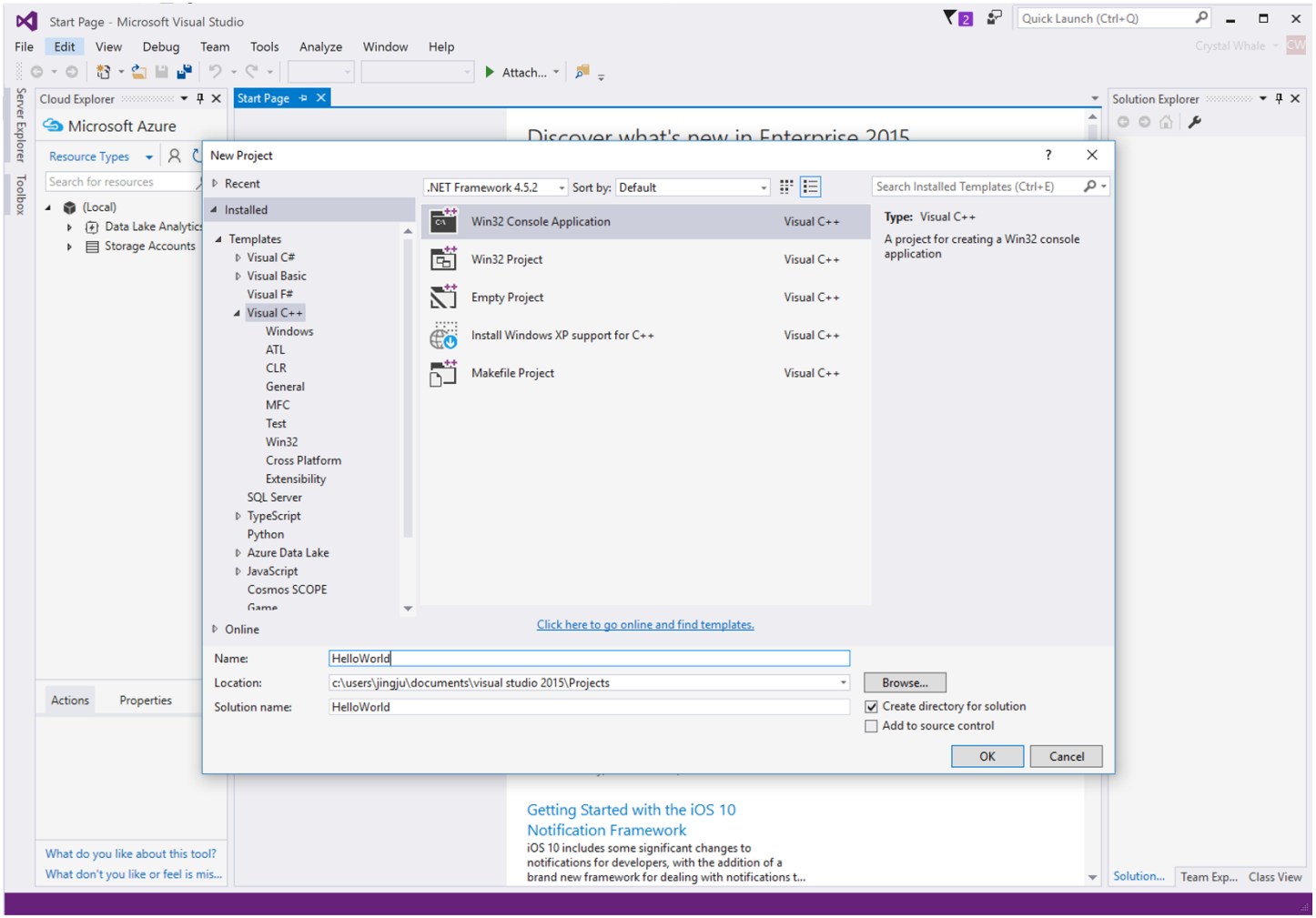
Microsoft visual studio 2015 home screen



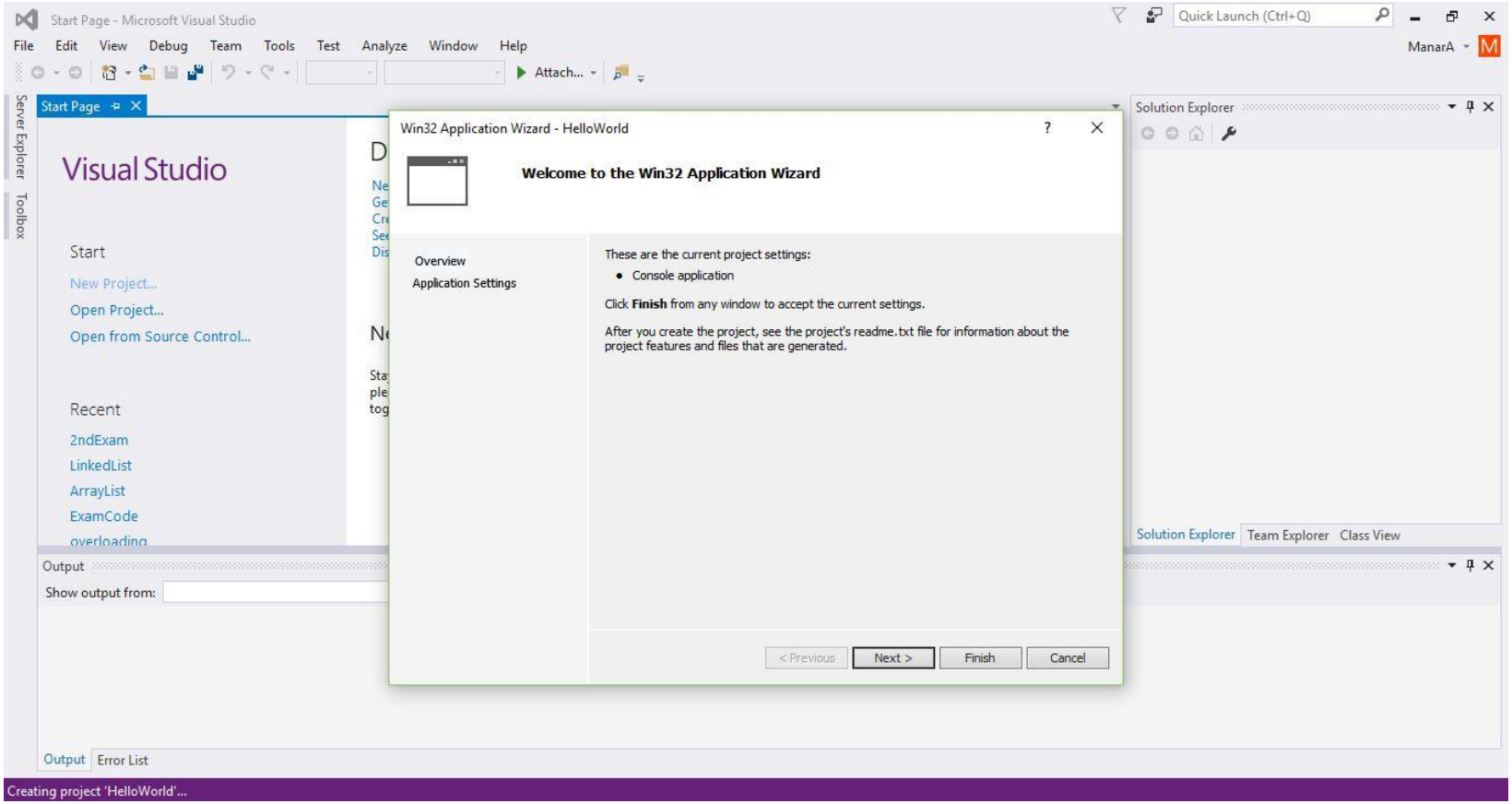
Before you create your first C++ project in Visual Studio, you need to install Visual C++ 2015 Tools for Windows Desktop:



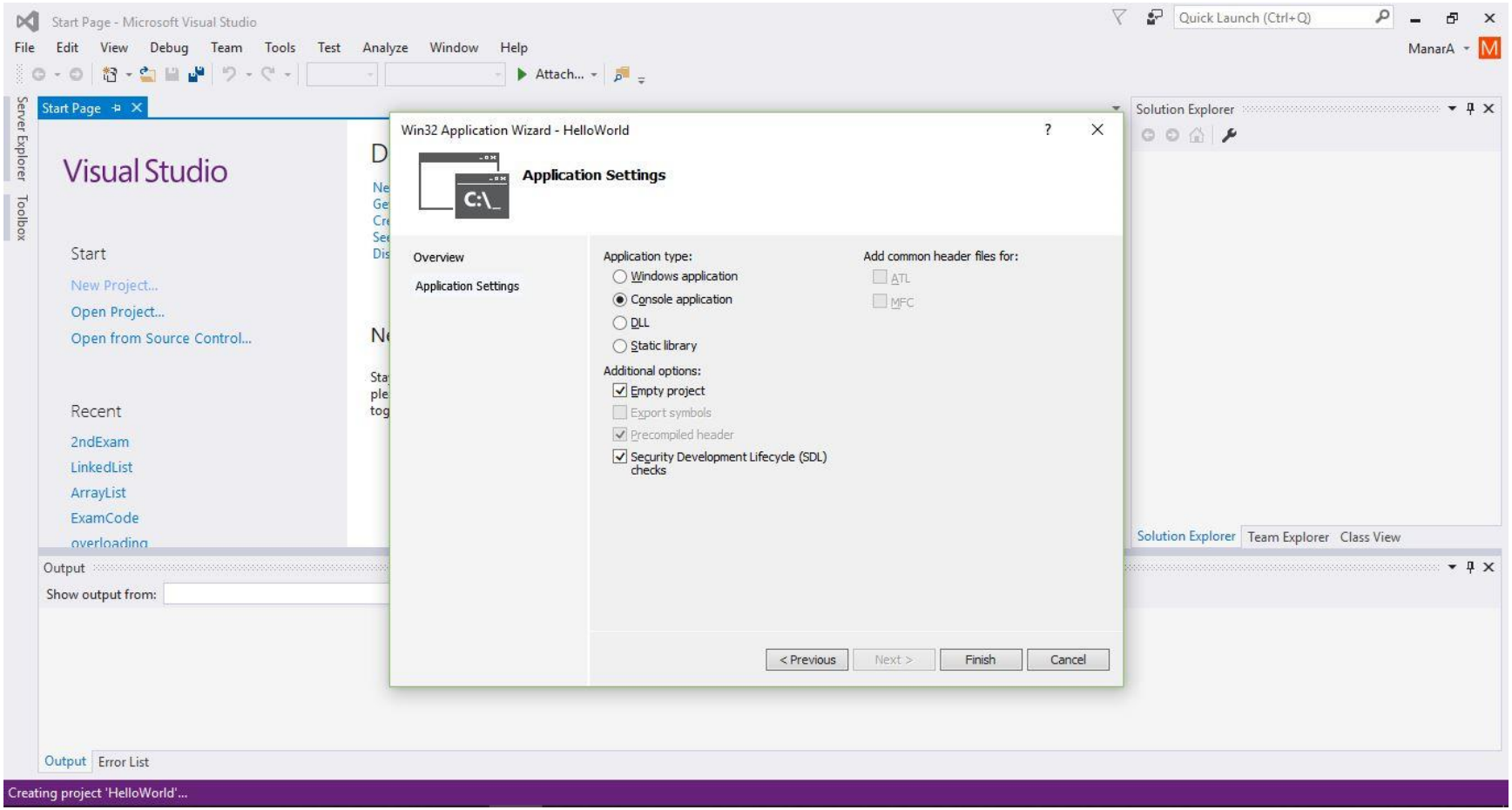
To create your HelloWorld project => File ->new->project, you can choose the Win32 Console Application template, Name your project and click “ok”



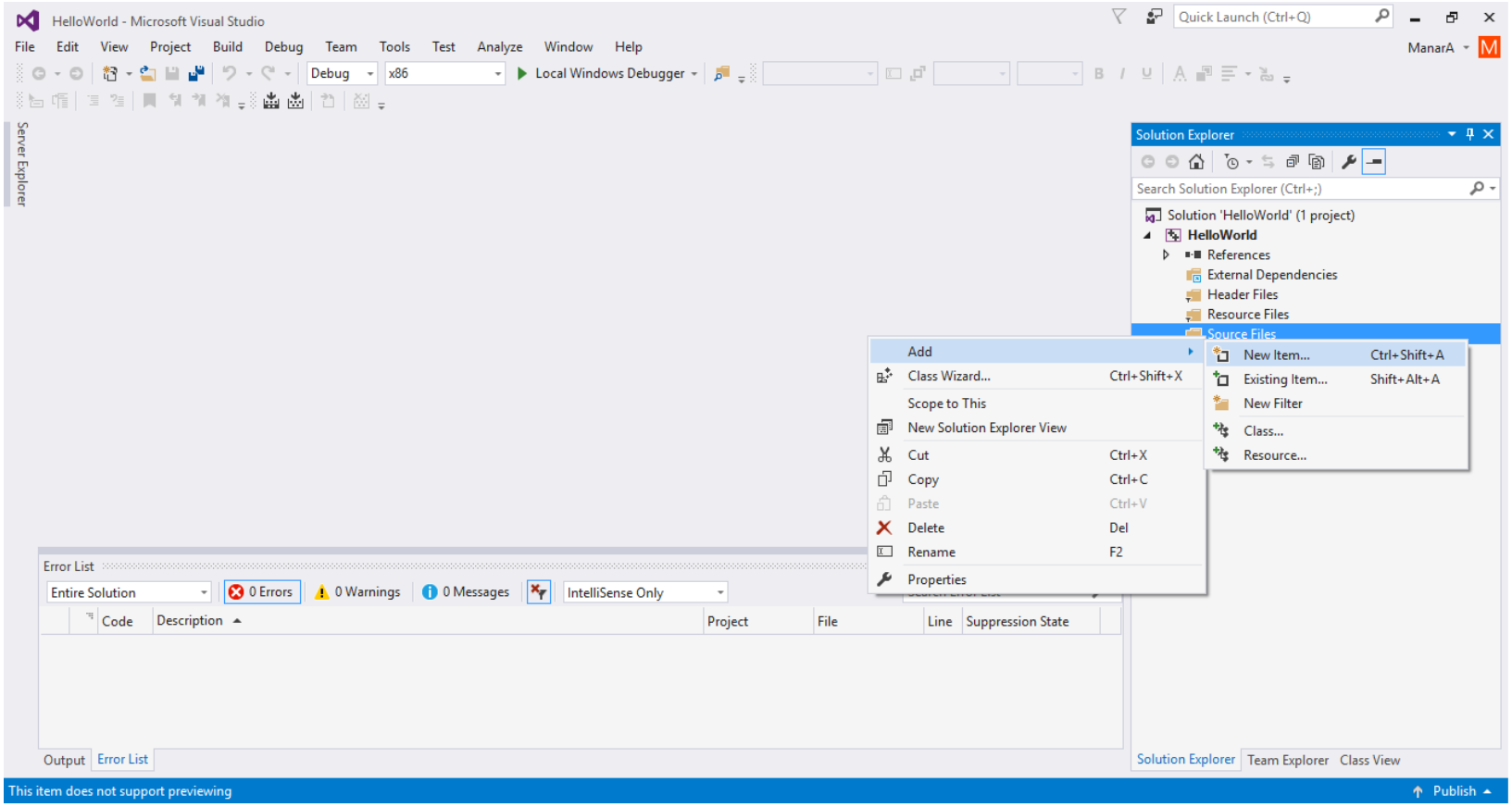
When this window appears click next



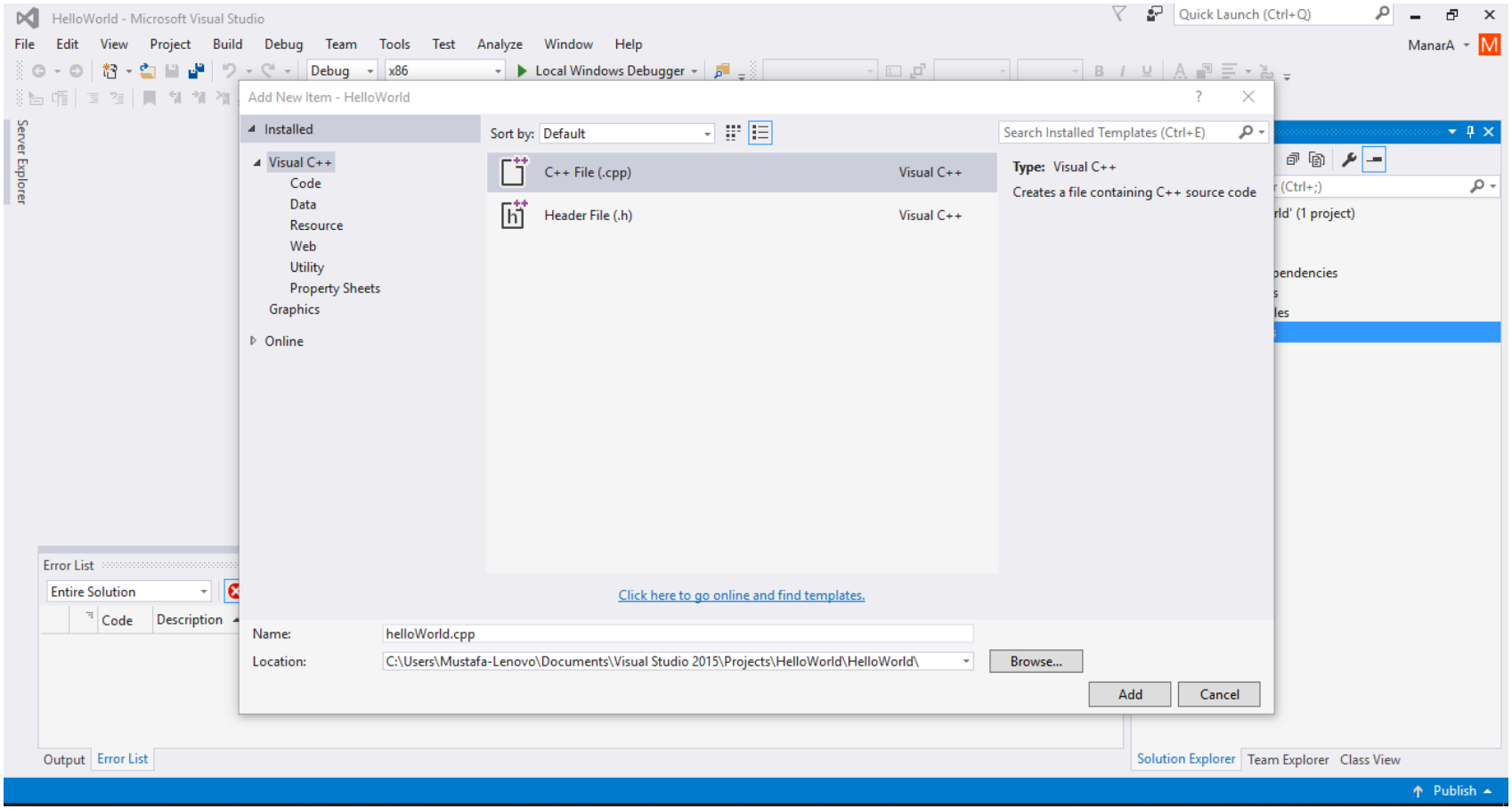
Choose Empty project and click finish



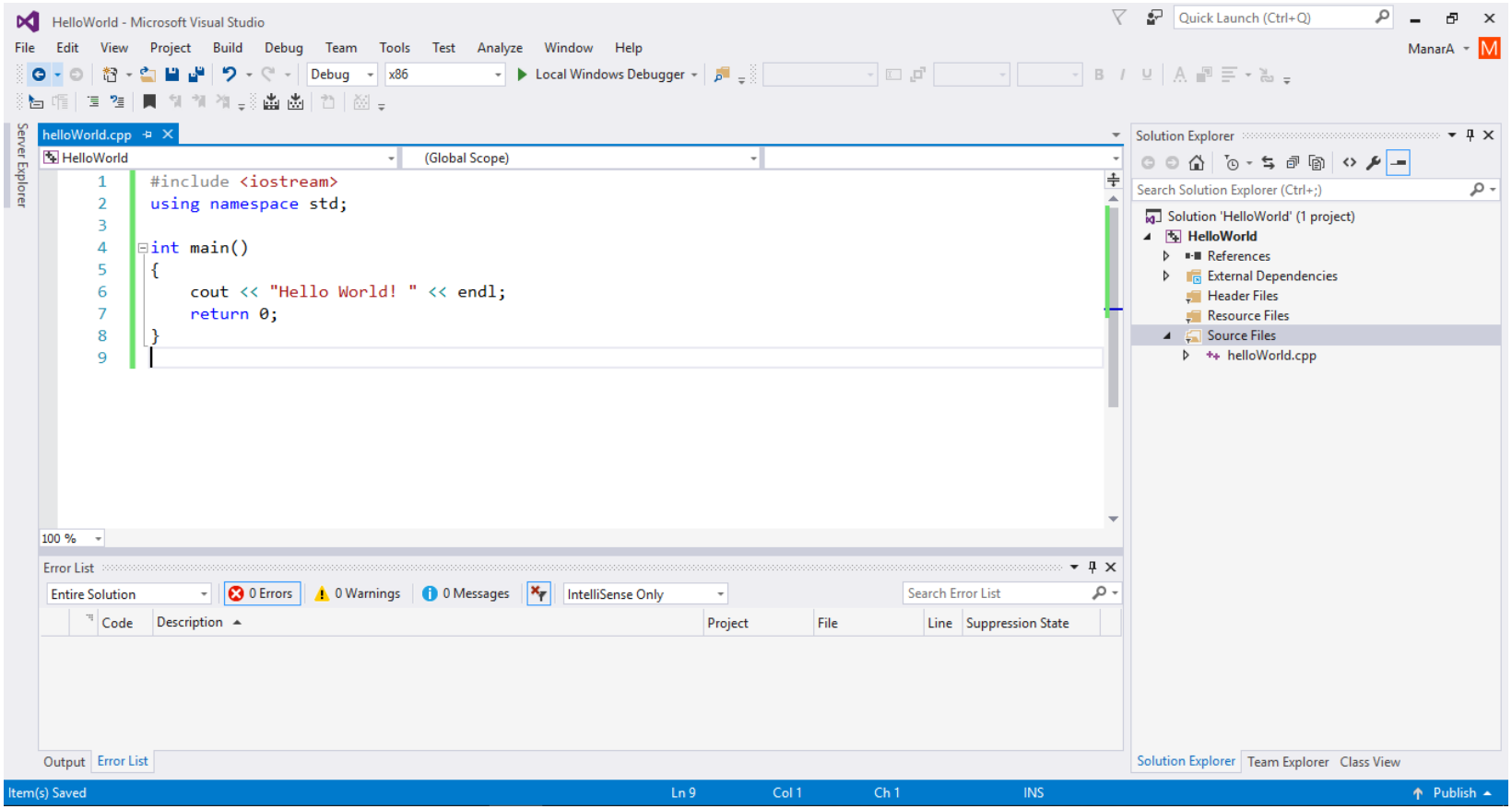
From solution explorer window, right click on source files and choose to add new item as follows



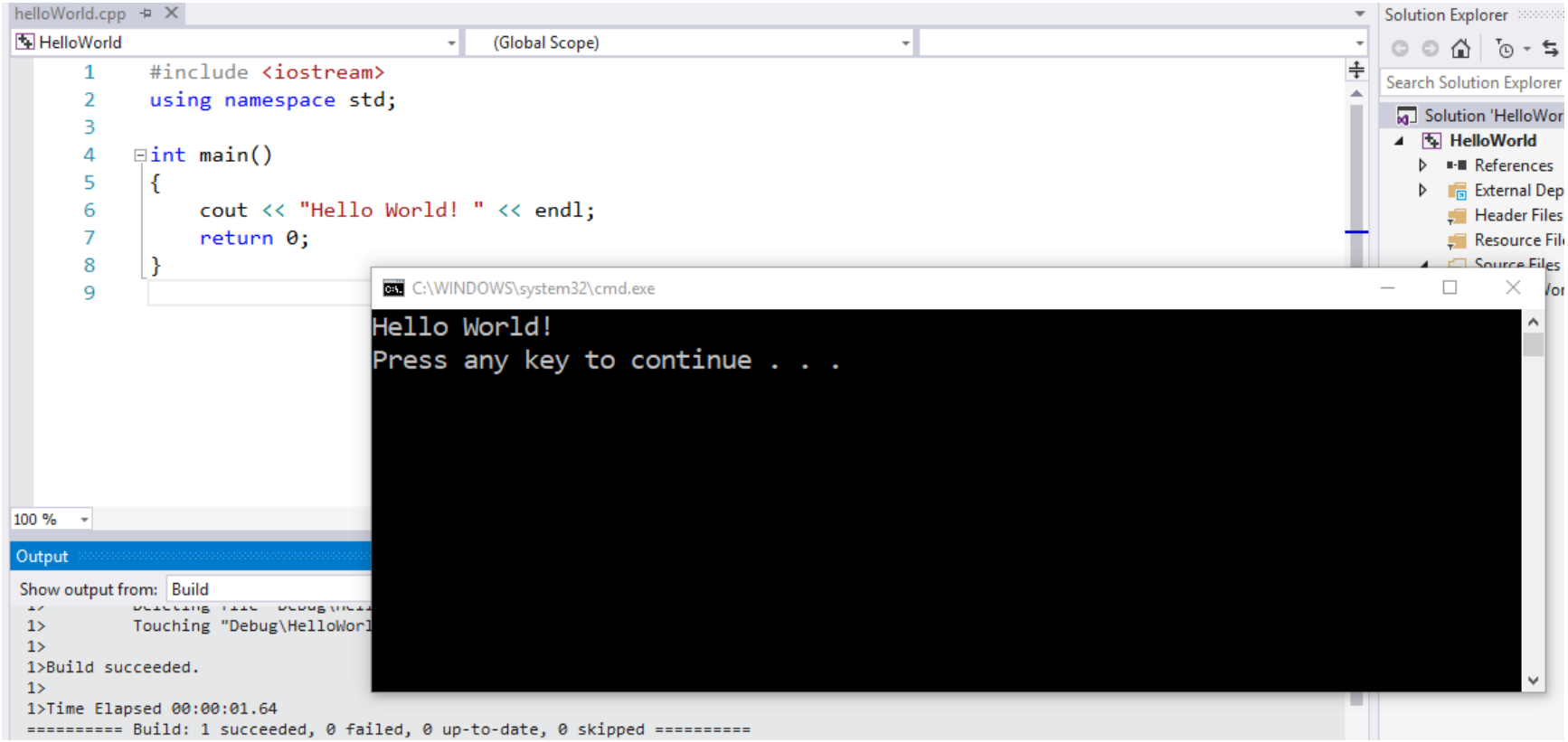
Select C++ File, give it a name and click add



Write your code in the .cpp file and click on the green triangle to run your program



This window will appear



Chapter 2:
Basic Elements of C++

Outlines

- In this chapter, you will study:
  - A Quick Look at a C++ Program
  - The Basics of a C++ Program (comments, Special Symbols, Keywords and identifiers)
  - Data Types
  - Data Types and Variables
  - Arithmetic Operators, Operator Precedence, and Expressions
  - Type Conversion (Casting)
  - string Type
  - Variables, Assignment Statements, and Input Statements
  - Increment and Decrement Operators
  - Output statements
  - Preprocessor Directives
  - Creating a C++ Program
  - Debugging: Understanding and Fixing Syntax Errors
  - Program Style and Form

Introduction

- Computer program
  - Sequence of statements whose objective is to accomplish a task
- Programming
  - Process of planning and creating a program
- Real-world analogy: a recipe for cooking

First C++ Program

```
1  //First C++ program
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      cout<<"Welcome to Computer Programming Course";
8      return 0;
9  }
```

Comments

- Comments are for the reader, not the compiler
- Two types:
  - Single line: begin with //

```
// This is a C++ program.
// Welcome to C++ Programming.
```

  - Multiple line: enclosed between /* and */

```
/*
   You can include comments that can
   occupy several lines.
*/
```

Preprocessor Directives

- C++ has a small number of operations
- Many functions and symbols needed to run a C++ program are provided as collection of libraries
- Every library has a name and is referred to by a header file
- Preprocessor directives are commands supplied to the preprocessor program
- All preprocessor commands begin with #
- No semicolon at the end of these commands

Preprocessor Directives (cont'd.)

- Syntax to include a header file:

```
#include <headerFileName>
```

- For example:

```
#include <iostream>
```

  - Causes the preprocessor to include the header file `iostream` in the program

- Preprocessor commands are processed before the program goes through the compiler

`namespace` and Using `cin` and `cout` in a Program

- `cin` and `cout` are declared in the header file `iostream`, but within `std` namespace
- To use `cin` and `cout` in a program, use the following two statements:

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

Main Function

- A C++ program is a collection of functions, one of which is the function `main`
- The first line of the function `main` is called the heading of the function:
  - `int main()`
- The statements enclosed between the curly braces { and } form the body of the function
- The program execution starts from the `main` function

Output

- The syntax of `cout` and << is:

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

  - Called an output statement
- The stream insertion operator is <<
- Expression evaluated and its value is printed at the current cursor position on the screen

Output (cont'd.)

- A manipulator is used to format the output
  - Example: `endl` causes insertion point to move to beginning of next line

EXAMPLE 2-21

Consider the following statements. The output is shown to the right of each statement.

	Statement	Output
1	`cout << 29 / 4 << endl;`	`7`
2	`cout << "Hello there." << endl;`	`Hello there.`
3	`cout << 12 << endl;`	`12`
4	`cout << "4 + 7" << endl;`	`4 + 7`
5	`cout << 4 + 7 << endl;`	`11`
6	`cout << 'A' << endl;`	`A`
7	`cout << "4 + 7 = " << 4 + 7 << endl;`	`4 + 7 = 11`
8	`cout << 2 + 3 * 5 << endl;`	`17`
9	`cout << "Hello \nthere." << endl;`	`Hello` `there.`

Output (cont'd.)

- The new line character is '\n'
  - May appear anywhere in the string

```
 cout << "Hello there.";
 cout << "My name is James.";
```

Output:

```
 Hello there.My name is James.
```

```
cout << "Hello there.\n";
 cout << "My name is James.";
```

Output :

```
 Hello there.
 My name is James.
```

Output (cont'd.)

TABLE 2-4 Commonly Used Escape Sequences

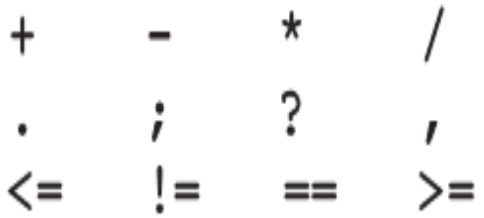
	Escape Sequence	Description
`\n`	Newline	Cursor moves to the beginning of the next line
`\t`	Tab	Cursor moves to the next tab stop
`\b`	Backspace	Cursor moves one space to the left
`\r`	Return	Cursor moves to the beginning of the current line (not the next line)
`\\`	Backslash	Backslash is printed
`\'`	Single quotation	Single quotation mark is printed
`\"`	Double quotation	Double quotation mark is printed

Output(cont'd) - Example

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      cout<<"abc\ndef"<<endl;
7      cout<<"abc\tdef"<<endl; //abc      def
8      cout<<"abc\bdef"<<endl; //abdef
9      cout<<"abc\rdef"<<endl; //def
10     cout<<"abc\\def"<<endl; //abc\def
11     cout<<"abc\'def"<<endl; //abc'def
12     cout<<"abc\"def"<<endl; //abc"def
13     return 0;
14 }
```

Special Symbols

- Token: the smallest individual unit of a program written in any language
- C++ tokens include special symbols, Keywords, and identifiers.
- Special symbols in C++ include:
  - Punctuators(e.g. [] () {} , ; : * #).
  - Operators(arithmetical operators, Relational operators, Logical operators, Unary operators, Assignment operators, Conditional operators, Comma operator).



Reserved Words (Keywords)

- Reserved word symbols (or keywords):
  - Cannot be redefined within program
  - Cannot be used for anything other than their intended use

  Examples:
    - `int`
    - `float`
    - `double`
    - `char`
    - `const`
    - `void`
    - `return`

Whitespaces

- Every C++ program contains whitespaces
  - Include blanks, tabs, and newline characters
- Used to separate special symbols, reserved words, and identifiers
- Proper utilization of whitespaces is important
  - Can be used to make the program more readable

Identifiers

- Identifier: the name of something [such as variables, type, template, class ,or function] that appears in a program[]
  - Consists of letters, digits, and the underscore character (_)
  - Must begin with a letter or underscore
- C++ is case sensitive
  - NUMBER is not the same as number
- Two predefined identifiers are cout and cin
- Unlike reserved words, predefined identifiers may be redefined, but it is not a good idea

Identifiers (cont'd.)

- Identifier restrictions:
  - Do not use C++ keywords.
  - Never start your identifier with a digit (number) always start it with alphabet or underscore.
  - Do not use white spaces, use underscores instead.
  - Do not use special symbols such as #, $,+,=,-,! etc.
- Legal identifiers in C++: `first, conversion ,payrate, counter1`

TABLE 2-1 Examples of Illegal Identifiers

Illegal Identifier	Description
`employee Salary`	There can be no space between `employee` and `Salary`.
`Hello!`	The exclamation mark cannot be used in an identifier.
`one+two`	The symbol + cannot be used in an identifier.
`2nd`	An identifier cannot begin with a digit.

Data Types

- Data type: set of values together with a set of operations
- C++ data types fall into three categories:
  - Simple data type
  - Structured data type
  - Pointers

Simple Data Types

- Three categories of simple data
  - Integral: integers (numbers without a decimal)
    - Can be further categorized:
      - `char, short, int, long, bool, unsigned char, unsigned short, unsigned int, unsigned long`
  - Floating-point: decimal numbers
  - Enumeration type: user-defined data type

Simple Data Types (cont'd.)

TABLE 2-2 Values and Memory Allocation for Three Simple Data Types

Data Type	Values	Storage (in bytes)
`int`	-2147483648 to 2147483647	4
`bool`	`true` and `false`	1
`char`	-128 to 127	1

- Different compilers may allow different ranges of values

`int` Data Type

- Examples:

```
-6728
0
78
+763
```

- Cannot use a comma within an integer
  - Commas are only used for separating items in a list

bool Data Type

- bool type
  - Two values: true and false
  - Manipulate logical (Boolean) expressions
- true and false
  - Logical values
- bool, true, and false
  - Reserved word
  - Any none zero value is considered as true.
    - bool x = -5; // x is true
    - bool y = 10; // y is true
    - bool w = 0; // w is false

`char` Data Type

- The smallest integral data type
- Used for single characters: letters, digits, and special symbols
- Each character is enclosed in single quotes
  - `'A', 'a', '0', '*', '+', '$', '&'`
- A blank space is a character
  - Written `' '`, with a space left between the single quotes

`char` Data Type (cont'd.)

- Different character data sets exist
- ASCII: American Standard Code for Information Interchange
  - Each of 128 values in ASCII code set represents a different character
  - Characters have a predefined ordering based on the ASCII numeric value
- Collating sequence: ordering of characters based on the character set code

ASCII Table

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

Floating-Point Data Types

- C++ uses scientific notation to represent real numbers (floating-point notation)

TABLE 2-3 Examples of Decimal Numbers in Scientific and C++ Floating-Point Notations

Decimal Number	Scientific Notation	C++ Floating-Point Notation
75.924	$7.5924 * 10^{1}$	7.592400E1
0.18	$1.8 * 10^{-1}$	1.800000E-1
0.0000453	$4.53 * 10^{-5}$	4.530000E-5
-1.482	$-1.482 * 10^{0}$	-1.482000E0
7800.0	$7.8 * 10^{3}$	7.800000E3

Floating-Point Data Types (cont'd.)

- `float`: represents any real number
  - Range: -3.4E+38 to 3.4E+38 (four bytes)
- `double`: represents any real number
  - Range: -1.7E+308 to 1.7E+308 (eight bytes)
- Minimum and maximum values of data types are system dependent

Floating-Point Data Types (cont'd.)

- Maximum number of significant digits (decimal places) for `float` values: 6 or 7
- Maximum number of significant digits for `double`: 15
- Precision: maximum number of significant digits
  - Float values are called single precision
  - Double values are called double precision

Variables

- Variable: memory location whose content may change during execution
- Data must be loaded into main memory before it can be manipulated
- Storing data in memory is a two-step process:
  - Instruct computer to allocate memory (define a variable)
  - Include statements to put data into memory (set its value)

Variables (cont'd.)

- To declare a variable, must specify the data type it will store
  - determines the size and layout of the variable's memory
  - The range of values that can be stored within that memory
  - The set of operations that can be applied to the variable.
- Syntax to declare a variable:

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

EXAMPLE 2-12

Consider the following statements:

```
double amountDue;
int counter;
char ch;
int x, y;
string name;
```

Putting Data into Variables

- Ways to place data into a variable:
  - Use C++'s assignment statement
  - Use input (read) statements

Assignment Statement

- The assignment statement takes the form:

```
variable = expression;
```

- Expression is evaluated and its value is assigned to the variable on the left side
- A variable is said to be initialized the first time a value is placed into it
- In C++, = is called the assignment operator

Assignment Statement (cont'd.)

EXAMPLE 2-13

Suppose you have the following variable declarations:

```
int num1, num2;
double sale;
char first;
string str;
```

Now consider the following assignment statements:

```
num1 = 4;
num2 = 4 * 5 - 11;
sale = 0.02 * 1000;
first = 'D';
str = "It is a sunny day.";
```

Assignment Statement (cont'd.)

EXAMPLE 2-14

Suppose that num1, num2, and num3 are `int` variables and the following statements are executed in sequence.

```
1. num1 = 18;
2. num1 = num1 + 27;
3. num2 = num1;
4. num3 = num2 / 5;
5. num3 = num3 / 4;
```

	Values of the Variables			Explanation
Before Statement 1	? num1	? num2	? num3	
After Statement 1	18 num1	? num2	? num3	
After Statement 2	45 num1	? num2	? num3	`num1 + 27 = 18 + 27 = 45`. This value is assigned to `num1`, which replaces the old value of `num1`.
After Statement 3	45 num1	45 num2	? num3	Copy the value of `num1` into `num2`.
After Statement 4	45 num1	45 num2	9 num3	`num2 / 5 = 45 / 5 = 9`. This value is assigned to `num3`. So `num3 = 9`.
After Statement 5	45 num1	45 num2	2 num3	`num3 / 4 = 9 / 4 = 2`. This value is assigned to `num3`, which replaces the old value of `num3`.

Declaring & Initializing Variables

- Not all types of variables are initialized automatically
- Variables can be initialized when declared:

```
int first=13, second=10;
char ch=' ';
double x=12.6;
```

- All variables must be initialized before they are used
  - But not necessarily during declaration

Allocating Memory with Constants and Variables

- Named constant: memory location whose content can't change during execution
- Syntax to declare a named constant:

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

- In C++, **`const`** is a reserved word

EXAMPLE 2-11

Consider the following C++ statements:

```
const double CONVERSION = 2.54;
const int NO_OF_STUDENTS = 20;
const char BLANK = ' ';
```

A C++ Program (cont'd.)

```
//*************************************************************
// Given the length and width of a rectangle, this C++ program
// computes and outputs the perimeter and area of the rectangle.
//*************************************************************

#include <iostream>

using namespace std;

int main()
{
    double length;
    double width;
    double area;
    double perimeter;

    cout << "Program to compute and output the perimeter and "
         << "area of a rectangle." << endl;

    length = 6.0;
```

Comments

Variable declarations. A statement such as `double length;` instructs the system to allocate memory space and name it `length`.

Assignment statement. This statement instructs the system to store `6.0` in the memory space `length`.

A C++ Program (cont'd.)

```
    width = 4.0;
    perimeter = 2 * (length + width);

    area = length * width;

    cout << "Length = " << length << endl;
    cout << "Width =  " << width << endl;
    cout << "Perimeter = " << perimeter << endl;
    cout << "Area = " << area << endl;

    return 0;
}
```

Assignment statement. This statement instructs the system to evaluate the expression `length * width` and store the result in the memory space `area`.

Output statements. An output statement instructs the system to display results.

A C++ Program (cont'd.)

- Sample run:

```
Program to compute and output the perimeter and area of a rectangle.
Length = 6
Width =  4
Perimeter = 20
Area = 24
```

Input (Read) Statement

- `cin` is used with `>>` to gather input

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

- This is called an input (read) statement
- The stream extraction operator is >>
- For example, if miles is a double variable

```
cin >> miles;
```

  - Causes computer to get a value of type `double` and places it in the variable `miles`

Input (Read) Statement (cont'd.)

- Using more than one variable in `cin` allows more than one value to be read at a time
- Example: if `feet` and `inches` are variables of type `int`, this statement:

```
cin >> feet >> inches;
```

  - Inputs two integers from the keyboard
  - Places them in variables `feet` and `inches` respectively

Example 2- 18

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string firstName;                                   //Line 1
    string lastName;                                    //Line 2
    int age;                                            //Line 3
    double weight;                                      //Line 4

    cout << "Enter first name, last name, age, "
         << "and weight, separated by spaces."
         << endl;                                       //Line 5

    cin >> firstName >> lastName;                       //Line 6
    cin >> age >> weight;                               //Line 7

    cout << "Name: " << firstName << " "
         << lastName << endl;                           //Line 8

    cout << "Age: " << age << endl;                     //Line 9
    cout << "Weight: " << weight << endl;               //Line 10

    return 0;                                           //Line 11
}
```

Arithmetic Operators, Operator Precedence, and Expressions

- C++ arithmetic operators:
  - + addition
  - - subtraction
  - * multiplication
  - / division
  - % modulus (or remainder) operator
- +, -, *, and / can be used with integral and floating-point data types
- Use % only with integral data types

Arithmetic Operators, Operator Precedence, and Expressions (cont'd.)

EXAMPLE 2-3

Arithmetic Expression	Result	Description
`5 / 2`	2	In the division `5 / 2`, the quotient is 2 and the remainder is 1. Therefore, `5 / 2` with the integral operands evaluates to the quotient, which is 2.
`14 / 7`	2	In the division `14 / 7`, the quotient is 2.
`34 % 5`	4	In the division `34 / 5`, the quotient is 6 and the remainder is 4. Therefore, `34 % 5` evaluates to the remainder, which is 4.
`4 % 6`	4	In the division `4 / 6`, the quotient is 0 and the remainder is 4. Therefore, `4 % 6` evaluates to the remainder, which is 4.

Arithmetic Operators, Operator Precedence, and Expressions (cont'd.)

EXAMPLE 2-4

Given length in inches, we write a program that determines and outputs the equivalent length in feet and (remaining) inches. Now there are 12 inches in a foot. Therefore, 100 inches equals 8 feet and 4 inches; similarly, 55 inches equals 4 feet and 7 inches. Note that 100 / 12 = 8 and 100 % 12 = 4; similarly, 55 / 12 = 4 and 55 % 12 = 7. From these examples, it follows that we can effectively use the operators / and % to accomplish our task. The desired program is as follows:

```
// Given length in inches, this program outputs the equivalent
// length in feet and remaining inch(es).

#include <iostream>

using namespace std;

int main()
{
    int inches; //variable to store total inches

    inches = 100;  //store 100 in the variable inches

    cout << inches << " inch(es) = ";  //output the value of
                                       //inches and the equal sign
    cout << inches / 12 << " feet (foot) and "; //output maximum
                                       //number of feet (foot)

    cout << inches % 12 << " inch(es)" << endl; //output
                                       //remaining inches

    return 0;
}
```

Arithmetic Operators, Operator Precedence, and Expressions (cont'd.)

- When you use / with integral data types, the integral result is truncated (no rounding).(5/2 = 2)
- When you use / with floating-point data types returns a floating point value[i.e. the fraction is kept] For example, 5.0 / 2 = 2.5, 5 / 2.0 = 2.5, and 5.0 / 2.0 = 2.5.
- Arithmetic expressions: contain values and arithmetic operators
- Operands: the number of values on which the operators will work
- Operators can be unary (one operand) or binary (two operands)

Order of Precedence

- All operations inside of () are evaluated first
- *, /, and % are at the same level of precedence and are evaluated next
- + and – have the same level of precedence and are evaluated last
- When operators are on the same level
  - Performed from left to right (associativity)
- `3 * 7 – 6 + 2 * 5 / 4 + 6` means

```
(((3 * 7) – 6) + ((2 * 5) / 4 )) + 6
```

Expressions

- Integral expression: all operands are integers
  - Yields an integral result
  - Example: `2 + 3 * 5`
- Floating-point expression: all operands are floating-point
  - Yields a floating-point result
  - Example: `12.8 * 17.5 - 34.50`

Mixed Expressions

- Mixed expression:
  - Has operands of different data types
  - Contains integers and floating-point
- Examples of mixed expressions:

```
2 + 3.5
6  /  4 + 3.9
5.4  *  2 - 13.6 + 18  /  2
```

13.0 / 2 + 1

- Remember that % (modulus which finds the remainder) is applied for integer values only. So, 9%4 = 1, but 9%2.5 → Syntax Error.

Mixed Expressions (cont'd.)

- Evaluation rules:
  - If operator has same types of operands
    - Evaluated according to the type of the operands
  - If operator has both types of operands
    - Integer is changed to floating-point
    - Operator is evaluated
    - Result is floating-point
  - Entire expression is evaluated according to precedence rules

Saving and Using the Value of an Expression

- To save the value of an expression:
  - Declare a variable of the appropriate data type
  - Assign the value of the expression to the variable that was declared
    - Use the assignment statement
- Wherever the value of the expression is needed, use the variable holding the value

Saving and Using the Value of an Expression (cont'd)

EXAMPLE 2-15

Suppose that you have the following declaration:

```
int a, b, c, d;
int x, y;
```

Further suppose that you want to evaluate the expressions $-b + (b^2 - 4ac)$ and $-b - (b^2 - 4ac)$ and assign the values of these expressions to x and y, respectively. Because the expression $b^2 - 4ac$ appears in both expressions, you can first calculate the value of this expression and save its value in d. You can then use the value of d to evaluate the expressions, as shown by the following statements:

```
d = b * b - 4 * a * c;
x = -b + d;
y = -b - d;
```

Type Conversion (Casting)

- Implicit type conversion: when value of one type is automatically changed to another type temporarily [done by the compiler]
- Examples:

```
bool value1 = 10;  // the compiler will
  implicitly convert 10 to true
int value2 =-13.7; // the compiler will
  implicitly convert -13.7 into -13.
```

- Cast operator: provides explicit type conversion [coded explicitly by the programmer]

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

Type Conversion (cont'd.)

EXAMPLE 2-9

Expression	Evaluates to
`static_cast<int>(7.9)`	`7`
`static_cast<int>(3.3)`	`3`
`static_cast<double>(25)`	`25.0`
`static_cast<double>(5+3)`	`=static_cast<double>(8)=8.0`
`static_cast<double>(15)/2`	`=15.0/2` (because `static_cast<double>(15)=15.0`) `=15.0/2.0=7.5`
`static_cast<double>(15/2)`	`=static_cast<double>(7)` (because `15/2=7`) `= 7.0`
`static_cast<int>(7.8 + static_cast<double>(15)/2)`	`=static_cast<int>(7.8+7.5)` `=static_cast<int>(15.3)` `= 15`
`static_cast<int>(7.8 + static_cast<double>(15/2))`	`=static_cast<int>(7.8 + 7.0)` `=static_cast<int>(14.8)` `= 14`

Increment and Decrement Operators

- Increment operator: increase variable by 1
  - Pre-increment: `++variable`
  - Post-increment: `variable++`
- Decrement operator: decrease variable by 1
  - Pre-decrement: `--variable`
  - Post-decrement: `variable–`
- What is the difference between the following?

```
x = 5;
y = ++x;
```

```
x = 5;
y = x++;
```

Increment and Decrement Operators Example 2-20

Suppose a and b are int variables and

```
a = 5;
b = 2 + (++a);
```

The first statement assigns 5 to a. To execute the second statement, first the expression 2 + (++a) is evaluated. Because the pre-increment operator is applied to a, first the value of a is incremented to 6. Then 2 is added to 6 to get 8, which is then assigned to b. Therefore, after the second statement executes, a is 6 and b is 8.

On the other hand, after the execution of the following statements:

```
a = 5;
b = 2 + (a++);
```

the value of a is 6 while the value of b is 7.

`string` Type

- Programmer-defined type supplied in ANSI/ISO Standard C++ library
- Sequence of zero or more characters enclosed in double quotation marks
- Null (or empty): a string with no characters
- Each character has a relative position in the string
  - Position of first character is 0
- Length of a string is number of characters in it
  - Example: length of "`William Jacob`" is 13
  - Position of character 'W' is 0
  - Position of character 'J' is 8

Using the `string` Data Type in a Program

- To use the `string` type, you need to access its definition from the header file `string`
- Include the following preprocessor directive:

```
#include  <string>
```

Input the `string` Type

- An input stream variable (`cin`) and `>>` operator can read a string into a variable of the data type `string`
- **Extraction operator**
  - Skips any leading whitespace characters
  - Reading stops at a whitespace character
- The function **`getline`**
  - Reads until end of the current line

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

- How To print the content of a string variable?

Input the `string` Type (Cont'd)

```
string name;
cin >> name;     //ahmad ali
//the value stored in name is ahmad only


string name;
getline(cin,name);     //ahmad ali
//the value stored in name is ahmad ali
```

Output the `string` Type

- Example:

```
cout << name;
```

  - Outputs the content of name on the screen
  - << continues to write the contents of name until it finds the null character
  - If name does not contain the null character, then strange output may occur
    - << continues to output data from memory adjacent to name until a '\0' is found

Creating a C++ Program

- C++ program has two parts:
  - Preprocessor directives
  - The program
- Preprocessor directives and program statements constitute C++ source code (.cpp)
- Compiler generates object code (.obj)
- Executable code is produced and saved in a file with the file extension .exe

Creating a C++ Program (cont'd.)

- A C++ program contains two types of statements:
  - Declaration statements: declare things, such as variables
  - Executable statements: perform calculations, manipulate data, create output, accept input, etc.

The Basics of a C++ Program

- Function (or subprogram): collection of statements; when executed, accomplishes something
  - May be predefined or standard
- Syntax rules: rules that specify which statements (instructions) are legal or valid
- Semantic rules: determine the meaning of the instructions Programming language: a set of rules, symbols, and special words

Debugging: Understanding and Fixing Syntax Errors

- Compile a program
  - Compiler will identify the syntax errors
  - Specifies the line numbers where the errors occur

```
Example2_Syntax_Errors.cpp
c:\chapter 2 source
  code\example2_syntax_errors.cpp(9) : error
  C2146: syntax error :
missing ';' before identifier 'num'
c:\chapter 2 source
  code\example2_syntax_errors.cpp(11) : error
  C2065: 'tempNum' :
undeclared identifier
```

Syntax

- Syntax rules: indicate what is legal and what is not legal
- Errors in syntax are found in compilation

```
int x;          //Line 1
int y           //Line 2: error
double z;       //Line 3

y = w + x;      //Line 4: error
```

Use of Blanks

- In C++, you use one or more blanks to separate numbers when data is input
- Blanks are also used to separate reserved words and identifiers from each other and from other symbols
- Blanks must never appear within a reserved word or identifier

Use of Semicolons, Brackets, and Commas

- All C++ statements end with a semicolon
  - Also called a statement terminator
- { and } are not C++ statements
  - Can be regarded as delimiters
- Commas separate items in a list

Semantics

- Semantics: set of rules that gives meaning to a language
  - Possible to remove all syntax errors in a program and still not have it run
  - Even if it runs, it may still not do what you meant it to do
- Ex: `2 + 3 * 5` and `(2 + 3) * 5`

  are both syntactically correct expressions, but have different meanings

Naming Identifiers

- Identifiers can be self-documenting:
  - `CENTIMETERS_PER_INCH`
- Avoid run-together words :
  - `annualsale`
  - Solution:
    - Capitalizing the beginning of each new word: `annualSale`
    - Inserting an underscore just before a new word: `annual_sale`

Prompt Lines

- Prompt lines: executable statements that inform the user what to do

```
cout << "Please enter a number between 1 and 10 and "
     << "press the return key" << endl;
cin >> num;
```

- Always include prompt lines when input is needed from users

Documentation

- A well-documented program is easier to understand and modify
- You use comments to document programs
- Comments should appear in a program to:
  - Explain the purpose of the program
  - Identify who wrote it
  - Explain the purpose of particular statements

Form and Style

- Consider two ways of declaring variables:
  - Method 1

```
int feet, inch;
double x, y;
```

  - Method 2

```
int feet,inch;double x,y;
```

- Both are correct; however, the second is hard to read

Chapter 3:
Input/ Output

Outline

- In this chapter, you will study:
  - I/O Streams and Standard I/O Devices
  - Input Failure
  - Using Predefined Functions in a Program

I/O Streams and Standard I/O Devices

- I/O: sequence of bytes (stream of bytes) from source to destination
  - Bytes are usually characters, unless program requires other types of information
  - Stream: sequence of characters from source to destination
  - Input stream: sequence of characters from an input device to the computer
  - Output stream: sequence of characters from the computer to an output device

I/O Streams and Standard I/O Devices (cont'd.)

- Use `iostream` header file to receive data from keyboard and send output to the screen
  - Contains definitions of two data types:
    - `istream`: input stream
    - `ostream`: output stream
  - Has two variables:
    - `cin`: stands for common input
    - `cout`: stands for common output

I/O Streams and Standard I/O Devices (cont'd.)

- Variable declaration is similar to:
  - `istream cin;`
  - `ostream cout;`
- To use `cin` and `cout`, the preprocessor directive `#include <iostream>` must be used
- Input stream variables: type `istream`
- Output stream variables: type `ostream`

`cin` and the Extraction Operator >>

- The syntax of an input statement using `cin` and the extraction operator >> is:

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

- The extraction operator >> is binary
  - Left-side operand is an input stream variable
    - Example: `cin`
  - Right-side operand is a variable

`cin` and the Extraction Operator >> (cont'd.)

- No difference between a single `cin` with multiple variables and multiple `cin` statements with one variable
- When scanning, `>>` skips all whitespace
  - Blanks and certain nonprintable characters
- `>>` distinguishes between character `2` and number `2` by the right-side operand of `>>`
  - If type `char` or `int` (or `double`), the `2` is treated as a character or as a number `2`

`cin` and the Extraction Operator >> (cont'd.)

TABLE 3-1 Valid Input for a Variable of the Simple Data Type

Data Type of `a`	Valid Input for `a`
`char`	One printable character except the blank
`int`	An integer, possibly preceded by a + or – sign
`double`	A decimal number, possibly preceded by a + or – sign. If the actual data input is an integer, the input is converted to a decimal number with the zero decimal part.

- Entering a `char` value into an `int` or `double` variable causes serious errors, called input failure

`cin` and the Extraction Operator >> (cont'd.)

- When reading data into a `char` variable
  - `>>` skips leading whitespace, finds and stores only the next character
  - Reading stops after a single character
- To read data into an `int` or `double` variable
  - `>>` skips leading whitespace, reads + or - sign (if any), reads the digits (including decimal)
  - Reading stops on whitespace non-digit character

`cin` and the Extraction Operator >> (cont'd.)

EXAMPLE 3-1

Suppose you have the following variable declarations:

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

The following statements show how the extraction operator >> works.

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

`cin` and the Extraction Operator >> (cont'd.)

EXAMPLE 3-2

Suppose you have the following variable declarations:

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

The following statements show how the extraction operator >> works.

	Statement	Input	Value Stored in Memory
1	`cin >> a >> ch >> z;`	`57 A 26.9`	`a = 57, ch = 'A', z = 26.9`
2	`cin >> a >> ch >> z;`	`57  A` `26.9`	`a = 57, ch = 'A', z = 26.9`

3	`cin >> a >> ch >> z;`	`57`
		`A`
		`26.9`
			`a = 57, ch = 'A', z = 26.9`

4	`cin >> a >> ch >> z;`	`57A26.9`	`a = 57, ch = 'A', z = 26.9`

`cin` and the Extraction Operator >> (cont'd.)

EXAMPLE 3-3

Suppose you have the following variable declarations:

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

The following statements show how the extraction operator >> works.

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

Input Failure

- Things can go wrong during execution
- If input data does not match corresponding variables, program may run into problems
- Trying to read a letter into an `int` or `double` variable will result in an input failure
- If an error occurs when reading data
  - Input stream enters the fail state

Input Failure (cont'd)

EXAMPLE 3-8

```
//Input Failure program

#include <iostream>
#include <string>

using namespace std;

int main()
{
    string name;                                          //Line 1
    int age = 0;                                          //Line 2
    int weight = 0;                                       //Line 3
    double height = 0.0;                                  //Line 4


    cout << "Line 5: Enter name, age, weight, and "
         << "height: ";                                   //Line 5
    cin >> name >> age >> weight >> height;               //Line 6
    cout << endl;                                         //Line 7

    cout << "Line 8: Name: " << name << endl;             //Line 8
    cout << "Line 9: Age: " << age << endl;               //Line 9
    cout << "Line 10: Weight: " << weight << endl;        //Line 10
    cout << "Line 11: Height: " << height << endl;        //Line 11

    return 0;                                             //Line 12
}
```

Input Failure (cont'd)

Sample Run 1

```
Line 5: Enter name, age, weight, and height: Sam 35 q56 6.2

Line 8: Name: Sam
Line 9: Age: 35
Line 10: Weight: 0
Line 11: Height: 0
```

Sample Run 2

```
Line 5: Enter name, age, weight, and height: Sam 35.0 156 6.2

Line 8: Name: Sam
Line 9: Age: 35
Line 10: Weight: 0
Line 11: Height: 0
```

Using Predefined Functions in a Program

- Function (subprogram): set of instructions
  - When activated, it accomplishes a task
- `main` executes when a program is run
- Other functions execute only when called
- C++ includes a wealth of functions
  - Predefined functions are organized as a collection of libraries called header files

Using Predefined Functions in a Program (cont'd.)

- Header file may contain several functions
- To use a predefined function, you need the name of the appropriate header file
  - You also need to know:
    - Function name
    - Number of parameters required
    - Type of each parameter
    - What the function is going to do

Using Predefined Functions in a Program (cont'd.)

- To use `pow` (power), include `cmath`
  - Two numeric parameters
  - Syntax: `pow(x,y)` = x^y
    - `x` and `y` are the arguments or parameters
  - In `pow(2,3)`, the parameters are `2` and `3`

Using Predefined Functions in a Program (cont'd.)

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

const double PI = 3.1416;

int main()
{
    double sphereRadius;                                    //Line 1
    double sphereVolume;                                    //Line 2
    double point1X, point1Y;                                //Line 3
    double point2X, point2Y;                                //Line 4
    double distance;                                        //Line 5

    string str;                                             //Line 6

    cout << "Line 7: Enter the radius of the sphere: ";     //Line 7
    cin >> sphereRadius;                                    //Line 8
    cout << endl;                                           //Line 9

    sphereVolume = (4 / 3) * PI * pow(sphereRadius, 3);     //Line 10

    cout << "Line 11: The volume of the sphere is: "
         << sphereVolume << endl << endl;                   //Line 11

    cout << "Line 12: Enter the coordinates of two "
         << "points in the X-Y plane: ";                    //Line 12
    cin >> point1X >> point1Y >> point2X >> point2Y;        //Line 13
    cout << endl;                                           //Line 14
```

Using Predefined Functions in a Program (cont'd.)

```
    distance = sqrt(pow(point2X - point1X, 2)
                    + pow(point2Y - point1Y, 2));          //Line 15

    cout << "Line 16: The distance between the points "
         << "(" << point1X << ", " << point1Y << ") and "
         << "(" << point2X << ", " << point2Y << ") is: "
         << distance << endl << endl;                      //Line 16

    str = "Programming with C++";                          //Line 17

    cout << "Line 18: The number of characters, "
         << "including blanks, in \"" << str << "\" is: "
         << str.length() << endl;                          //Line 18

    return 0;                                              //Line 19
}
```

Chapter 4:
Control Structures I (Selection)

Outline

- In this chapter, you will study :
  - Control Structures
  - Relational Operators
  - Relational Operators and the `string` Type
  - Logical (Boolean) Operators and Logical Expressions
  - Selection: `if` and `if...else`
  - `switch` Structures

Control Structures

- A computer can proceed:
  - In sequence
  - Selectively (branch): making a choice
  - Repetitively (iteratively): looping
  - By calling a function
- Two most common control structures:
  - Selection
  - Repetition

Control Structures (cont'd.)

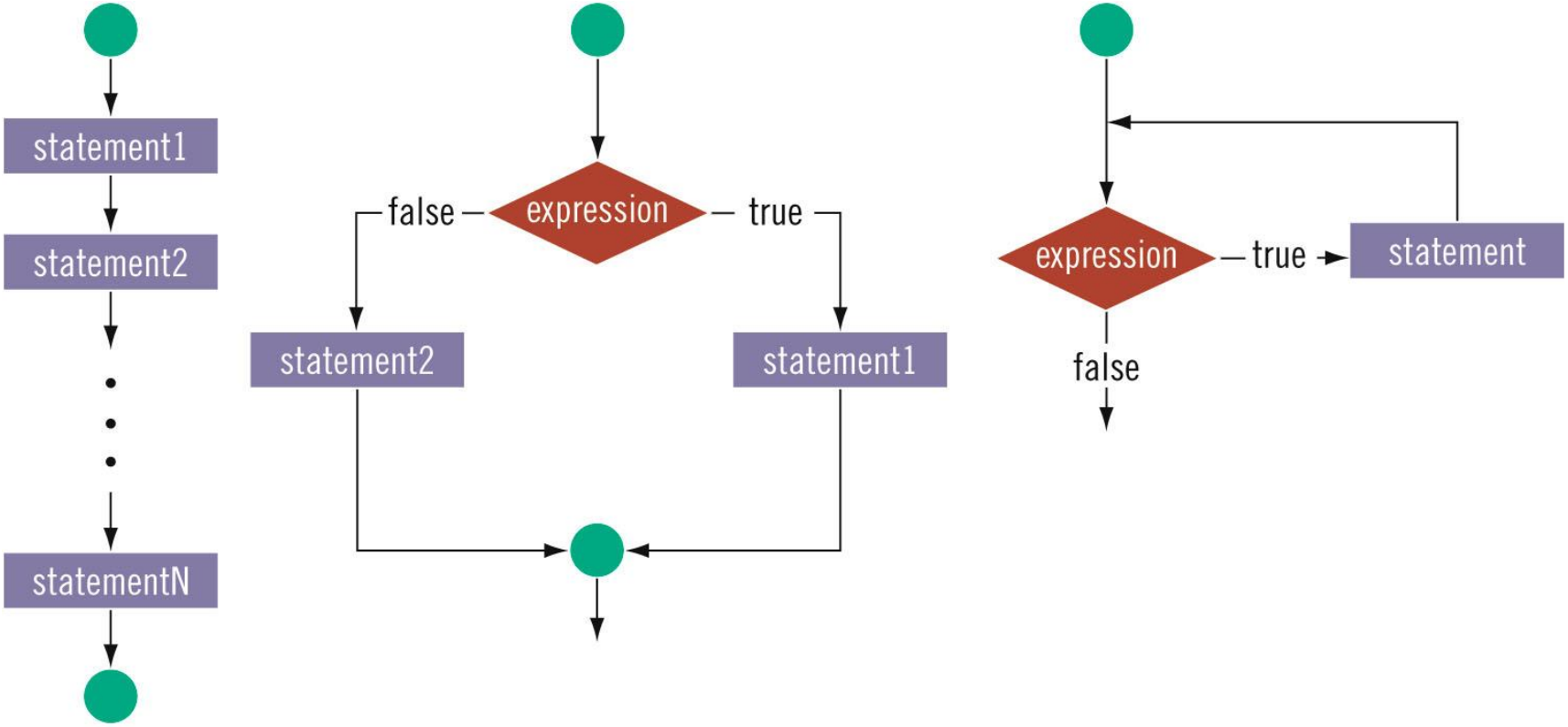


FIGURE 4-1 Flow of execution

Relational Operators

- Conditional statements: only executed if certain conditions are met
- Condition: represented by a logical (Boolean) expression that evaluates to a logical (Boolean) value of `true` or `false`
- Relational operators:
  - Allow comparisons
  - Require two operands (binary)
  - Evaluate to `true` or `false`

Relational Operators (cont'd.)

TABLE 4-1 Relational Operators in C++

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

Relational Operators and Simple Data Types

- Relational operators can be used with all three simple data types:

EXAMPLE 4-1

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

Comparing Characters

- Expression of `char` values with relational operators
  - Result depends on machine's collating sequence
  - ASCII character set
- Logical (Boolean) expressions
  - Expressions such as `4 < 6` and `'R' > 'T'`
  - Returns an integer value of `1` if the logical expression evaluates to `true`
  - Returns an integer value of `0` otherwise

Relational Operators and the `string` Type

- Relational operators can be applied to strings
  - Strings are compared character by character, starting with the first character
  - Comparison continues until either a mismatch is found or all characters are found equal
  - If two strings of different lengths are compared and the comparison is equal to the last character of the shorter string
    - The shorter string is less than the larger string

Relational Operators and the `string` Type (cont'd.)

- Suppose we have the following declarations:

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

Relational Operators and the `string` Type (cont'd.)

Expression	Value /Explanation
`str1 < str2`	`true` `str1 = "Hello"` and `str2 = "Hi"`. The first characters of `str1` and `str2` are the same, but the second character `'e'` of `str1` is less than the second character `'i'` of `str2`. Therefore, `str1 < str2` is `true`.
`str1 > "Hen"`	`false` `str1 = "Hello"`. The first two characters of `str1` and `"Hen"` are the same, but the third character `'l'` of `str1` is less than the third character `'n'` of `"Hen"`. Therefore, `str1 > "Hen"` is `false`.
`str3 < "An"`	`true` `str3 = "Air"`. The first characters of `str3` and `"An"` are the same, but the second character `'i'` of `"Air"` is less than the second character `'n'` of `"An"`. Therefore, `str3 < "An"` is `true`.

Relational Operators and the `string` Type (cont'd.)

`str1 == "hello"`	**false** `str1 = "Hello"`. The first character `'H'` of `str1` is less than the first character `'h'` of `"hello"` because the ASCII value of `'H'` is 72, and the ASCII value of `'h'` is 104. Therefore, `str1 == "hello"` is **false**.
`str3 <= str4`	**true** `str3 = "Air"` and `str4 = "Bill"`. The first character `'A'` of `str3` is less than the first character `'B'` of `str4`. Therefore, `str3 <= str4` is **true**.
`str2 > str4`	**true** `str2 = "Hi"` and `str4 = "Bill"`. The first character `'H'` of `str2` is greater than the first character `'B'` of `str4`. Therefore, `str2 > str4` is **true**.

Relational Operators and the `string` Type (cont'd.)

Expression	Value/Explanation
`str4 >= "Billy"`	`false` `str4 = "Bill"`. It has four characters, and `"Billy"` has five characters. Therefore, `str4` is the shorter string. All four characters of `str4` are the same as the corresponding first four characters of `"Billy"`, and `"Billy"` is the larger string. Therefore, `str4 >= "Billy"` is `false`.
`str5 <= "Bigger"`	`true` `str5 = "Big"`. It has three characters, and `"Bigger"` has six characters. Therefore, `str5` is the shorter string. All three characters of `str5` are the same as the corresponding first three characters of `"Bigger"`, and `"Bigger"` is the larger string. Therefore, `str5 <= "Bigger"` is `true`.

Logical (Boolean) Operators and Logical Expressions

- Logical (Boolean) operators: enable you to combine logical expressions

TABLE 4-2 Logical (Boolean) Operators in C++

Operator	Description
!	not
&&	and
||	or

Logical (Boolean) Operators and Logical Expressions (cont'd.)

TABLE 4-3 The ! (Not) Operator

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

EXAMPLE 4-3

Expression	Value	Explanation
`!('A' > 'B')`	`true`	Because `'A' > 'B'` is `false`, `!('A' > 'B')` is `true`.
`!(6 <= 7)`	`false`	Because `6 <= 7` is `true`, `!(6 <= 7)` is `false`.

Logical (Boolean) Operators and Logical Expressions (cont'd.)

TABLE 4-4 The && (And) Operator

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

EXAMPLE 4-4

Expression	Value	Explanation
`(14 >= 5) && ('A' < 'B')`	true	Because `(14 >= 5)` is true, `('A' < 'B')` is true, and true && true is true, the expression evaluates to true.
`(24 >= 35) && ('A' < 'B')`	false	Because `(24 >= 35)` is false, `('A' <'B')` is true, and false && true is false, the expression evaluates to false.

Logical (Boolean) Operators and Logical Expressions (cont'd.)

TABLE 4-5 The || (Or) Operator

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

EXAMPLE 4-5

Expression	Value	Explanation
`(14 >= 5) || ('A' > 'B')`	true	Because (14 >= 5) is true, ('A' > 'B') is false, and true || false is true, the expression evaluates to true.
`(24 >= 35) || ('A' > 'B')`	false	Because (24 >= 35) is false, ('A' > 'B') is false, and false || false is false, the expression evaluates to false.
`('A' <= 'a') || (7 != 7)`	true	Because ('A' <= 'a') is true, (7 != 7) is false, and true || false is true, the expression evaluates to true.

Order of Precedence

- Relational and logical operators are evaluated from left to right
  - The associativity is left to right
- Parentheses can override precedence

Order of Precedence (cont'd.)

TABLE 4-6 Precedence of Operators

Operators	Precedence
!, +, – (unary operators)	first
*, /, %	second
+, –	third
<, <=, >=, >	fourth
==, !=	fifth
&&	sixth
||	seventh
= (assignment operator)	last

Order of Precedence (cont'd.)

EXAMPLE 4-6

Suppose you have the following declarations:

```
bool found = true;
int age = 20;
double hours = 45.30;
double overTime = 15.00;
int count = 20;
char ch = 'B';
```

Order of Precedence (cont'd.)

Expression	Value / Explanation
`!found`	`false` Because `found` is `true`, `!found` is `false`.
`hours > 40.00`	`true` Because `hours` is `45.30` and `45.30 > 40.00` is `true`, the expression `hours > 40.00` evaluates to `true`.
`!age`	`false` `age` is 20, which is nonzero, so `age` is `true`. Therefore, `!age` is `false`.
`!found && (age >= 18)`	`false` `!found` is `false`; `age > 18` is `20 > 18` is `true`. Therefore, `!found && (age >= 18)` is `false && true`, which evaluates to `false`.
`!(found && (age >= 18))`	`false` Now, `found && (age >= 18)` is `true && true`, which evaluates to `true`. Therefore, `!(found && (age >= 18))` is `!true`, which evaluates to `false`.

Order of Precedence (cont'd.)

Expression	Value / Explanation
`hours + overTime <= 75.00`	`true` Because `hours + overTime` is `45.30 + 15.00 = 60.30` and `60.30 <= 75.00` is `true`, it follows that `hours + overTime <= 75.00` evaluates to `true`.
`(count >= 0) && (count <= 100)`	`true` Now, `count` is `20`. Because `20 >= 0` is `true`, `count >= 0` is `true`. Also, `20 <= 100` is `true`, so `count <= 100` is `true`. Therefore, `(count >= 0) && (count <= 100)` is `true && true`, which evaluates to `true`.
`('A' <= ch && ch <= 'Z')`	`true` Here, `ch` is `'B'`. Because `'A' <= 'B'` is `true`, `'A' <= ch` evaluates to `true`. Also, because `'B' <= 'Z'` is `true`, `ch <= 'Z'` evaluates to `true`. Therefore, `('A' <= ch && ch <= 'Z')` is `true && true`, which evaluates to `true`.

Example

- Examples using logical operators (assume a = 5 and b = 2):
- !(a > 2) → false
- (a > b) && (b >= 1) → true
- (a < b) && (b >=1) → false
- (a < b) || (b <= 1) → true

The `int` Data Type and Logical (Boolean) Expressions

- Earlier versions of C++ did not provide built-in data types that had Boolean values
- Logical expressions evaluate to either 1 or 0
  - Logical expression value was stored in a variable of the data type `int`
- Can use the `int` data type to manipulate logical (Boolean) expressions

The `bool` Data Type and Logical (Boolean) Expressions

- The data type `bool` has logical (Boolean) values `true` and `false`
- `bool`, `true`, and `false` are reserved words
- The identifier `true` has the value `1`
- The identifier `false` has the value `0`

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

- `if` and `if...else` statements can be used to create:
  - One-way selection
  - Two-way selection
  - Multiple selections

One-Way Selection

- One-way selection syntax:

```
if (expression)
    statement
```

- Statement is executed if the value of the expression is `true`
- Statement is bypassed if the value is `false`; program goes to the next statement
- `Expression` is called a decision maker

One-Way Selection (cont'd.)

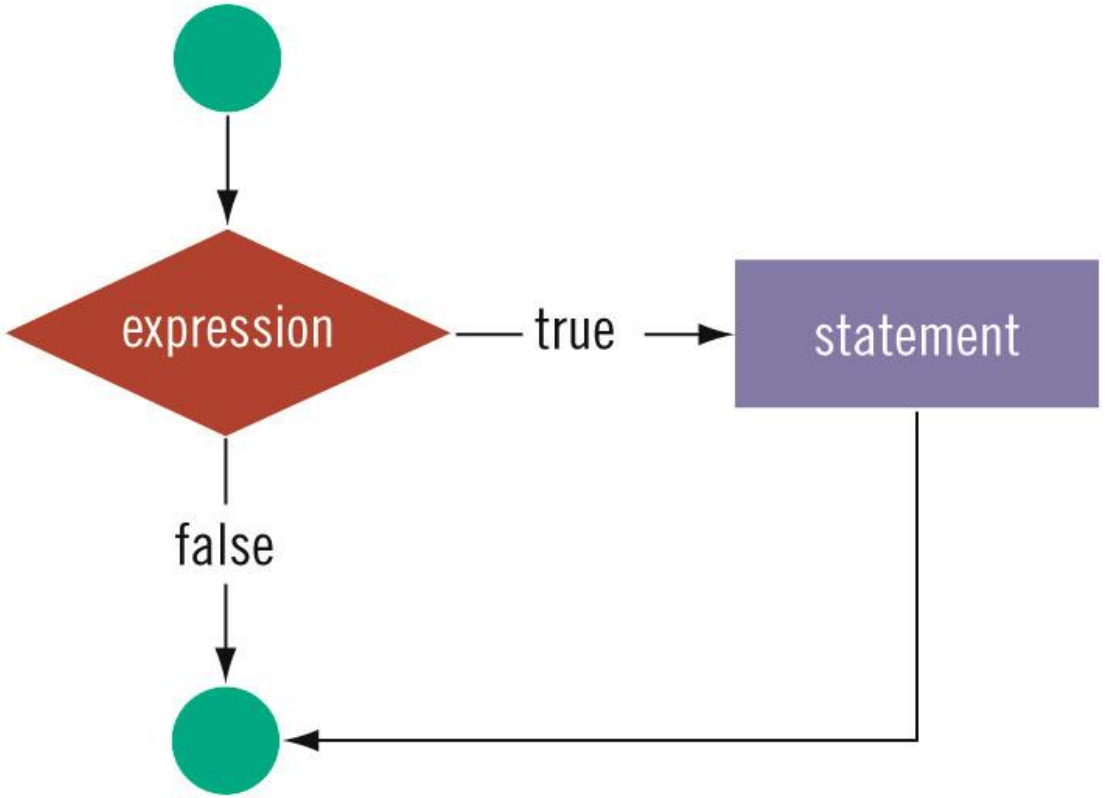


FIGURE 4-2 One-way selection

One-Way Selection Example

EXAMPLE 4-8

```
//Program to compute and output the penalty on an unpaid
//credit card balance. The program assumes that the interest
//rate on the unpaid balance is 1.5% per month.

#include <iostream>                                          //Line 1
#include <iomanip>                                           //Line 2

using namespace std;                                         //Line 3

const double INTEREST_RATE = 0.015;                          //Line 4

int main ()                                                  //Line 5
{                                                            //Line 6
    double creditCardBalance;                                //Line 7
    double payment;                                          //Line 8
    double balance;                                          //Line 9
    double penalty = 0.0;                                    //Line 10

    cout << fixed << showpoint << setprecision(2);           //Line 11
```

One-Way Selection Example

```
    cout << "Line 12: Enter credit card balance: ";          //Line 12
    cin >> creditCardBalance;                                //Line 13
    cout << endl;                                            //Line 14

    cout << "Line 15: Enter the payment: ";                  //Line 15
    cin >> payment;                                          //Line 16
    cout << endl;                                            //Line 17

    balance = creditCardBalance - payment;                   //Line 18

    if (balance > 0)                                         //Line 19
        penalty = balance * INTEREST_RATE;                   //Line 20

    cout << "Line 21: The balance is: $" << balance
         << endl;                                            //Line 21
    cout << "Line 22: The penalty to be added to your "
         << "next month bill is: $" << penalty << endl;      //Line 22

    return 0;                                                //Line 23
}                                                            //Line 24
```

Two-Way Selection

- Two-way selection syntax:

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

- If expression is `true`, `statement1` is executed; otherwise, `statement2` is executed
  - `statement1` and `statement2` are any C++ statements

Two-Way Selection (cont'd.)

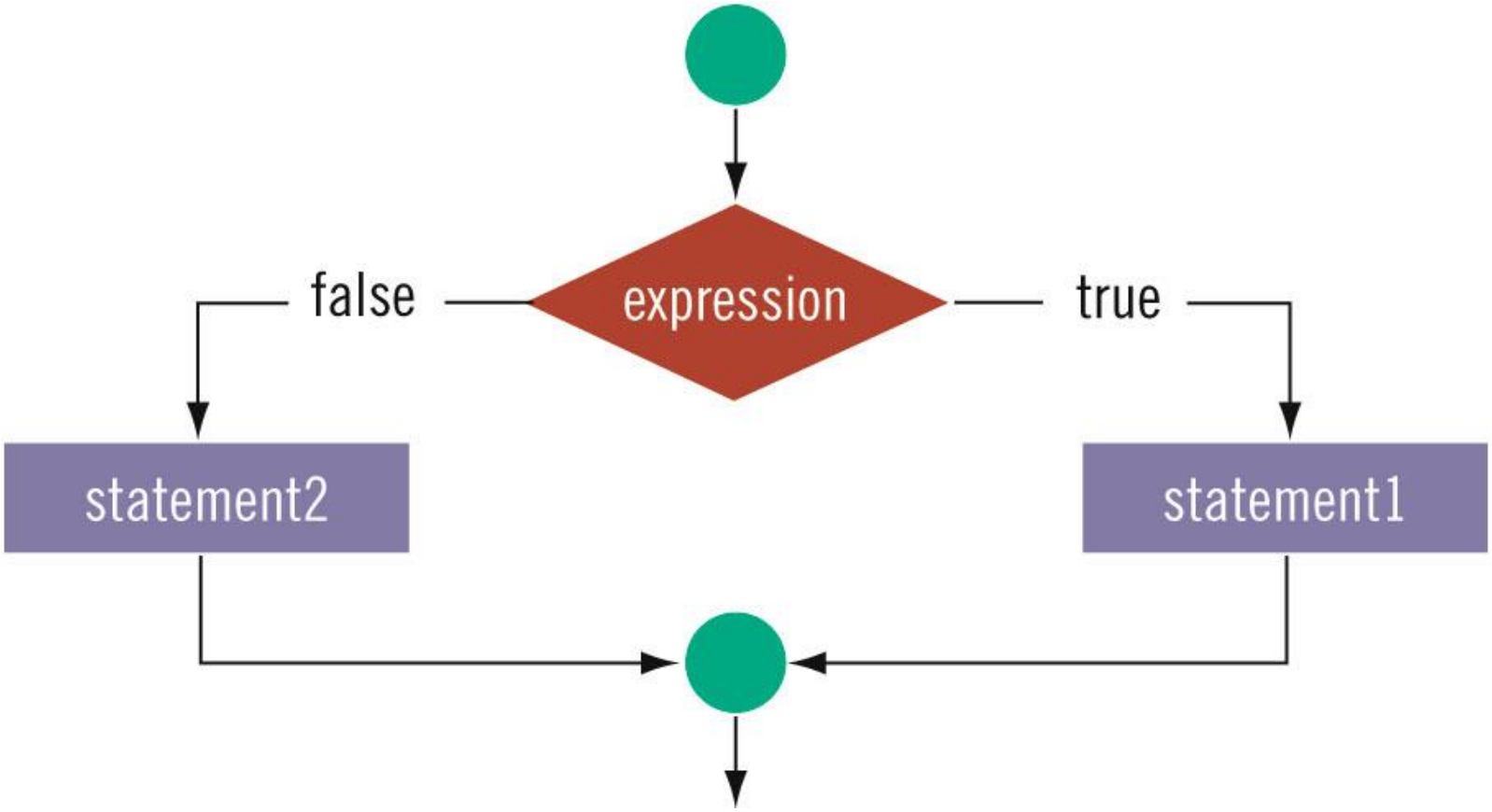


FIGURE 4-3 Two-way selection

Two-Way Selection Example

EXAMPLE 4-11

Consider the following statements:

```
if (hours > 40.0)                                    //Line 1
    wages = 40.0 * rate +
            1.5 * rate * (hours - 40.0);             //Line 2
else                                                 //Line 3
    wages = hours * rate;                            //Line 4
```

Two-Way Selection Example

EXAMPLE 4-12

The following statements show an example of a syntax error:

```
if (hours > 40.0);                                    //Line 1
    wages = 40.0 * rate +
            1.5 * rate * (hours - 40.0);              //Line 2
else                                                  //Line 3
    wages = hours * rate;                             //Line 4
```

Two-Way Selection Example

EXAMPLE 4-14

Consider the following statements:

```
if (score >= 60)                          //Line 1
    cout << "Passing" << endl;            //Line 2
    cout << "Failing" << endl;            //Line 3
```

Compound (Block of) Statements

- Compound statement (block of statements):

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

- A compound statement functions like a single statement

Compound (Block of) Statements (cont'd.)

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

Multiple Selections: Nested `if`

- Nesting: one control statement is located within another
- An `else` is associated with the most recent `if` that has not been paired with an `else`

Multiple Selections: Nested `if` (cont'd.)

EXAMPLE 4-16

Assume that `score` is a variable of type `int`. Based on the value of `score`, the following code outputs the grade:

```
if (score >= 90)
    cout << "The grade is A." << endl;
else if (score >= 80)
    cout << "The grade is B." << endl;
else if (score >= 70)
    cout << "The grade is C." << endl;
else if (score >= 60)
    cout << "The grade is D." << endl;
else
    cout << "The grade is F." << endl;
```

Multiple Selections: Nested `if` (cont'd.)

EXAMPLE 4-19

Assume that all variables are properly declared, and consider the following statements:

```
if (gender == 'M')                    //Line 1
    if (age < 21 )                    //Line 2
        policyRate = 0.05;            //Line 3
    else                              //Line 4
        policyRate = 0.035;           //Line 5
else if (gender == 'F')               //Line 6
    if (age < 21 )                    //Line 7
        policyRate = 0.04;            //Line 8
    else                              //Line 9
        policyRate = 0.03;            //Line 10
```

In this code, the `else` in Line 4 is paired with the `if` in Line 2. Note that for the `else` in Line 4, the most recent incomplete `if` is the `if` in Line 2. The `else` in Line 6 is paired with the `if` in Line 1. The `else` in Line 9 is paired with the `if` in Line 7. Once again, the indentation does not determine the pairing, but it communicates the pairing.

Example using nested if

```
#include <iostream>
using namespace std;
int main()
{
    int x = 6, y = 2;
    if (x > y)
        cout << "x is greater than y\n";
    else if (y > x)
        cout << "y is greater than x\n";
    else
        cout << "x and y are equal\n";
    return 0;
}
```

The output of this program is :
x is greater than y.

If we assign the values of x & y as follow**: int x = 2; int y = 6;**
then the output is:
y is greater than x.

If we assign the values of x & y as follow: **int x = 2; int y = 2;**
then the output is:
x and y are equal.

Comparing `if…else` Statements with a Series of `if` Statements

a.
```
if (month == 1)                          //Line 1
    cout << "January" << endl;           //Line 2
else if (month == 2)                     //Line 3
    cout << "February" << endl;          //Line 4
else if (month == 3)                     //Line 5
    cout << "March" << endl;             //Line 6
else if (month == 4)                     //Line 7
    cout << "April" << endl;             //Line 8
else if (month == 5)                     //Line 9
    cout << "May" << endl;               //Line 10
else if (month == 6)                     //Line 11
    cout << "June" << endl;              //Line 12
```

Comparing `if...else` Statements with `if` Statements (cont'd.)

b.
```
if (month == 1)
    cout << "January" << endl;
if (month == 2)
    cout << "February" << endl;
if (month == 3)
    cout << "March" << endl;
if (month == 4)
    cout << "April" << endl;
if (month == 5)
    cout << "May" << endl;
if (month == 6)
    cout << "June" << endl;
```

Short-Circuit Evaluation

- Short-circuit evaluation: evaluation of a logical expression stops as soon as the value of the expression is known

Short-Circuit Evaluation

EXAMPLE 4-20

Consider the following expressions:

```
(age >= 21) || ( x == 5)          //Line 1
(grade == 'A') && (x >= 7)        //Line 2
```

For the expression in Line 1, suppose that the value of age is 25. Because (25 >= 21) is true and the logical operator used in the expression is ||, the expression evaluates to true. Due to short-circuit evaluation, the computer does not evaluate the expression (x == 5). Similarly, for the expression in Line 2, suppose that the value of grade is 'B'. Because ('B' == 'A') is false and the logical operator used in the expression is &&, the expression evaluates to false. The computer does not evaluate (x >= 7).

Comparing Floating-Point Numbers for Equality: A Precaution

- Comparison of floating-point numbers for equality may not behave as you would expect
  - Example:
    - `1.0 == 3.0/7.0 + 2.0/7.0 + 2.0/7.0` evaluates to `false`
    - Why? `3.0/7.0 + 2.0/7.0 + 2.0/7.0 = 0.9999999999999989`
- Solution: use a tolerance value
  - Example: `if fabs(x - y) < 0.000001`

Associativity of Relational Operators: A Precaution

```
#include <iostream>

using namespace std;

int main()
{
     int num;

     cout << "Enter an integer: ";
     cin >> num;
     cout << endl;

     if (0 <= num <= 10)
         cout << num << " is within 0 and 10." << endl;


    else
        cout << num << " is not within 0 and 10." << endl;

    return 0;
}
```

Associativity of Relational Operators: A Precaution (cont'd.)

- `num = 5`

`0 <= num <= 10`	`= 0 <= 5 <= 10`	
	`= (0 <= 5) <= 10`	(Because relational operators are evaluated from left to right)
	`= 1 <= 10`	(Because `0 <= 5` is `true`, `0 <= 5` evaluates to 1)
	`= 1 (true)`	

- `num = 20`

`0 <= num <= 10`	`= 0 <= 20 <= 10`	
	`= (0 <= 20) <= 10`	(Because relational operators are evaluated from left to right)
	`= 1 <= 10`	(Because `0 <= 20` is `true`, `0 <= 20` evaluates to 1)
	`= 1 (true)`	

Avoiding Bugs by Avoiding Partially Understood Concepts and Techniques

- Must use concepts and techniques correctly
  - Otherwise solution will be either incorrect or deficient
- If you do not understand a concept or technique completely
  - Don’t use it
  - Save yourself an enormous amount of debugging time

Input Failure and the `if` Statement

- If input stream enters a fail state
  - All subsequent input statements associated with that stream are ignored
  - Program continues to execute
  - May produce erroneous results
- Can use `if` statements to check status of input stream
- If stream enters the fail state, include instructions that stop program execution

Confusion Between the Equality (==) and Assignment (=) Operators

- C++ allows you to use any expression that can be evaluated to either `true` or `false` as an expression in the `if` statement:

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

- The appearance of = in place of == resembles a *silent killer*
  - It is not a syntax error
  - It is a logical error

Conditional Operator (?:)

- Conditional operator (`?:`)
  - Ternary operator: takes 3 arguments
- Syntax for the conditional operator:
  `expression1 ? expression2 : expression3`
- If `expression1` is `true`, the result of the conditional expression is `expression2`
  - Otherwise, the result is `expression3`
- Example: `max = (a >= b) ? a : b;`

Conditional Operator (?:) Examples

Conditional Operator	Equivalent if else	Output

int A = 15, B = 2;
cout << (A > B ? A : B)
<< " is greater \n";
	int A = 15, B = 2;
	if(A>B)
	cout << A << " is greater \n";
	else
	cout<<B<<<< " is greater\n";
		15 is greater

int x, y = 15;
x = (y < 10) ? 100 : -40;
cout << "value of x: " << x ;
	int x, y = 15;
	if (y < 10)
	x=100;
	else
	x= -40;
	cout << "value of x: " << x;
		value of x: -40

Conditional Operator (?:) Examples

Conditional Operator	Equivalent if else	Output

int n;
cout << "Enter a number : ";
cin >> n;
(n% 2 == 0) ? cout << n <<
":Even number\n" : cout << n <<
":Odd number\n";
	int n;
	cout << "Enter a number : ";
	cin >> n;
	if(n% 2 == 0)
	  cout<<n<<" :Even number\n" ;
	else
	  cout<<n<<" :Odd number\n";	

`switch` Structures

- `switch` structure: alternate to `if-else`
- `switch` (integral) expression is evaluated first
- Value of the expression determines which corresponding action is taken
- Expression is sometimes called the selector

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

`switch` Structures (cont'd.)

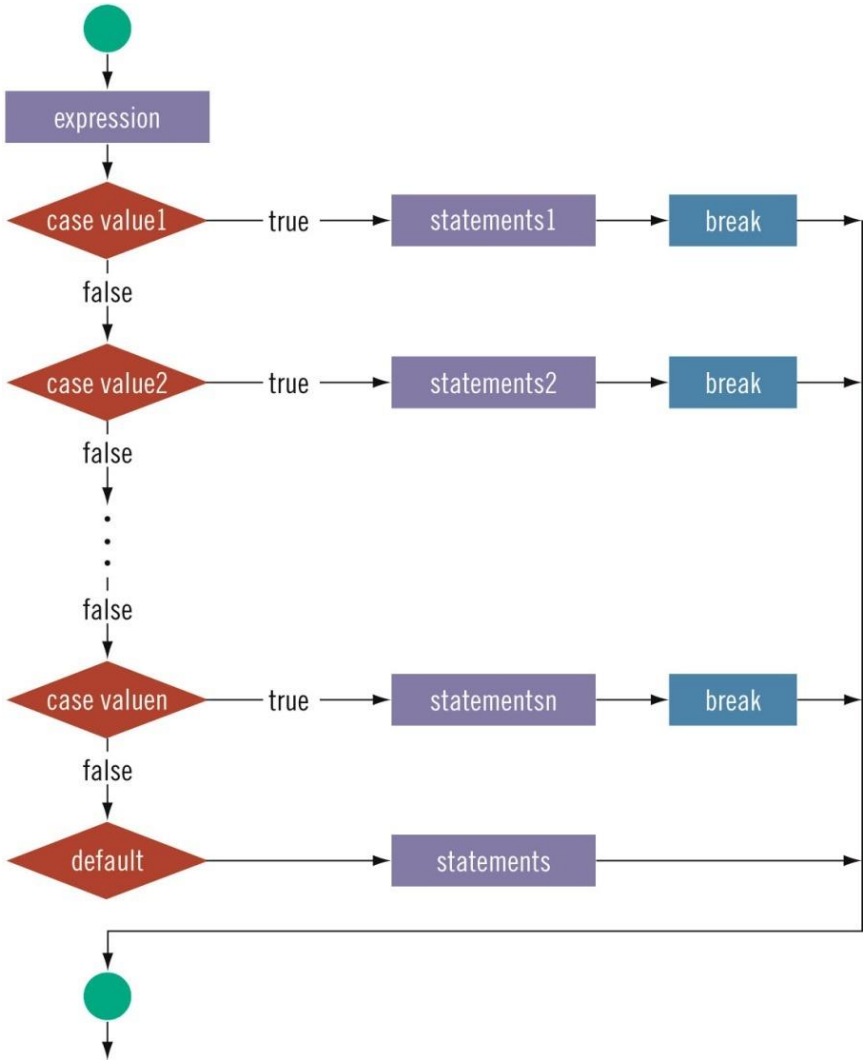


FIGURE 4-4 `switch` statement

`switch` Structures (cont'd.)

- One or more statements may follow a case label
- Braces are not needed to turn multiple statements into a single compound statement
- When a case value is matched, all statements after it execute until a `break` is encountered
- The `break` statement may or may not appear after each statement
- `switch`, `case`, `break`, and `default` are reserved words

`switch` Structures (ex. 4-21)

```
switch (grade)
{
case 'A':
     cout << "The grade point is 4.0.";
     break;
case 'B':
     cout << "The grade point is 3.0.";
     break;
case 'C':
     cout << "The grade point is 2.0.";
     break;
case 'D':
     cout << "The grade point is 1.0.";
     break;
case 'F':
     cout << "The grade point is 0.0.";
     break;
default:
     cout << "The grade is invalid.";
}
```

`switch` Structures (ex. 4-23)

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

Switch statement

```
int i, n;
cin >> i;
switch (i)
{
case 0:
case 1: n = 10;
    break;

case 2: n = 500;
    break;

default:n = 0;
    break;
}
cout << n << endl;
```

Equivalent nested if else

```
int i, n;
cin >> i;

if (i == 0 || i == 1)
    n = 10;
else if (i == 2)
    n = 500;
else
    n = 0;
cout << n << endl;
```

Avoiding Bugs: Revisited

- To output results correctly
  - Consider whether the `switch` structure must include a `break` statement after each `cout` statement

Programming Example

- Refer to page number 233 in text book and study "Cable Company Billing” example

Chapter 5:
Control Structures II (Repetition)

Objectives

- In this chapter, you will study:
  - Why Is Repetition Needed?
  - `while` Looping (Repetition) Structure
  - `for` Looping (Repetition) Structure
  - `do...while` Looping (Repetition) Structure
  - `break` and `continue` Statements
  - Nested Control Structures
  - Debugging loops

Why Is Repetition Needed?

- Repetition allows efficient use of variables
- Can input, add, and average multiple numbers using a limited number of variables
- For example, to add five numbers:
  - Declare a variable for each number, input the numbers and add the variables together
  - Create a loop that reads a number into a variable and adds it to a variable that contains the sum of the numbers

`while` Looping (Repetition) Structure

- Syntax of the `while` statement:

```
while (expression)
    statement
```

- `statement` can be simple or compound
- `expression` acts as a decision maker and is usually a logical expression
- `statement` is called the body of the loop
- The parentheses are part of the syntax

`while` Looping (Repetition) Structure (cont'd.)

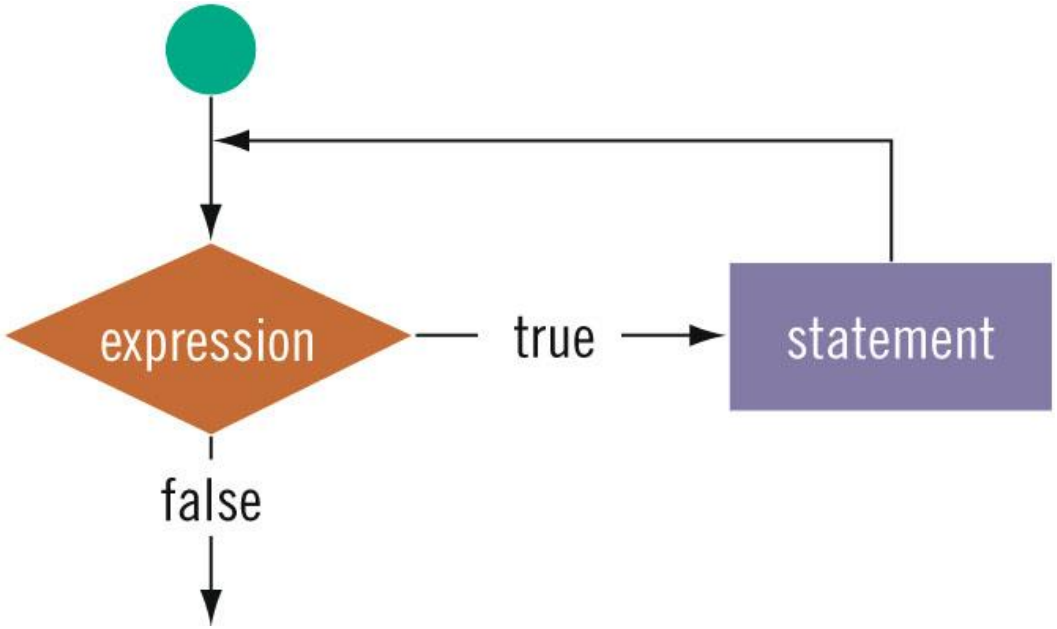


FIGURE 5-1 `while` loop

`while` Looping (Repetition) Structure (cont'd.)

EXAMPLE 5-1

Consider the following C++ program segment: (Assume that `i` is an `int` variable.)

```
i = 0;                          //Line 1

while (i <= 20)                 //Line 2
{
    cout << i << " ";           //Line 3
    i = i + 5;                  //Line 4
}

cout << endl;
```

Sample Run:

```
0 5 10 15 20
```

`while` Looping (Repetition) Structure (cont'd.)

- `i` in Example 5-1 is called the loop control variable (LCV)
- Infinite loop: continues to execute endlessly
  - Avoided by including statements in loop body that assure the exit condition is eventually false

`while` Looping (Repetition) Structure (cont'd.)

EXAMPLE 5-2

Consider the following C++ program segment:

```
i = 20;                          //Line 1
while (i < 20)                   //Line 2
{
    cout << i << " ";            //Line 3
    i = i + 5;                   //Line 4
}
cout << endl;                    //Line 5
```

It is easy to overlook the difference between this example and Example 5-1. In this example, in Line 1, `i` is set to 20. Because `i` is 20, the expression `i < 20` in the `while` statement (Line 2) evaluates to `false`. Because initially the loop entry condition, `i < 20`, is `false`, the body of the `while` loop never executes. Hence, no values are output, and the value of `i` remains 20.

Case 1: Counter-Controlled `while` Loops

- When you know exactly how many times the statements need to be executed
  - Use a counter-controlled `while` loop

```
counter = 0;          //initialize the loop control variable

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

Case 1: Counter-Controlled `while` Loops (Ex. 5-3)

- 10 Students at a local middle school volunteered to sell fresh baked cookies to raise funds to increase the number of computers for the computer lab. Each student reported the number of boxes he/she sold. We will write a program that will do the following:
  - Ask each student about the total number of boxes of cookies he/she sold
  - Output the total number of boxes of cookies sold
  - Output the total revenue generated by selling the cookies
  - Output the average number of boxes sold by each student
- Assume the cost of each box of cookies = 5$.

Case 2: Sentinel-Controlled `while` Loops

- Sentinel variable is tested in the condition
- Loop ends when sentinel is encountered

```
cin >> variable;                  //initialize the loop control variable

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

Example 5-5: Telephone Digits

- Example 5-5 provides an example of a sentinel-controlled loop
- The program converts uppercase letters to their corresponding telephone digit

Case 3: Flag-Controlled `while` Loops

- Flag-controlled `while` loop: uses a `bool` variable to control the loop

```
found = false;        //initialize the loop control variable

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

Number Guessing Game

- Example 5-6 implements a number guessing game using a flag-controlled while loop
- Uses the function `rand` of the header file `cstdlib` to generate a random number
  - `rand()` returns an `int` value between 0 and 32767
  - To convert to an integer >= 0 and < 100:
    - `rand() % 100`

Number Guessing Game

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main()
{
        //declare the variables
    int num;          //variable to store the random
                      //number
    int guess;        //variable to store the number
                      //guessed by the user
    bool isGuessed;   //boolean variable to control
                      //the loop

    srand(time(0));                                   //Line 1
    num = rand() % 100;                               //Line 2

    isGuessed = false;                                //Line 3

    while (!isGuessed)                                //Line 4
    {                                                 //Line 5
        cout << "Enter an integer greater"
             << " than or equal to 0 and "
             << "less than 100: ";                    //Line 6

        cin >> guess;                                 //Line 7
        cout << endl;                                 //Line 8
```

Number Guessing Game

```
        if (guess == num)                                   //Line 9
        {                                                   //Line 10
            cout << "You guessed the correct "
                 << "number." << endl;                      //Line 11
            isGuessed = true;                               //Line 12
        }                                                   //Line 13
        else if (guess < num)                               //Line 14
            cout << "Your guess is lower than the "
                 << "number.\n Guess again!"
                 << endl;                                   //Line 15
        else                                                //Line 16
            cout << "Your guess is higher than "
                 << "the number.\n Guess again!"
                 << endl;                                   //Line 17
    } //end while                                          //Line 18

    return 0;
}
```

More on Expressions in `while` Statements

- The expression in a `while` statement can be complex
  - Example:

```
while ((noOfGuesses < 5) && (!isGuessed))
{
   . . .
}
```

Programming Example: Fibonacci Number

- Consider the following sequence of numbers:
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Called the Fibonacci sequence
- Given the first two numbers of the sequence (say, a1 and a2)
  - n^{th} number a_n, n >= 3, of this sequence is given by:

    $a_n = a_{n-1} + a_{n-2}$

Programming Example: Fibonacci Number (cont'd.)

- Fibonacci sequence
  - n^{th} Fibonacci number
  - $a_2 = 1$
  - $a_1 = 1$
  - Determine the n^{th} number a_n, n >= 3

Programming Example: Fibonacci Number (cont'd.)

- Suppose $a_2 = 6$ and $a_1 = 3$
  - $a_3 = a_2 + a_1 = 6 + 3 = 9$
  - $a_4 = a_3 + a_2 = 9 + 6 = 15$
- Write a program that determines the n^{th} Fibonacci number, given the first two numbers

Programming Example: Input and Output

- Input: first two Fibonacci numbers and the desired Fibonacci number
- Output: n^{th} Fibonacci number

Programming Example: Problem Analysis and Algorithm Design

- Algorithm:
  - Get the first two Fibonacci numbers
  - Get the desired Fibonacci number
    - Get the position, *n*, of the number in the sequence
  - Calculate the next Fibonacci number
    - Add the previous two elements of the sequence
  - Repeat Step 3 until the n^{th} Fibonacci number is found
  - Output the n^{th} Fibonacci number

Programming Example: Variables

```
int previous1;  //variable to store the first Fibonacci number
int previous2;  //variable to store the second Fibonacci number
int current;    //variable to store the current
                //Fibonacci number
int counter;    //loop control variable
int nthFibonacci; //variable to store the desired
                  //Fibonacci number
```

Programming Example: Main Algorithm

- Prompt the user for the first two numbers—that is, `previous1` and `previous2`
- Read (input) the first two numbers into `previous1` and `previous2`
- Output the first two Fibonacci numbers
- Prompt the user for the position of the desired Fibonacci number

Programming Example: Main Algorithm (cont'd.)

- Read the position of the desired Fibonacci number into `nthFibonacci`
  - `if (nthFibonacci == 1)`
    The desired Fibonacci number is the first Fibonacci number; copy the value of `previous1` into `current`
  - `else if (nthFibonacci == 2)`
    The desired Fibonacci number is the second Fibonacci number; copy the value of `previous2` into `current`

Programming Example: Main Algorithm (cont'd.)

- `else` calculate the desired Fibonacci number as follows:
  - Start by determining the third Fibonacci number
  - Initialize `counter` to 3 to keep track of the calculated Fibonacci numbers.
  - Calculate the next Fibonacci number, as follows:
    ```
    current = previous2 + previous1;
    ```

Programming Example: Main Algorithm (cont'd.)

- (cont'd.)
  - Assign the value of `previous2` to `previous1`
  - Assign the value of `current` to `previous2`
  - Increment `counter`
  - Repeat until Fibonacci number is calculated:

```
while (counter <= nthFibonacci)
{
    current = previous2 + previous1;
    previous1 = previous2;
    previous2 = current;
    counter++;
}
```

Programming Example: Main Algorithm (cont'd.)

- Output the `nthFibonacci` number, which is current

`for` Looping (Repetition) Structure

- `for` loop: called a counted or indexed `for` loop
- Syntax of the `for` statement:

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

- The `initial statement`, `loop condition`, and `update statement` are called `for` loop control statements

`for` Looping (Repetition) Structure (cont'd.)

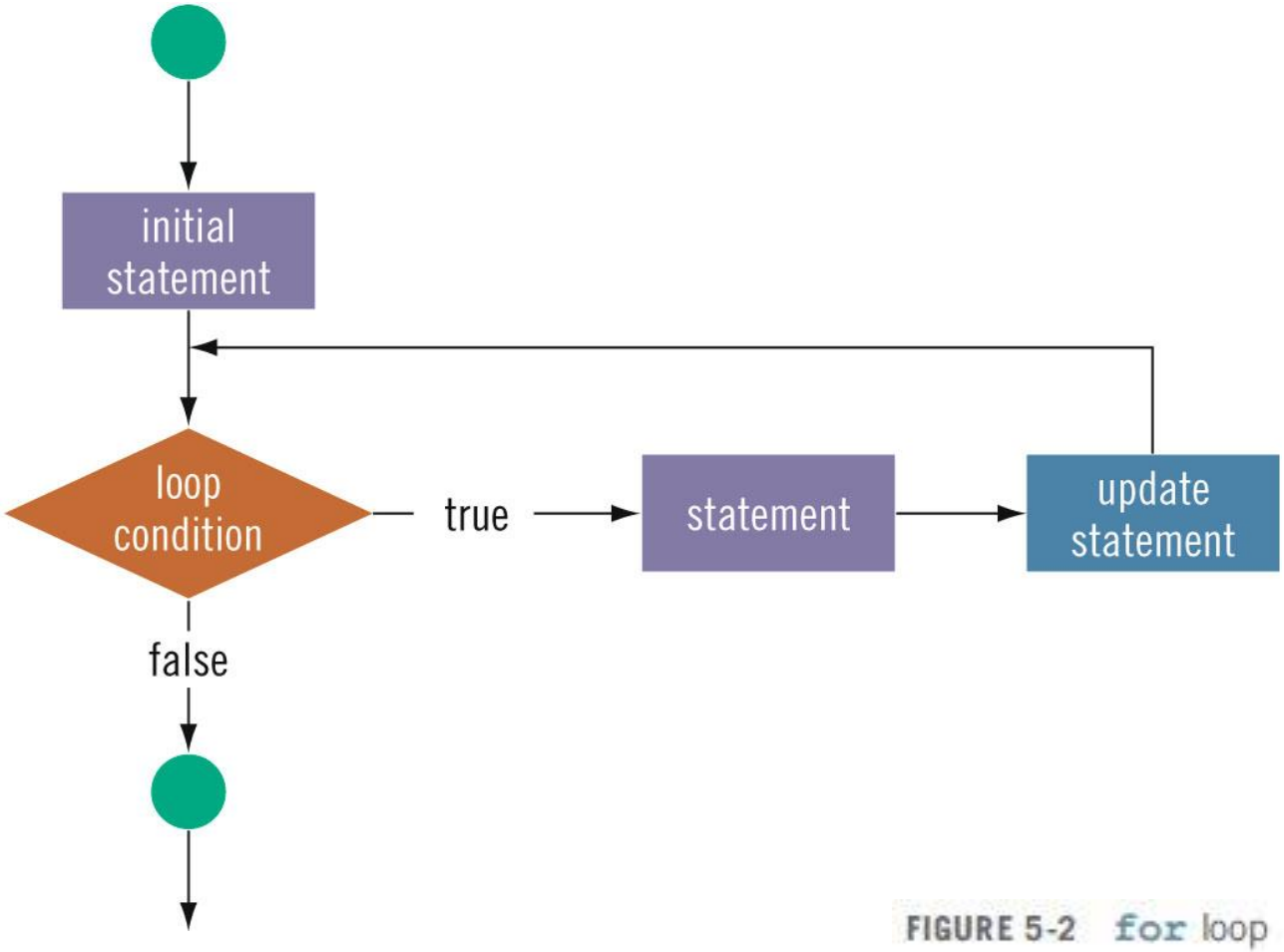


FIGURE 5-2 `for` loop

`for` Looping (Repetition) Structure (cont'd.)

EXAMPLE 5-9

The following `for` loop prints the first 10 nonnegative integers:

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

The `initial statement`, `i = 0;`, initializes the `int` variable `i` to 0. Next, the loop condition, `i < 10`, is evaluated. Because `0 < 10` is `true`, the print statement executes and outputs 0. The `update statement`, `i++`, then executes, which sets the value of `i` to 1. Once again, the `loop condition` is evaluated, which is still `true`, and so on. When `i` becomes 10, the `loop condition` evaluates to `false`, the `for` loop terminates, and the statement following the `for` loop executes.

`for` Looping (Repetition) Structure (cont'd.)

EXAMPLE 5-10

1. The following `for` loop outputs `Hello!` and a star (on separate lines) five times:

```
for (i = 1; i <= 5; i++)
{
    cout << "Hello!" << endl;
    cout << "*" << endl;
}
```

2. Consider the following `for` loop:

```
for (i = 1; i <= 5; i++)
    cout << "Hello!" << endl;
    cout << "*" << endl;
```

This loop outputs `Hello!` five times and the star only once.

`for` Looping (Repetition) Structure (cont'd.)

- The following is a semantic error:

EXAMPLE 5-11

The following `for` loop executes five empty statements:

```
for (i = 0; i < 5; i++);      //Line 1
    cout << "*" << endl;      //Line 2
```

The semicolon at the end of the `for` statement (before the output statement, Line 1) terminates the `for` loop. The action of this `for` loop is empty, that is, null.

- The following is a legal (but infinite) `for` loop:

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

`for` Looping (Repetition) Structure (cont'd.)

EXAMPLE 5-12

You can count backward using a `for` loop if the `for` loop control expressions are set correctly.

For example, consider the following `for` loop:

```
for (i = 10; i >= 1; i--)
    cout << " " << i;
cout << endl;
```

The output is:

```
10 9 8 7 6 5 4 3 2 1
```

In this `for` loop, the variable `i` is initialized to `10`. After each iteration of the loop, `i` is decremented by `1`. The loop continues to execute as long as `i >= 1`.

`for` Looping (Repetition) Structure (cont'd.)

EXAMPLE 5-13

You can increment (or decrement) the loop control variable by any fixed number. In the following `for` loop, the variable is initialized to `1`; at the end of the `for` loop, `i` is incremented by `2`. This `for` loop outputs the first `10` positive odd integers.

```
for (i = 1; i <= 20; i = i + 2)
    cout << " " << i;
cout << endl;
```

do…while Looping (Repetition) Structure

- Syntax of a `do...while` loop:

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

- The `statement` executes first, and then the `expression` is evaluated
  - As long as `expression` is true, loop continues
- To avoid an infinite loop, body must contain a statement that makes the `expression` false

`do…while` Looping (Repetition) Structure (cont'd.)

- The statement can be simple or compound
- Loop always iterates at least once

do...while Looping (Repetition) Structure (cont'd.)

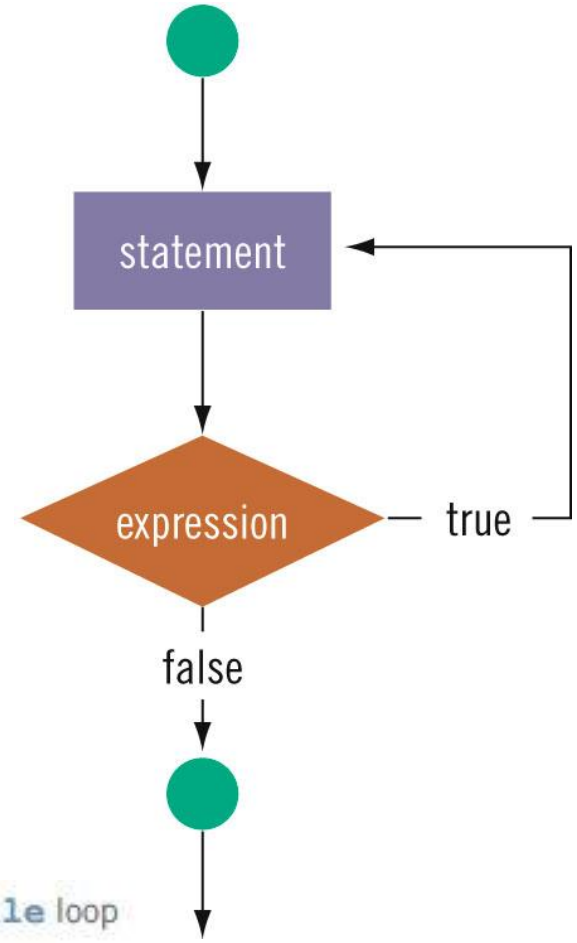


FIGURE 5-3 do...while loop

do...while Looping (Repetition) Structure (cont'd.)

EXAMPLE 5-18

```
i = 0;

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

The output of this code is:

```
0 5 10 15 20
```

After 20 is output, the statement:

```
i = i + 5;
```

changes the value of i to 25 and so i <= 20 becomes false, which halts the loop.

do...while Looping (Repetition) Structure (cont'd.)

EXAMPLE 5-19

Consider the following two loops:

a.
```
i = 11;
while (i <= 10)
{
    cout << i << " ";
    i = i + 5;
}
cout << endl;
```

b.
```
i = 11;
do
{
    cout << i << " ";
    i = i + 5;
}
while (i <= 10);

cout << endl;
```

In (a), the `while` loop produces nothing. In (b), the `do...while` loop outputs the number 11 and also changes the value of `i` to 16.

Choosing the Right Looping Structure

- All three loops have their place in C++
  - If you know or can determine in advance the number of repetitions needed, the `for` loop is the correct choice
  - If you do not know and cannot determine in advance the number of repetitions needed, and it could be zero, use a `while` loop
  - If you do not know and cannot determine in advance the number of repetitions needed, and it is at least one, use a `do...while` loop

`break` and `continue` Statements

- `break` and `continue` alter the flow of control
- `break` statement is used for two purposes:
  - To exit early from a loop
    - Can eliminate the use of certain (flag) variables
  - To skip the remainder of a `switch` structure
- After `break` executes, the program continues with the first statement after the structure

`break` and `continue` Statements (cont'd.)

- `continue` is used in `while`, `for`, and `do…while` structures
- When executed in a loop
  - It skips remaining statements and proceeds with the next iteration of the loop

Using break and continue example

```
sum = 0;
isNegative = false;

cin >> num;

while (cin && !isNegative)
{
    if (num < 0)    //if num is negative, terminate the loop
                    //after this iteration
    {
        cout << "Negative number found in the data." << endl;
        isNegative = true;
    }
    else
    {
        sum = sum + num;
        cin >> num;
    }
}
```

Using break and continue example

```
sum = 0;
cin >> num;

while (cin)
{
    if (num < 0)    //if num is negative, terminate the loop
    {
        cout << "Negative number found in the data." << endl;
        break;
    }

    sum = sum + num;
    cin >> num;
}
```

Using break and continue example

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

    sum = sum + num;
    cin >> num;
}
```

Nested Control Structures

- To create the following pattern:

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

- We can use the following code:

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

Nested Control Structures (cont'd.)

- What is the result if we replace the first for statement with this?

```
for (i = 5; i >= 1; i--)
```

- Answer:

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

Nested Control Structures (cont'd.)

- Write the pseudocode to create the following multiplication table:

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

Debugging Loops

- Loops are harder to debug than sequence and selection structures
- Use loop invariant
  - Set of statements that remains true each time the loop body is executed
- Most common error associated with loops is off-by-one

Chapter 6:
User-Defined Functions

C++ PROGRAMMING:
From Problem Analysis to Program Design

Outline

In this chapter, you will study:

- Predefined Functions
- User-Defined Functions
- Value-Returning Functions
- Void Functions
- Value Parameters
- Reference Variables as Parameters
- Value and Reference Parameters and Memory Allocation

Outline

In this chapter, you will also study:

- Reference Parameters and Value-Returning Functions
- Scope of an Identifier
- Global Variables, Named Constants, and Side Effects
- Static and Automatic Variables
- Function Overloading: An Introduction
- Functions with Default Parameters

Introduction

- Functions are often called modules
- They are like miniature programs that can be combined to form larger programs
- They allow complicated programs to be divided into manageable pieces

Predefined Functions

- In C++, a function is similar to that of a function in algebra
  - It has a name
  - It does some computation
- Some of the predefined mathematical functions are:

```
sqrt(x)
pow(x, y)
floor(x)
```

Predefined Functions (cont'd.)

- Predefined functions are organized into separate libraries
  - I/O functions are in `iostream` header
  - Math functions are in `cmath` header
- To use predefined functions, you must include the header file using an `include` statement
- See Table 6-1 in the text for some common predefined functions

Predefined Functions (cont'd.)

Function	Header File	Purpose	Parameter(s) Type	Result
`abs(x)`	`<cmath>`	Returns the absolute value of its argument: `abs(-7) = 7`	`int` (`double`)	`int` (`double`)
`ceil(x)`	`<cmath>`	Returns the smallest whole number that is not less than `x`: `ceil(56.34) = 57.0`	`double`	`double`
`islower(x)`	`<cctype>`	Returns 1 (`true`) if `x` is a lowercase letter; otherwise, it returns 0 (`false`); `islower('h')` is 1 (`true`)	`int`	`int`
`isupper(x)`	`<cctype>`	Returns 1 (`true`) if `x` is an uppercase letter; otherwise, it returns 0 (`false`); `isupper('K')` is 1 (`true`)	`int`	`int`
`pow(x, y)`	`<cmath>`	Returns x^y; if `x` is negative, `y` must be a whole number: `pow(0.16, 0.5) = 0.4`	`double`	`double`
`sqrt(x)`	`<cmath>`	Returns the nonnegative square root of `x`; `x` must be nonnegative: `sqrt(4.0) = 2.0`	`double`	`double`

User-Defined Functions

- Value-returning functions: have a return type
  - Return a value of a specific data type using the `return` statement
- Void functions: do not have a return type
  - *Do not* use a `return` statement to return a value

Value-Returning Functions

- To use these functions, you must:
  - Include the appropriate header file in your program using the include statement
  - Know the following items:
    - Name of the function
    - Number of parameters, if any
    - Data type of each parameter
    - Data type of the value returned: called the type of the function

Value-Returning Functions (cont'd.)

- Can use the value returned by a value-returning function by:
  - Saving it for further calculation
  - Using it in some calculation
  - Printing it
- A value-returning function is used in an assignment or in an output statement

Value-Returning Functions (cont'd.)

- Heading (or function header): first line of the function
  - Example: `int abs(int number)`
- Formal parameter: variable declared in the heading
  - Example: `number`
- Actual parameter: variable or expression listed in a call to a function
  - Example: `x = pow(u, v)`

Syntax: Value-Returning Function

- Syntax:

```
functionType functionName(formal parameter list)
{
    statements
}
```

- `functionType` is also called the data type or return type

Syntax: Formal Parameter List

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

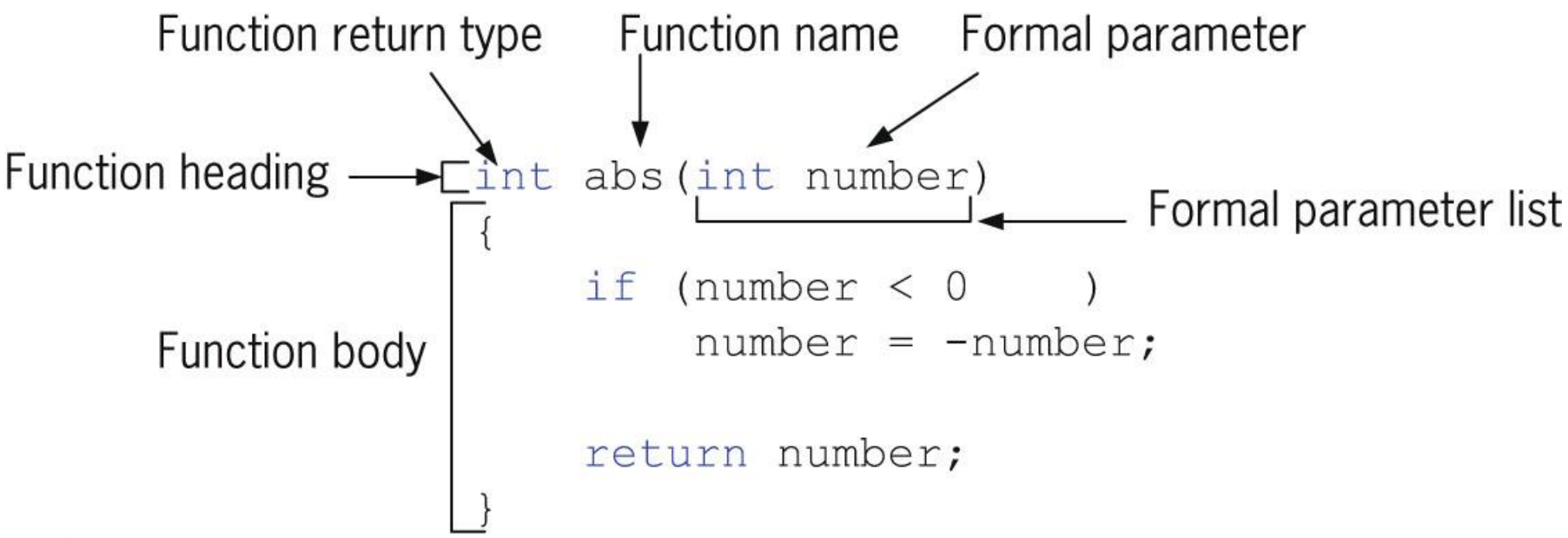


FIGURE 6-1 Various parts of the function abs

Function Call

- Syntax to call a value-returning function:

```
functionName(actual parameter list)
```

Syntax: Actual Parameter List

- Syntax of the actual parameter list:

```
expression or variable, expression or variable, ...
```

- Formal parameter list can be empty:

```
functionType functionName()
```

- A call to a value-returning function with an empty formal parameter list is:

```
functionName()
```

`return` Statement

- Function returns its value via the `return` statement
  - It passes this value outside the function

Syntax: `return` Statement

- Syntax:

```
return expr;
```

- In C++, `return` is a reserved word
- When a return statement executes
  - **Function immediately terminates**
  - Control goes back to the caller
- When a `return` statement executes in the function `main`, the program terminates

Syntax: `return` Statement (cont'd.)

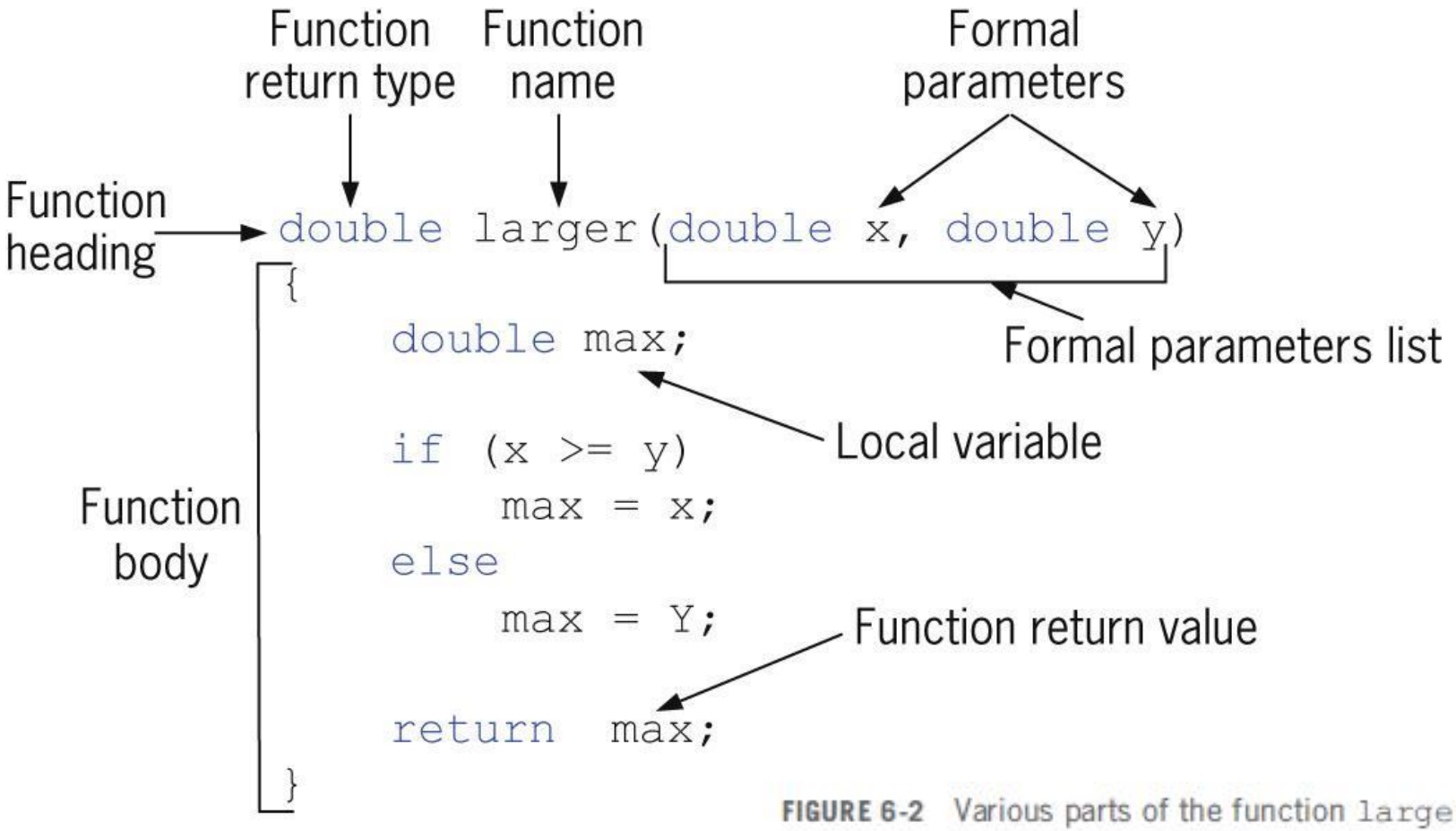


FIGURE 6-2 Various parts of the function `larger`

Example 6-2

Now that the function larger is written, the following C++ code illustrates how to use it:

```
double one = 13.00;
double two = 36.53;
double maxNum;
```

Consider the following statements:

```
cout << "The larger of 5 and 6 is " << larger(5, 6)
     << endl;                                          //Line 1

cout << "The larger of " << one << " and " << two
     << " is " << larger(one, two) << endl;            //Line 2

cout << "The larger of " << one << " and 29 is "
     << larger(one, 29) << endl;                       //Line 3

maxNum = larger(38.45, 56.78);                         //Line 4
```

Syntax: `return` Statement (cont'd.)

- In a function call, you specify only the actual parameter, not its data type.
- The following statements contain incorrect calls to the function `larger` and would result in syntax errors

```
x = larger(int one, 29);                //illegal
y = larger(int one, int 29);            //illegal
cout << larger(int one, int two);       //illegal
```

Syntax: `return` Statement (cont'd.)

- Once a function is written, you can use it anywhere in the program. Even as a parameter to another function

```
double compareThree(double x, double y, double z)
{
        return larger(x, larger(y, z));
}
```

Function Prototype

- Function prototype: function heading without the body of the function
- Syntax:

```
functionType functionName(parameter list);
```

- Not necessary to specify the variable name in the parameter list
- Data type of each parameter must be specified

Function Prototype Example

```
double larger(double x, double y);
double compareThree(double x, double y, double z);

int main()
{
    double one, two;                                    //Line 1

    cout << "Line 2: The larger of 5 and 10 is "
         << larger(5, 10) << endl;                      //Line 2

    cout << "Line 3: Enter two numbers: ";              //Line 3
    cin >> one >> two;                                  //Line 4
    cout << endl;                                       //Line 5

    cout << "Line 6: The larger of " << one
         << " and " << two << " is "
         << larger(one, two) << endl;                   //Line 6
```

Function Prototype Example (cont'd)

```
    cout << "Line 7: The largest of 43.48, 34.00, "
         << "and 12.65 is "
         << compareThree(43.48, 34.00, 12.65)
         << endl;                                    //Line 7

    return 0;
}

double larger(double x, double y)
{
    double max;

    if (x >= y)
        max = x;
    else
        max = y;

    return max;
}

double compareThree (double x, double y, double z)
{
    return larger(x, larger(y, z));
}
```

Value-Returning Functions: Some Peculiarities

```
int secret(int x)
{
    if (x > 5)                  //Line 1
        return 2 * x;           //Line 2
}
```

A correct definition of the function secret is:

```
int secret(int x)
{
    if (x > 5)                  //Line 1
        return 2 * x;           //Line 2

    return x;                   //Line 3
}
```

Value-Returning Functions: Some Peculiarities (cont'd.)

```
return x, y;   //only the value of y will be returned
```

```
int funcRet1()
{
    int x = 45;

    return 23, x;   //only the value of x is returned
}

int funcRet2(int z)
{
    int a = 2;
    int b = 3;

    return 2 * a + b, z + b; //only the value of z + b is returned
}
```

EXAMPLE 6-4 (ROLLING A PAIR OF DICE)

- write the function `rollDice` that takes as a parameter **the desired sum of the numbers to be rolled** and returns the **number of times the dice are rolled to roll the desired sum**.

EXAMPLE 6-4 (ROLLING A PAIR OF DICE) (cont'd)

```
int rollDice(int num)
{
    int die1;
    int die2;
    int sum;
    int rollCount = 0;

    srand(time(0));

    do
    {
        die1 = rand() % 6 + 1;
        die2 = rand() % 6 + 1;
        sum = die1 + die2;
        rollCount++;
    }
    while (sum != num);

    return rollCount;
}
```

More Examples of Value-Returning Functions

- For more examples refer to text book page 352

Flow of Execution

- Execution always begins at the first statement in the function `main`
- Other functions are executed only when called
- Function prototypes appear before any function definition
  - Compiler translates these first
- Compiler can then correctly translate a function call

Flow of Execution (cont'd.)

- Function call transfers control to the first statement in the body of the called function
- When the end of a called function is executed, control is passed back to the point immediately following the function call
  - Function's returned value replaces the function call statement

Void Functions

- User-defined void functions can be placed either before or after the function `main`
- If user-defined void functions are placed after the function `main`
  - The function prototype must be placed before the function `main`
- Void function does not have a return type
  - `return` statement without any value is typically used to exit the function early

Void Functions (cont'd.)

- Formal parameters are optional
- A call to a void function is a stand-alone statement
- Void function definition syntax:

```
void functionName(formal parameter list)
{
     statements
}
```

Void Functions (cont'd.)

- Formal parameter list syntax:

```
dataType& variable, dataType& variable, ...
```

- Function call syntax:

```
functionName(actual parameter list);
```

- Actual parameter list syntax:

```
expression or variable, expression or variable, ...
```

Void Functions (cont'd.)

- Value parameter: a formal parameter that receives a copy of the content of corresponding actual parameter
- Reference parameter: a formal parameter that receives the location (memory address (&)) of the corresponding actual parameter

Example 6-8

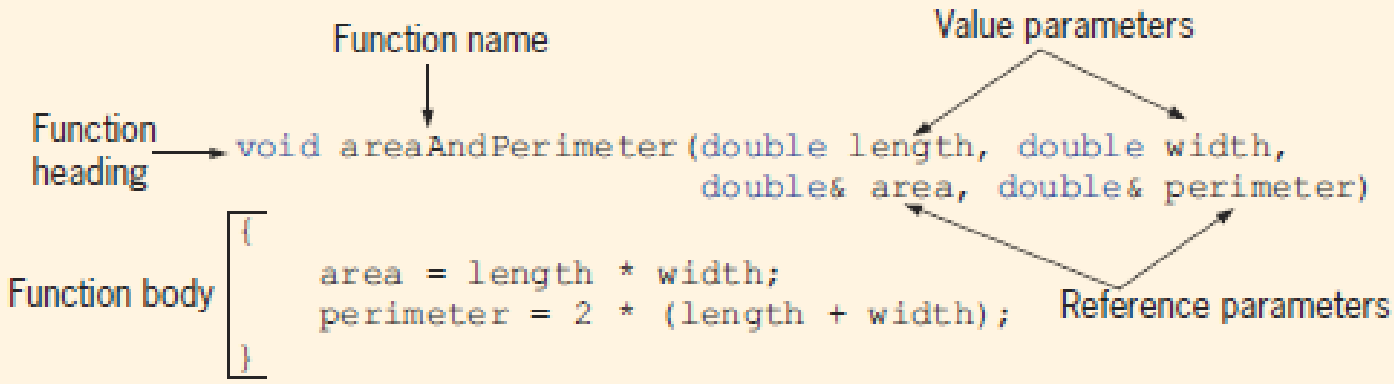


FIGURE 6-4 Various parts of the function `areaAndPerimeter`

- For more examples refer to text book page 366

Value Parameters

- If a formal parameter is a value parameter:
  - The value of the corresponding actual parameter is copied into it
  - Formal parameter has its own copy of the data
- During program execution
  - Formal parameter manipulates the data stored in its own memory space

Value Parameters Example

```
void funcValueParam(int num);

int main()
{
    int number = 6;                                          //Line 1

    cout << "Line 2: Before calling the function "
         << "funcValueParam, number = " << number
         << endl;                                            //Line 2

    funcValueParam(number);                                  //Line 3

    cout << "Line 4: After calling the function "
         << "funcValueParam, number = " << number
         << endl;                                            //Line 4

      return 0;
}
```

Value Parameters Example (cont'd)

```
void funcValueParam(int num)
{
    cout << "Line 5: In the function funcValueParam, "
         << "before changing, num = " << num
         << endl;                                          //Line 5

    num = 15;                                              //Line 6
    cout << "Line 7: In the function funcValueParam, "
         << "after changing, num = " << num
         << endl;                                          //Line 7
}
```

Reference Variables as Parameters

- If a formal parameter is a reference parameter
  - It receives the memory address of the corresponding actual parameter
- During program execution to manipulate data
  - Changes to formal parameter will change the corresponding actual parameter

Reference Variables as Parameters (cont'd.)

- Reference parameters are useful in three situations:
  - Returning more than one value
  - Changing the actual parameter
  - When passing the address would save memory space and time

Example 6-12 (Calculate Grade)

- Write a program that takes a course score (a value between 0 and 100) and determines a student's course grade. This program has three functions: main, getScore, and printGrade, as follows:
- main
  - Get the course score.
  - Print the course grade.
- getScore
  - Prompt the user for the input.
  - Get the input.
  - Print the course score.
- printGrade
  - Calculate the course grade.
  - Print the course grade.

Example 6-12 (Calculate Grade) (cont'd)

```
void getScore(int& score);
void printGrade(int score);

int main()
{
    int courseScore;

    cout << "Line 1: Based on the course score, \n"
         << "   this program computes the "
         << "course grade." << endl;                    //Line 1

    getScore(courseScore);                              //Line 2

    printGrade(courseScore);                            //Line 3

    return 0;
}
```

Example 6-12 (Calculate Grade) (cont'd)

```
void getScore(int& score)
{
    cout << "Line 4: Enter course score: ";          //Line 4
    cin >> score;                                    //Line 5
    cout << endl << "Line 6: Course score is "
         << score << endl;                           //Line 6
}

void printGrade(int cScore)
{
    cout << "Line 7: Your grade for the course is "; //Line 7

    if (cScore >= 90)                                //Line 8
        cout << "A." << endl;
    else if (cScore >= 80)
        cout << "B." << endl;
    else if(cScore >= 70)
        cout << "C." << endl;
    else if (cScore >= 60)
        cout << "D." << endl;
    else
        cout << "F." << endl;
}
```

Value and Reference Parameters and Memory Allocation

- When a function is called
  - Memory for its formal parameters and its local variables is allocated in the function data area
- For a value parameter, the actual parameter's value is copied into the formal parameter's memory cell
  - Changes to the formal parameter do not affect the actual parameter's value

Value and Reference Parameters and Memory Allocation (cont'd.)

- For a reference parameter, the actual parameter's address passes to the formal parameter
  - Both formal and actual parameters refer to the same memory location
  - During execution, changes made to the formal parameter's value permanently change the actual parameter's value

Example 6-14

```
void addFirst(int& first, int& second);
void doubleFirst(int one, int two);
void squareFirst(int& ref, int val);

int main()
{
    int num = 5;

    cout << "Line 1: Inside main: num = " << num
         << endl;                                          //Line 1

    addFirst(num, num);                                    //Line 2
    cout << "Line 3: Inside main after addFirst:"
         << " num = " << num << endl;                      //Line 3

    doubleFirst(num, num);                                 //Line 4
    cout << "Line 5: Inside main after "
         << "doubleFirst: num = " << num << endl;          //Line 5

    squareFirst(num, num);                                 //Line 6
    cout << "Line 7: Inside main after "
         << "squareFirst: num = " << num << endl;          //Line 7

    return 0;
}
```

Example 6-14 (Cont'd)

```
void addFirst(int& first, int& second)
{
    cout << "Line 8: Inside addFirst:  first = "
         << first << ", second = " << second << endl; //Line 8

    first = first + 2;                                //Line 9

    cout << "Line 10: Inside addFirst:  first = "
         << first << ", second = " << second << endl; //Line 10

    second = second * 2;                              //Line 11

    cout << "Line 12: Inside addFirst:  first = "
         << first << ", second = " << second << endl; //Line 12
}
```

Example 6-14 (Cont'd)

```
double larger(double x, double y);
double compareThree(double x, double y, double z);

int main()
{
    double one, two;                                    //Line 1

    cout << "Line 2: The larger of 5 and 10 is "
         << larger(5, 10) << endl;                      //Line 2

    cout << "Line 3: Enter two numbers: ";              //Line 3
    cin >> one >> two;                                  //Line 4
    cout << endl;                                       //Line 5

    cout << "Line 6: The larger of " << one
         << " and " << two << " is "
         << larger(one, two) << endl;                   //Line 6
```

Example 6-14 (Cont'd)

```
void squareFirst(int& ref, int val)
{
    cout << "Line 18: Inside squareFirst: ref = "
         << ref << ", val = " << val << endl;          //Line 18

    ref = ref * ref;                                   //Line 19

    cout << "Line 20: Inside squareFirst: ref = "
         << ref << ", val = " << val << endl;          //Line 20

    val = val + 2;                                     //Line 21

    cout << "Line 22: Inside squareFirst: ref = "
         << ref << ", val = " << val << endl;          //Line 22
}
```

Reference Parameters and Value-Returning Functions

- Can also use reference parameters in a value-returning function
  - Not recommended
- By definition, a value-returning function returns a single value via `return` statement
- If a function needs to return more than one value, change it to a void function and use reference parameters to return the values

Scope of an Identifier

- Scope of an identifier: where in the program the identifier is accessible
- Local identifier: identifiers declared within a function (or block)
- Global identifier: identifiers declared outside of every function definition
- **C++ does not allow nested functions**
  - Definition of one function cannot be included in the body of another function

Scope of an Identifier (cont'd.)

- Rules when an identifier is accessed:
  - Global identifiers are accessible by a function or block if:
    - Declared before function definition
    - Function name different from identifier
    - Parameters to the function have different names
    - All local identifiers have different names

Scope of an Identifier (cont'd.)

- Rules when an identifier is accessed (cont'd.):
  - Nested block
    - Identifier accessible from declaration to end of block in which it is declared
    - Within nested blocks if no identifier with same name exists
  - Scope of function name similar to scope of identifier declared outside any block
    - i.e., function name scope = global variable scope

Scope of an Identifier (cont'd.)

- Some compilers initialize global variables to default values
- Scope resolution operator in C++ is `::`
- By using the scope resolution operator
  - A global variable declared before the definition of a function (or block) can be accessed by the function (or block)
  - Even if the function (or block) has an identifier with the same name as the global variable

Scope of an Identifier (cont'd.)

- To access a global variable declared after the definition of a function, the function must not contain any identifier with the same name
  - Reserved word `extern` indicates that a global variable has been declared elsewhere

Scope of an Identifier Example

```
const double RATE = 10.50;
int z;
double t;
void one(int x, char y);
void two(int a, int b, char x);
void three(int one, double y, int z);

int main()
{
     int num, first;
     double x, y, z;
     char name, last;
        .
        .
        .
     return 0;
}

void one(int x, char y)
{
        .
        .
        .
}
```

Scope of an Identifier Example (Cont'd)

```
int w;

void two(int a, int b, char x)
{
    int count;
        .
        .
}

void three(int one, double y, int z)
{
    char ch;
    int a;
        .
        .
    //Block four
    {
         int x;
         char a;
             .
    } //end Block four
        .
        .
}
```

Global Variables, Named Constants, and Side Effects

- Using global variables causes side effects
- A function that uses global variables is not independent
- If more than one function uses the same global variable:
  - Can be difficult to debug problems with it
  - Problems caused in one area of the program may appear to be from another area
- Global named constants have no side effects

Global Variables Example

```
int t;

void funOne(int& a);

int main()
{
    t = 15;                                                    //Line 1

    cout << "Line 2: In main: t = " << t << endl;              //Line 2

    funOne(t);                                                 //Line 3

    cout << "Line 4: In main after funOne: "
         << " t = " << t << endl;                              //Line 4

    return 0;                                                  //Line 5
}
```

Global Variables Example (cont'd)

```
void funOne(int& a)
{
    cout << "Line 6: In funOne: a = " << a
         << " and t = " << t << endl;                    //Line 6

    a = a + 12;                                          //Line 7
    cout << "Line 8: In funOne: a = " << a
         << " and t = " << t << endl;                    //Line 8

    t = t + 13;                                          //Line 9

    cout << "Line 10: In funOne: a = " << a
         << " and t = " << t << endl;                    //Line 10
}
```

Static and Automatic Variables

- Automatic variable: memory is allocated at block entry and deallocated at block exit
  - By default, variables declared within a block are automatic variables
- Static variable: memory remains allocated as long as the program executes
  - Global variables declared outside of any block are static variables

Static and Automatic Variables (cont'd.)

- Can declare a static variable within a block by using the reserved word static
- Syntax:

```
static dataType identifier;
```

- Static variables declared within a block are local to the block
  - Have same scope as any other local identifier in that block

Static and Automatic Variables Example

```
void test()
{
    static int x = 0;
    int y = 10;

    x = x + 2;
    y = y + 1;

    cout << "Inside test x = " << x << " and y = "
         << y << endl;
}

int main()
{
    int count;
    for (count = 1; count <= 5; count++)
        test();

    return 0;
}
```

Debugging: Using Drivers and Stubs

- Driver program: separate program used to test a function
- When results calculated by one function are needed in another function, use a function stub
- Function stub: a function that is not fully coded

Function Overloading: An Introduction

- In a C++ program, several functions can have the same name
- Function overloading: creating several functions with the same name
- Function signature: the name and formal parameter list of the function
  - Does *not* include the return type of the function

Function Overloading (cont'd.)

- Two functions are said to have different formal parameter lists if both functions have either:
  - A different number of formal parameters
  - If the number of formal parameters is the same, but the data type of the formal parameters differs in at least one position
- Overloaded functions must have different function signatures

Function Overloading (cont'd.)

```
int largerInt(int x, int y);
char largerChar(char first, char second);
double largerDouble(double u, double v);
string largerString(string first, string second);
```

```
int larger(int x, int y);
char larger(char first, char second);
double larger(double u, double v);
string larger(string first, string second);
```

Function Overloading (cont'd.)

- The parameter list supplied in a call to an overloaded function determines which function is executed

Functions with Default Parameters

- In a function call, the number of actual and formal parameters must be the same
  - C++ relaxes this condition for functions with default parameters
- Can specify the value of a default parameter in the function prototype
- If you do not specify the value for a default parameter when calling the function, the default value is used

Functions with Default Parameters (cont'd.)

- All default parameters must be the rightmost parameters of the function
- If a default parameter value is not specified:
  - You must omit all of the arguments to its right
- Default values can be constants, global variables, or function calls
- Cannot assign a constant value as a default value to a reference parameter

Functions with Default Parameters (cont'd.)

Consider the following function prototype:

```
void funcExp(int x, int y, double t, char z = 'A', int u = 67,
             char v = 'G', double w = 78.34);
```

Suppose you have the following statements:

```
int a, b;
char ch;
double d;
```

The following function calls are legal:

1. `funcExp(a, b, d);`
2. `funcExp(a, 15, 34.6, 'B', 87, ch);`
3. `funcExp(b, a, 14.56, 'D');`

The following function calls are illegal:

1. `funcExp(a, 15, 34.6, 46.7);`
2. `funcExp(b, 25, 48.76, 'D', 4567, 78.34);`

Recursive Function

- Recursive Function is the function that call it self
- General formula

```
void recursion() {
  recursion(); /* function calls itself */
}
int main() {
   recursion();
}
```

- Recursive functions are very useful in solving many mathematical problems, e.g. calculating the factorial of a number, generating Fibonacci series

Recursive Function (cont'd)

- Incorrect use of recursive functions might lead to infinite loop
- To avoid infinite running of recursive function, there are two properties that a recursive function must have −
  - Base criteria − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
  - Progressive approach − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Recursive Function (Number factorial Example)

```
int factorial(int i) {
     if(i <= 1)
               return 1;
     return i * factorial(i - 1);
}

int  main() {
     int i = 5;
     cout<<"Factorial of "<< i<<" is "<< factorial(i);
     return 0;
}
```

Recursive Function (Number factorial Example)

```
int fibonacci(int i) {
     if(i == 0)
           return 0;
          if(i == 1)
               return 1;
     return fibonacci(i-1) + fibonacci(i-2);
}
int  main() {
     int i;
     for (i = 0; i < 10; i++)
     {
          cout<< fibonacci(i)<<endl;
     }
     return 0;
}
```

Recursive Function (is Palindrome Example)

```
bool isPalindrome(int start, int end, string str) {
     if(start - end >= 0)
          return true;
     if(str[start] != str[end])
          return false;
     return isPalindrome(start+1,end-1,str);
}

int main(){
     string str = "nursesrun";
     cout<<isPalindrome(0,str.length()-1,str)<<endl;
     str =   "abccba";
     cout<<isPalindrome(0,str.length()-1,str)<<endl;
     str =   "abccdba";
     cout<<isPalindrome(0,str.length()-1,str)<<endl;
}
```

Arrays as Parameters to Functions

- Arrays are passed by reference only
- Do not use symbol `&` when declaring an array as a formal parameter
- Size of the array is usually omitted
  - If provided, it is ignored by the compiler
- Example:

```
void funcArrayAsParam(int listOne[],
double listTwo[])
```

Arrays as Parameters to Functions (cont'd)

```
void initialize(int list[], int listSize)
{
   int count;
   for (count = 0; count < listSize; count++)
      list[count] = 0;
}
```

Constant Arrays as Formal Parameters

- Can prevent a function from changing the actual parameter when passed by reference
  - Use `const` in the declaration of the formal parameter
- Example:

```
void example(int x[], const int y[], int sizeX, int sizeY)
```

Constant Arrays as Formal Parameters Example

```
void initializeArray(int list[], int listSize)
{
    int index;
    for (index = 0; index < listSize; index++)
        list[index] = 0;
}

void fillArray(int list[], int listSize)
{
    int index;
    for (index = 0; index < listSize; index++)
        cin >> list[index];
}
```

Constant Arrays as Formal Parameters Example (cont'd)

```
void printArray(const int list[], int listSize)
{
    int index;
    for (index = 0; index < listSize; index++)
        cout << list[index] << " ";
}

int sumArray(const int list[], int listSize)
{
    int index;
    int sum = 0;
    for (index = 0; index < listSize; index++)
        sum = sum + list[index];
    return sum;
}
```

Constant Arrays as Formal Parameters Example (cont'd)

```
int indexLargestElement(const int list[], int listSize)
{
    int index;
    int maxIndex = 0;
    for (index = 1; index < listSize; index++)
            if (list[maxIndex] < list[index])
                maxIndex = index;
    return maxIndex;
}

void copyArray(int list1[], int src, int list2[],
    int tar, int numOfElements)
{
    for (int index = src; index < src + numOfElements; index++)
    {
        list2[index] = list1[tar];
        tar++;
    }
}
```

Functions Cannot Return a Value of the Type Array

- **C++ does not allow functions to return a value of type array**

Passing Two-Dimensional Arrays as Parameters to Functions

- Two-dimensional arrays are passed by reference as parameters to a function
  - Base address is passed to formal parameter
- Two-dimensional arrays are stored in row order
- When declaring a two-dimensional array as a formal parameter, can omit size of first dimension, but not the second

Passing Two-Dimensional Arrays as Parameters to Functions (cont'd)

Suppose we have the following declaration:

```
const int NUMBER_OF_ROWS = 6;
const int NUMBER_OF_COLUMNS = 5;
```

Consider the following definition of the function printMatrix:

```
void printMatrix(int matrix[][NUMBER_OF_COLUMNS],
                 int noOfRows)
{
    int row, col;

    for (row = 0; row < noOfRows; row++)
    {
        for (col = 0; col < NUMBER_OF_COLUMNS; col++)
            cout << setw(5) << matrix[row][col] << " ";

        cout << endl;
    }
}
```

Chapter 8
Arrays and Strings

Outline

- In this chapter, you will study:
  - Arrays
  - Searching an Array for a Specific Item
  - C-Strings (Character Arrays)
  - Parallel Arrays
  - Two- and Multidimensional Arrays

Introduction

- Simple data type: variables of these types can store only one value at a time
- Structured data type: a data type in which each data item is a collection of other data items

Arrays

- Array: a collection of a fixed number of components, all of the same data type
- One-dimensional array: components are arranged in a list form
- Syntax for declaring a one-dimensional array:

```
dataType arrayName[intExp];
```

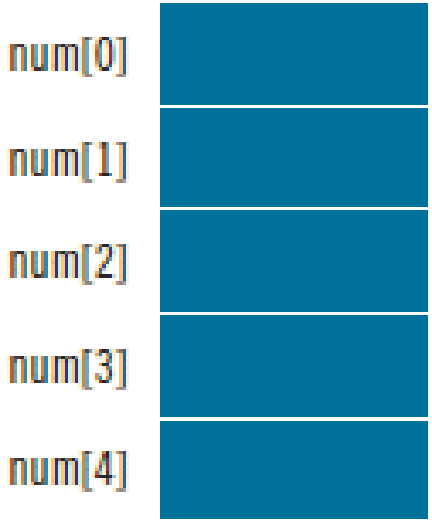
- `intExp`: any constant expression that evaluates to a positive integer

Example 8-1

- The statement:

```
int num[5];
```

- declares an array **num** of five components. Each component is of type int. The components are num[0], num[1] num[2], num[3], and num[4].



Accessing Array Components

- General syntax:

```
arrayName[indexExp]
```

- `indexExp`: called the index
  - An expression with a nonnegative integer value
- Value of the index is the position of the item in the array
- `[]`: array subscripting operator
  - Array index always starts at 0

Accessing Array Components (cont'd.)

```
int list[10];
```

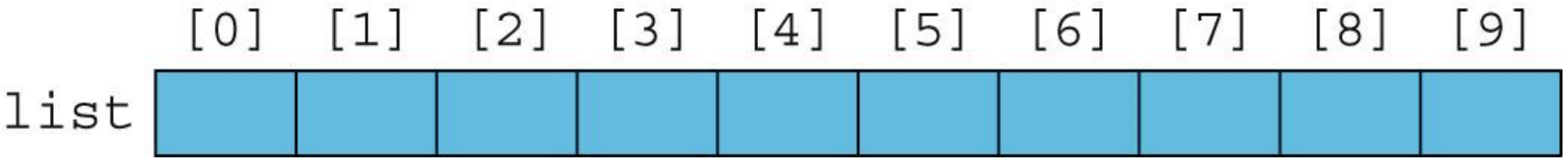


FIGURE 8-3 Array `list`

```
list[5] = 34;
```

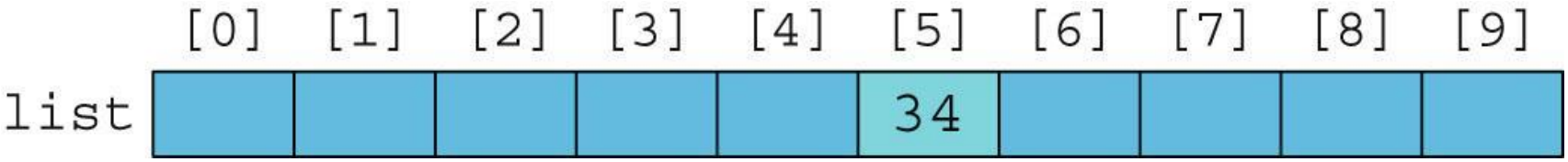


FIGURE 8-4 Array `list` after execution of the statement `list[5]= 34;`

Accessing Array Components (cont'd.)

```
list[3] = 10;
list[6] = 35;
list[5] = list[3] + list[6];
```

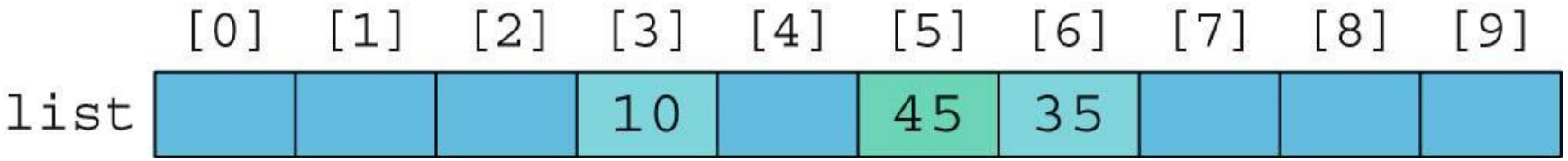


FIGURE 8-5 Array `list` after execution of the statements `list[3]= 10;`, `list[6]= 35;`, and `list[5] = list[3] + list[6];`

Processing One-Dimensional Arrays

- Basic operations on a one-dimensional array:
  - Initializing
  - Inputting data
  - Outputting data stored in an array
  - Finding the largest and/or smallest element
- Each operation requires ability to step through elements of the array
  - Easily accomplished by a loop

Processing One-Dimensional Arrays (cont'd.)

- Given the declaration:

```
int list[100];  //array of size 100
int i;
```

- Use a `for` loop to access array elements:

```
for (i = 0; i < 100; i++)    //Line 1
     cin >> list[i];         //Line 2
```

Example 8-3

- Write the required code to do the following:

1. Define an array `sales` of 10 components of type `double`.

```
double sales[10];
```

- initializes every component of the array `sales` to 0.0

```
for (int index = 0; index < 10; index++)
     sales[index] = 0.0;
```

Example 8-3 (cont'd)

3. Reading data from user into an array:

```
for (index = 0; index < 10; index++)
     cin >> sales[index];
```

4. Printing an array

```
for (index = 0; index < 10; index++)
     cout << sales[index] << " ";
```

Example 8-3 (cont'd)

5. Finding the sum and average of an array

```
double sum = 0;
for (index = 0; index < 10; index++)
   sum = sum + sales[index];
double average = sum / 10;
```

Example 8-3 (cont'd)

6. Largest element in the array:

```
double maxIndex = 0;
for (index = 1; index < 10; index++)
  if (sales[maxIndex] < sales[index])
    maxIndex = index;
largestSale = sales[maxIndex];
```

Array Index Out of Bounds

- Index of an array is in bounds if the index is `>=0` and `<= ARRAY_SIZE-1`
  - Otherwise, the index is out of bounds
- In C++, there is no guard against indices that are out of bounds

Array Initialization During Declaration

- Arrays can be initialized during declaration
  - Values are placed between curly braces
  - Size determined by the number of initial values in the braces
- Example:

```
double sales[] = {12.25, 32.50, 16.90,
23, 45.68};
```

Partial Initialization of Arrays During Declaration

- The statement:

```
int list[10] = {0};
```

  - Declares an array of 10 components and initializes all of them to zero

- The statement:

```
int list[10] = {8, 5, 12};
```

  - Declares an array of 10 components and initializes `list[0]` to 8, `list[1]` to 5, `list[2]` to 12
  - All other components are initialized to 0

Some Restrictions on Array Processing

- Aggregate operation: any operation that manipulates the entire array as a single unit is Not allowed on arrays in C++
- Example:

```
int myList[5] = {0, 4, 8, 12, 16};  //Line 1
int yourList[5];  //Line 2

yourList = myList;  //illegal

cin >> yourList; //illegal
```

- Solution:

```
for (int index = 0; index < 5; index ++)
     yourList[index] = myList[index];
```

Base Address of an Array and Array in Computer Memory

- Base address of an array: address (memory location) of the first array component
- Example:
  - If `list` is a one-dimensional array, its base address is the address of `list[0]`
- When an array is passed as a parameter, the base address of the actual array is passed to the formal parameter

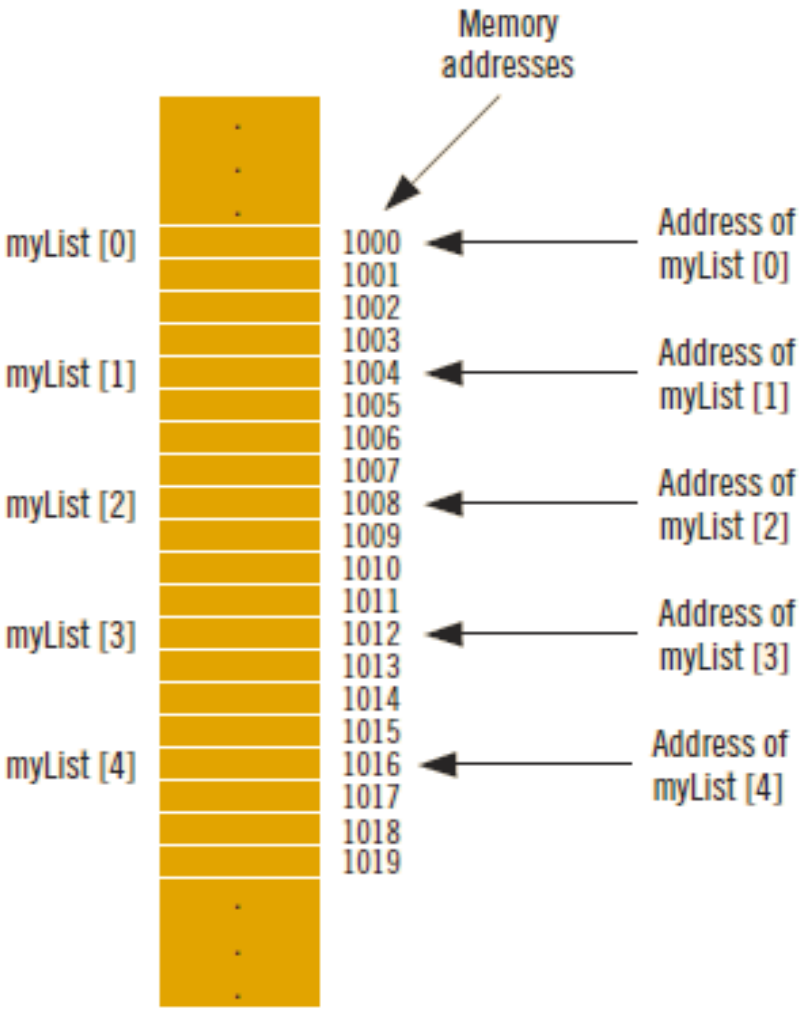
Base Address of an Array and Array in Computer Memory

What is the output of the following statements?

```
cout << myList;

if (myList <= yourList)
{

}
```



Other Ways to Declare Arrays

- Examples:

```
const int NO_OF_STUDENTS = 20;
int testScores[NO_OF_STUDENTS];
```

```
const int SIZE = 50;                    //Line 1
typedef double list[SIZE];              //Line 2

list yourList;                          //Line 3
list myList;                            //Line 4
```

Searching an Array for a Specific Item

- Sequential search (or linear search):
  - Searching a list for a given item, starting from the first array element
  - Compare each element in the array with value being searched for
  - Continue the search until item is found or no more data is left in the list

Pseudocode for Searching an Array for a Specific Item

```
int seqSearch(const int list[], int listLength, int searchItem)
{
    int loc = 0;
    bool found = false;
    while (loc < listLength && !found)
        if (list[loc] == searchItem)
            found = true;
        else
            loc++;
    if (found)
        return loc;
    else
        return -1;
}
```

Selection Sort

- Selection sort: rearrange the list by selecting an element and moving it to its proper position
- Steps:
  - Find the smallest element in the unsorted portion of the list
  - Move it to the top of the unsorted portion by swapping with the element currently there
  - Start again with the rest of the list

Selection Sort (cont'd.)

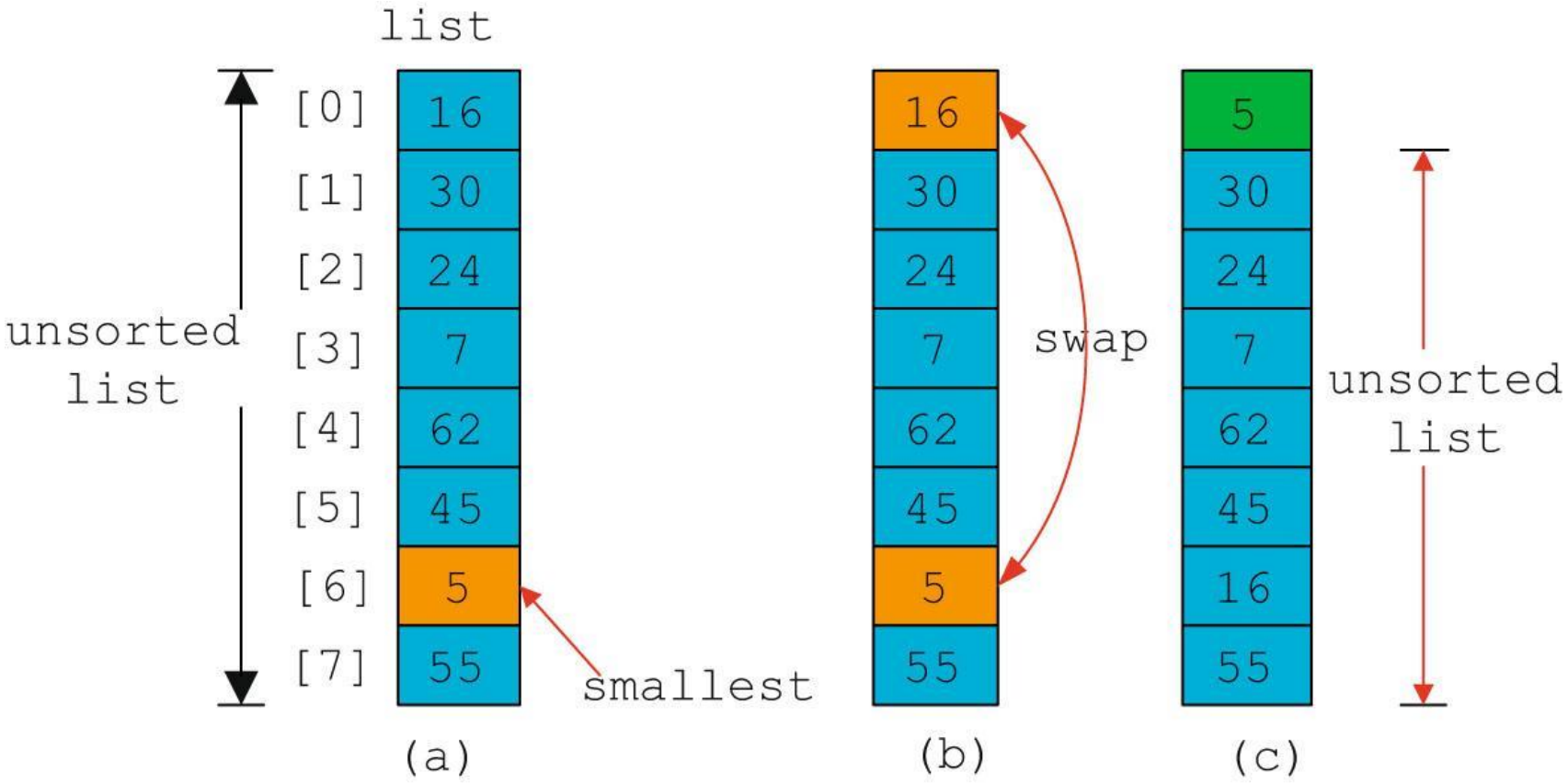


FIGURE 8-10 Elements of `list` during the first iteration

Selection Sort (cont'd.)

```
void selectionSort(int list[], int length)
{
    int index;
    int smallestIndex;
    int location;
    int temp;
    for (index = 0; index < length - 1; index++)
    {
        smallestIndex = index;
        for (location = index + 1; location < length; location++)
            if (list[location] < list[smallestIndex])
                smallestIndex = location;

        temp = list[smallestIndex];
        list[smallestIndex] = list[index];
        list[index] = temp;
    }
}
```

C-Strings (Character Arrays)

- Character array: an array whose components are of type `char`
- C-strings are null-terminated (`'\0'`) character arrays
- Example:
  - `'A'` is the character `A`
  - `"A"` is the C-string `A`
  - `"A"` represents two characters, `'A'` and `'\0'`

C-Strings (Character Arrays) (cont'd.)

- Example:

```
char name[16];
```

- Since C-strings are null terminated and `name` has 16 components, the largest string it can store has 15 characters
- If you store a string whose length is less than the array size, the last components are unused

C-Strings (Character Arrays) (cont'd.)

- Size of an array can be omitted if the array is initialized during declaration
- Example:

```
char name[] = "John";
```

  - Declares an array of length 5 and stores the C-string "John" in it
- Useful string manipulation functions
  - `strcpy`, `strcmp`, and `strlen`

C-Strings (Character Arrays) (cont'd.)

TABLE 8-1 `strcpy`, `strcmp`, and `strlen` Functions

Function	Effect
`strcpy(s1, s2)`	Copies the string `s2` into the string variable `s1` The length of `s1` should be at least as large as `s2`
`strcmp(s1, s2)`	Returns a value < `0` if `s1` is less than `s2` Returns `0` if `s1` and `s2` are the same Returns a value > `0` if `s1` is greater than `s2`
`strlen(s)`	Returns the length of the string `s`, excluding the null character

String Comparison

- C-strings are compared character by character using the collating sequence of the system
  - Use the function `strcmp`
- If using the ASCII character set:
  - "Air" < "Boat"
  - "Air" < "An"
  - "Bill" < "Billy"
  - "Hello" < "hello"

String Comparison Example

Suppose you have the following statements:

```
char studentName[21];
char myname[16];
char yourname[16];
```

The following statements show how string functions work:

Statement	Effect
`strcpy(myname, "John Robinson");`	`myname = "John Robinson"`
`strlen("John Robinson");`	Returns 13, the length of the string `"John Robinson"`

String Comparison Example (cont'd)

```
int len;
len = strlen("Sunny Day");

strcpy(yourname, "Lisa Miller");
strcpy(studentName, yourname);

strcmp("Bill", "Lisa");

strcpy(yourname, "Kathy Brown");
strcpy(myname, "Mark G. Clark");
strcmp(myname, yourname);
```

Stores 9 into len

yourname = "Lisa Miller"
studentName = "Lisa Miller"

Returns a value < 0

yourname = "Kathy Brown"
myname = "Mark G. Clark"
Returns a value > 0

Reading and Writing Strings

- Most rules for arrays also apply to C-strings (which are character arrays)
- Aggregate operations, such as assignment and comparison, are not allowed on arrays
- C++ does allow aggregate operations for the input and output of C-strings

String Input

- Example:

```
cin >> name;
```

  – Stores the next input C-string into name

- To read strings with blanks, use `get` function:

```
cin.get(str, m+1);
```

  – Stores the next `m` characters into `str` but the newline character is not stored in `str`

  – If input string has fewer than `m` characters, reading stops at the newline character

String Output

- Example:

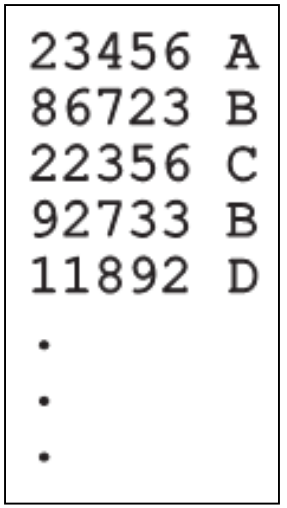
```
cout << name;
```

  - Outputs the content of name on the screen
  - `<<` continues to write the contents of `name` until it finds the null character
  - If `name` does not contain the null character, then strange output may occur
    - `<<` continues to output data from memory adjacent to `name` until a `'\0'` is found

Parallel Arrays

- Two (or more) arrays are called parallel if their corresponding components hold related information
- Example:

```
int studentId[50];
char courseGrade[50];
```



Two- and Multidimensional Arrays

- Two-dimensional array: collection of a fixed number of components (of the same type) arranged in two dimensions
  - Sometimes called matrices or tables
- Declaration syntax:

```
dataType  arrayName[intExp1][intExp2];
```

  - `intExp1` and `intExp2` are expressions with positive integer values specifying the number of rows and columns in the array

Accessing Array Components

- Accessing components in a two-dimensional array:

```
arrayName[indexExp1][indexExp2]
```

  - Where `indexExp1` and `indexExp2` are expressions with positive integer values, and specify the row and column position

- Example:

```
sales[5][3] = 25.75;
```

Accessing Array Components (cont'd.)

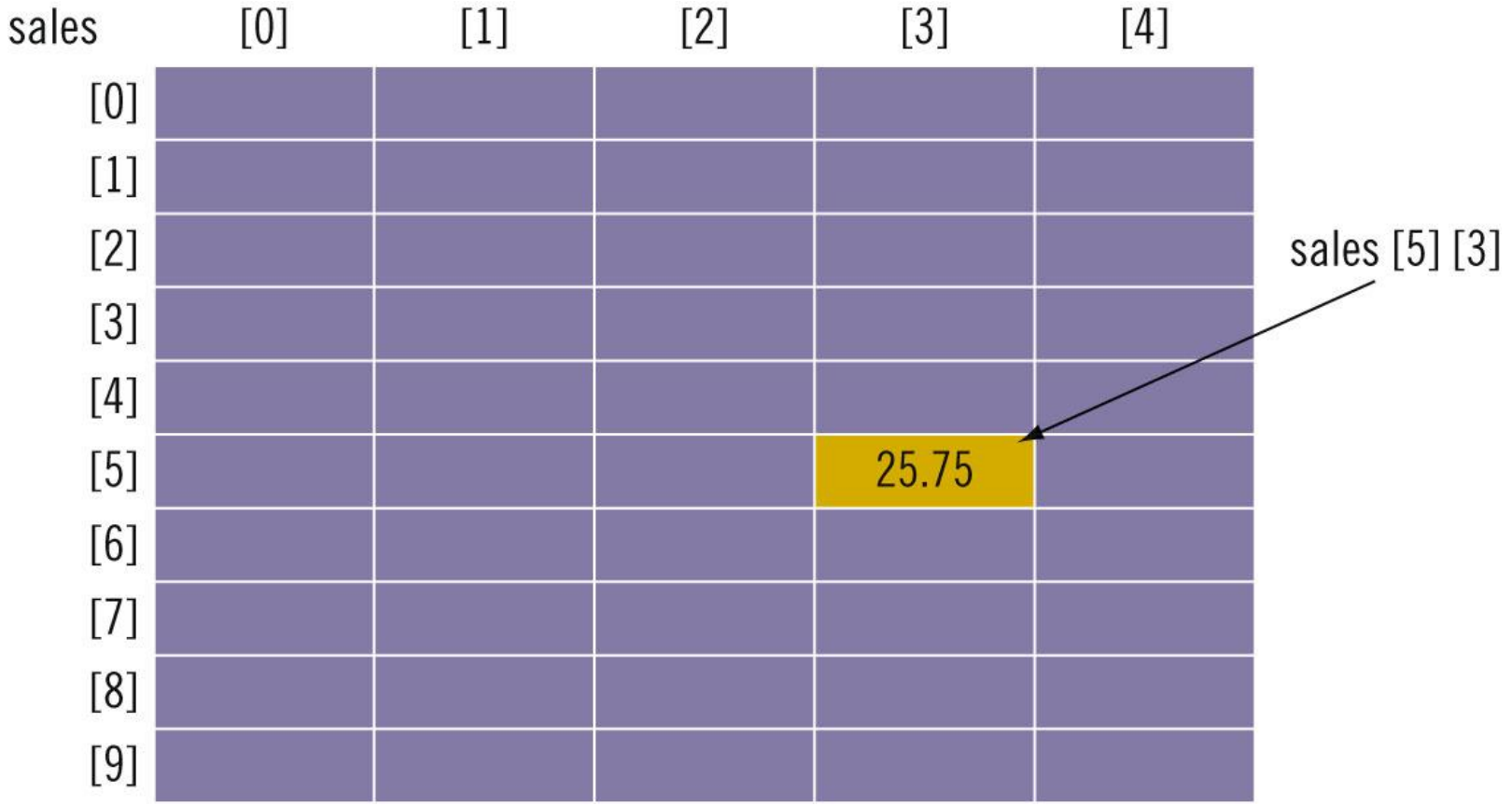


FIGURE 8-14 sales[5][3]

Two-Dimensional Array Initialization During Declaration

- Two-dimensional arrays can be initialized when they are declared:
  - Elements of each row are enclosed within braces and separated by commas
  - All rows are enclosed within braces
  - For number arrays, unspecified elements are set to 0

Two-Dimensional Array Initialization During Declaration

```
int board[4][3] = {{2, 3, 1},
                   {15, 25, 13},
                   {20, 4, 7},
                   {11, 18, 14}};
```

board	[0]	[1]	[2]
[0]	2	3	1
[1]	15	25	13
[2]	20	4	7
[3]	11	18	14

Processing Two-Dimensional Arrays

- Ways to process a two-dimensional array:
  - Process entire array
  - Row processing: process a single row at a time
  - Column processing: process a single column at a time
- Each row and each column of a two-dimensional array is a one-dimensional array
  - To process, use algorithms similar to processing one-dimensional arrays

Initialization

- Examples:
  - To initialize row number 4 (fifth row) to 0:

```
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    matrix[row][col] = 0;
```

  - To initialize the entire matrix to 0:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        matrix[row][col] = 0;
```

Print

- Use a nested loop to output the components of a two dimensional array:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cout << setw(5) << matrix[row][col] << " ";

    cout << endl;
}
```

Input

- Examples:
  - To input into row number 4 (fifth row):

```
row = 4;

for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    cin >> matrix[row][col];
```

  - To input data into each component of matrix:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cin >> matrix[row][col];
```

Sum by Row

- Example:
  - To find the sum of row number 4:

```
sum = 0;
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    sum = sum + matrix[row][col];
```

Sum by Column

- Example:
  - To find the sum of each individual column:

```
  //Sum of each individual column
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
{
     sum = 0;
     for (row = 0; row < NUMBER_OF_ROWS; row++)
          sum = sum + matrix[row][col];

     cout << "Sum of column " << col + 1 << " = " << sum
          << endl;
}
```

Largest Element in Each Row and Each Column

- Example:
  - To find the largest element in each row:

```
  //Largest element in each row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    largest = matrix[row][0]; //Assume that the first element
                              //of the row is the largest.
    for (col = 1; col < NUMBER_OF_COLUMNS; col++)
        if (largest < matrix[row][col])
            largest = matrix[row][col];

    cout << "The largest element in row " << row + 1 << " = "
         << largest << endl;
}
```

Arrays of Strings

- Strings in C++ can be manipulated using either the data type `string` or character arrays (`C`-strings)
- On some compilers, the data type `string` may not be available in Standard C++ (i.e., non-ANSI/ISO Standard C++)

Arrays of Strings and the `string` Type

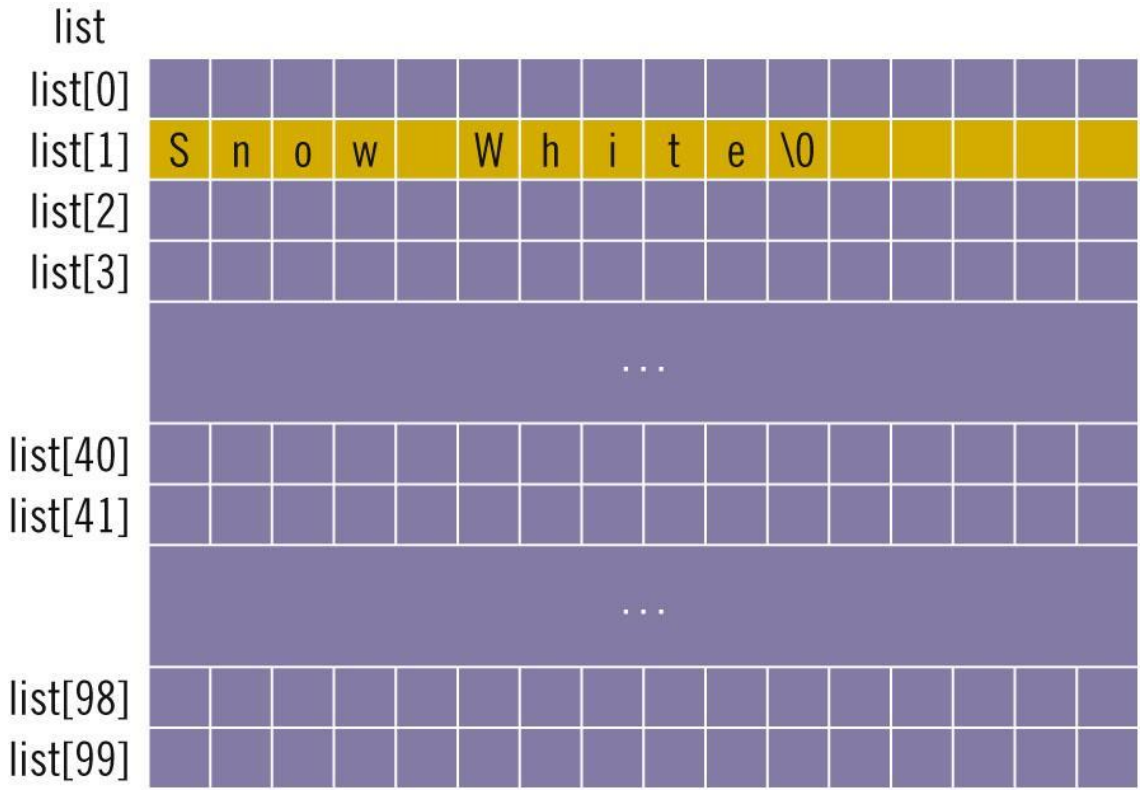
- To declare an array of 100 components of type `string`:

```
string list[100];
```

- Basic operations, such as assignment, comparison, and input/output, can be performed on values of the `string` type
- The data in `list` can be processed just like any one-dimensional array

Arrays of Strings and C-Strings (Character Arrays)

```
char list[100][16];
strcpy(list[1], "Snow White");
```



Arrays of Strings and C-Strings (Character Arrays)

- The following for loop is used to read and store string in each row:

```
for (j = 0; j < 100; j++)
    cin.get(list[j], 16);
```

- The following for loop outputs the string in each row:

```
for (j = 0; j < 100; j++)
    cout << list[j] << endl;
```