

Chapter 1:

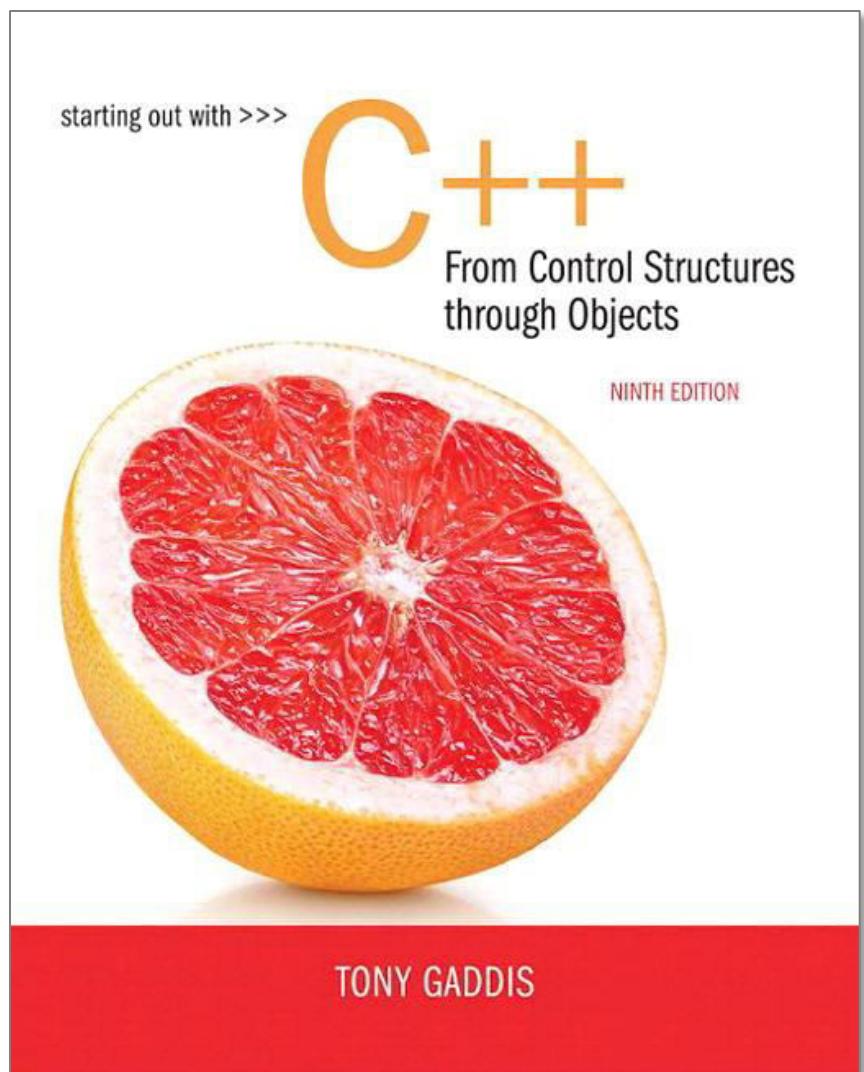
Introduction

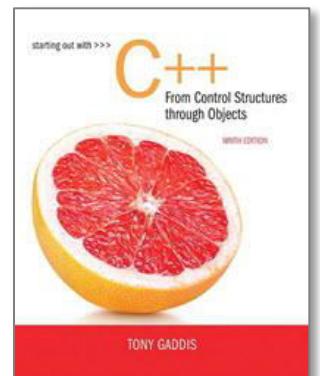
to

Computers

and

Programming





1.1

Why Program?



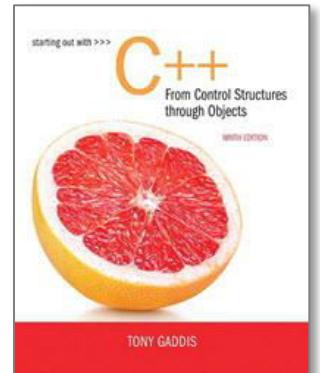
Why Program?

Computer – programmable machine designed to follow instructions

Program – instructions in computer memory to make it do something

Programmer – person who writes instructions (programs) to make computer perform a task

SO, without programmers, no programs; without programs, a computer cannot do anything



1.2

Computer Systems: Hardware and Software



Main Hardware Component Categories:

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



Main Hardware Component Categories

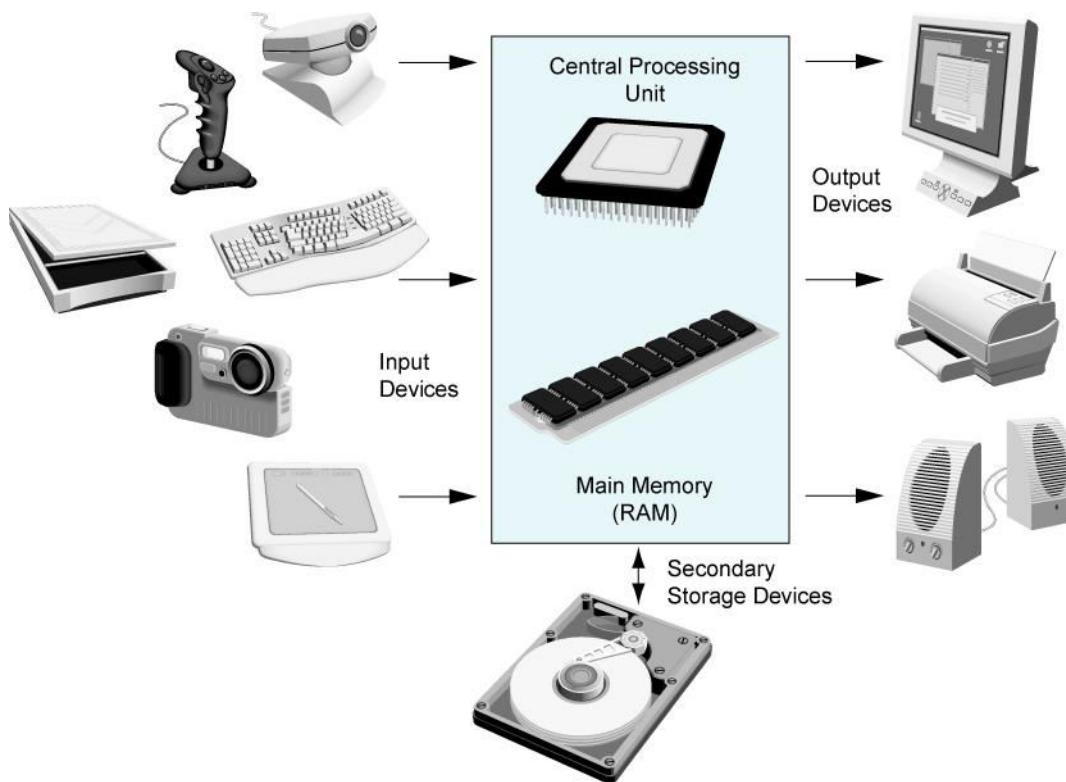


Figure 1-2

Central Processing Unit (CPU)

Comprised of:

Control Unit

Retrieves and decodes program instructions

Coordinates activities of all other parts of computer

Arithmetic & Logic Unit

Hardware optimized for high-speed numeric calculation

Hardware designed for true/false, yes/no decisions



CPU Organization

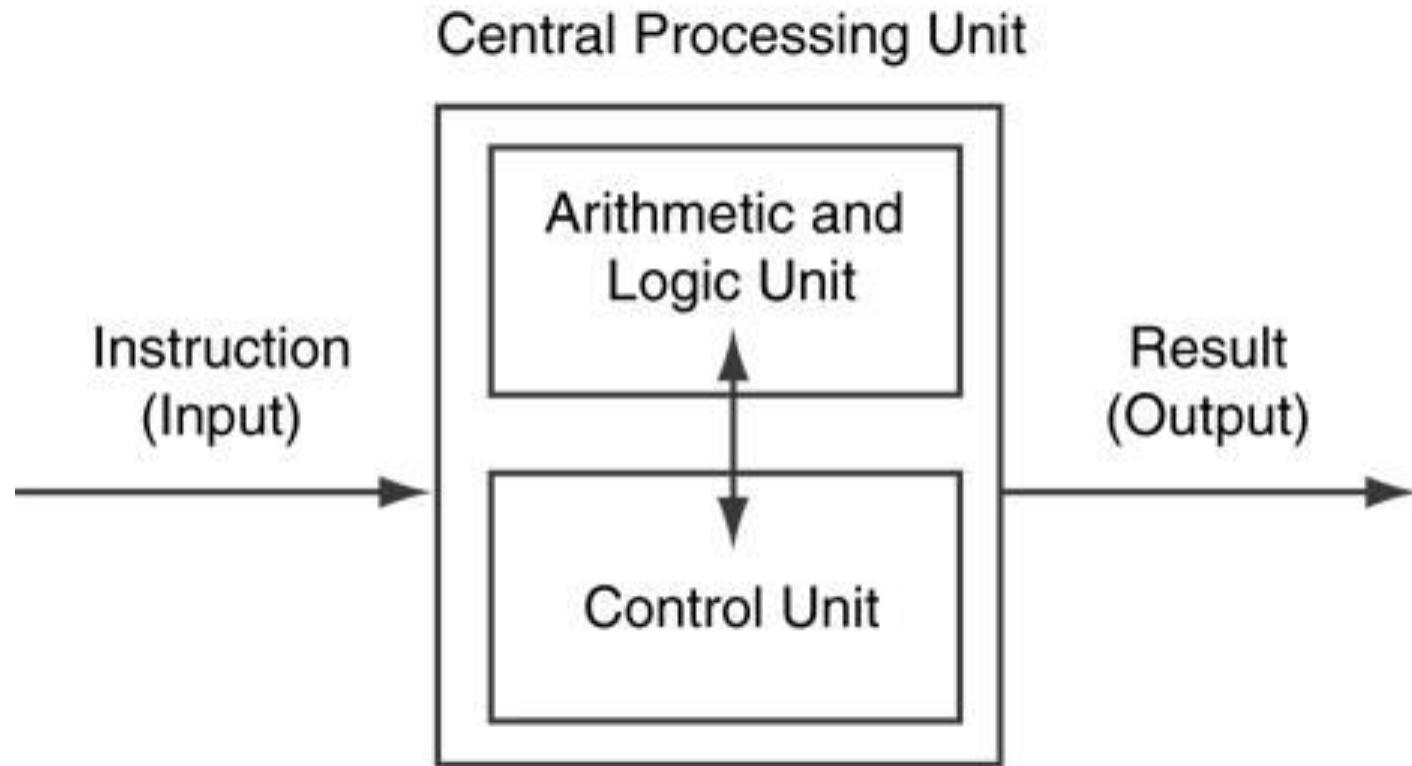


Figure 1-3

Main Memory

- It is volatile. Main memory is erased when program terminates or computer is turned off
- Also called Random Access Memory (RAM)
- Organized as follows:
 - bit: smallest piece of memory. Has values 0 (off, false) or 1 (on, true)
 - byte: 8 consecutive bits. Bytes have addresses.

Main Memory

- Addresses – Each byte in memory is identified by a unique number known as an *address*.

Main Memory

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

- In Figure 1-4, the number 149 is stored in the byte with the address 16, and the number 72 is stored at address 23.

Secondary Storage

- Non-volatile: data retained when program is not running or computer is turned off
- Comes in a variety of media:
 - magnetic: traditional hard drives that use a moveable mechanical arm to read/write
 - solid-state: data stored in chips, no moving parts
 - optical: CD-ROM, DVD
 - Flash drives, connected to the USB port

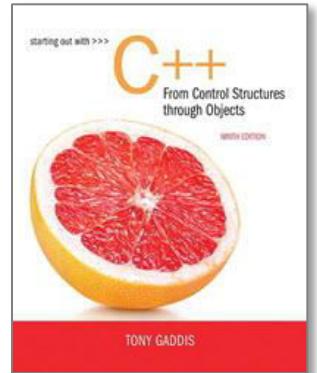
Input Devices

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



Software-Programs That Run on a Computer

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



1.3

Programs and Programming Languages



Programs and Programming Languages

- A program is a set of instructions that the computer follows to perform a task
- We start with an *algorithm*, which is a set of well-defined steps.

Example Algorithm for Calculating Gross Pay

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



Machine Language

- Orange Although the previous algorithm defines the steps for calculating the gross pay, it is not ready to be executed on the computer.
- Orange The computer only executes *machine language* instructions

Machine Language

- Machine language instructions are binary numbers, such as

1011010000000101

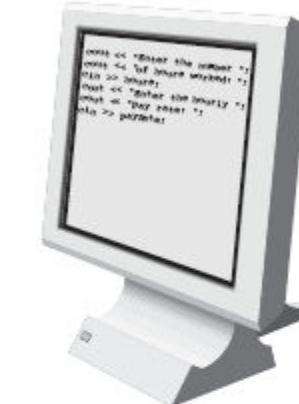
- Rather than writing programs in machine language, programmers use *programming languages*.

Programs and Programming Languages

- Types of languages:

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

High level (Easily read by humans)



Low level (machine language)
10100010 11101011



Some Well-Known Programming Languages (Table 1-1 on Page 10)

BASIC
FORTRAN
COBOL

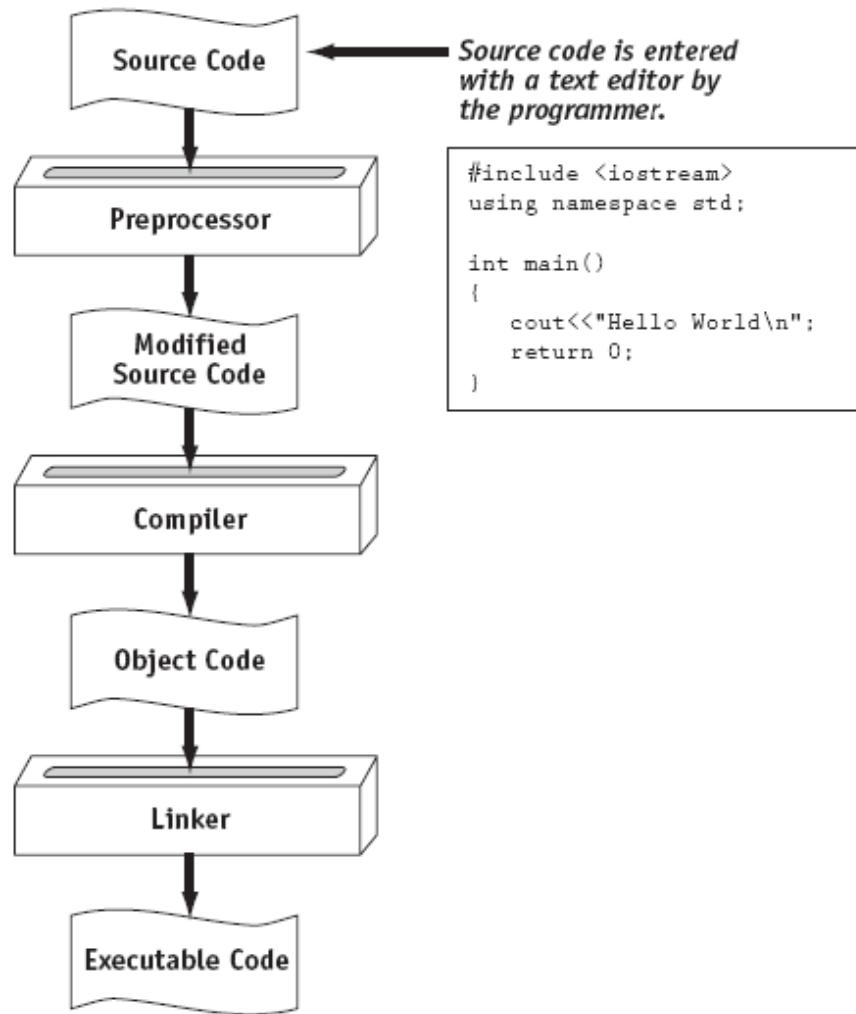


C++
Ruby
Java
Visual Basic
C#
JavaScript
Python

From a High-Level Program to an Executable File

- a) Create file containing the program with a text editor.
 - b) Run preprocessor to convert source file directives to source code program statements.
 - c) Run compiler to convert source program into machine instructions.
 - d) Run linker to connect hardware-specific code to machine instructions, producing an executable file.
- Steps b–d are often performed by a single command or button click.
 - Errors detected at any step will prevent execution of following steps.

From a High-Level Program to an Executable File



Integrated Development Environments (IDEs)

- An integrated development environment, or IDE, combine all the tools needed to write, compile, and debug a program into a single software application.
- Examples are Microsoft Visual C++, Turbo C++ Explorer, CodeWarrior, etc.

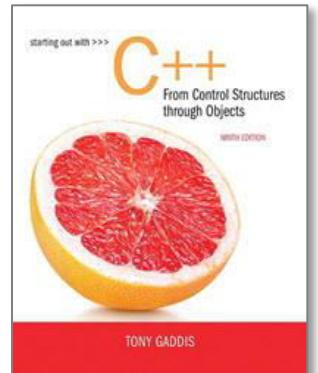


Integrated Development Environments (IDEs)

The screenshot shows the Microsoft Visual Studio interface with the following details:

- Title Bar:** Gross Pay - Microsoft Visual Studio
- Menu Bar:** FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ANALYZE WINDOW HELP
- Toolbar:** Includes icons for New, Open, Save, Print, Find, Copy, Paste, Cut, Undo, Redo, and various debugging symbols.
- Quick Launch:** Local Windows Debugger
- Code Editor:** GrossPay.cpp (Global Scope) - main()

```
// This program calculates the user's pay.  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    double hours, rate, pay;  
  
    // Get the number of hours worked.  
    cout << "How many hours did you work? ";  
    cin >> hours;  
  
    // Get the hourly pay rate.  
    cout << "How much do you get paid per hour? ";  
    cin >> rate;  
  
    // Calculate the pay.  
    pay = hours * rate;  
  
    // Display the pay.  
    cout << "You have earned $" << pay << endl;  
    return 0;  
}
```
- Solution Explorer:** Gross Pay (selected), External Dependencies, Header Files, Resource Files, Source Files (GrossPay.cpp)
- Properties:** main VCCodeFunction (Name: main, File: c:\users\tony\d, FullName: main, IsInjected: False)
- Status Bar:** Ready, Ln 23, Col 2, Ch 2, INS



1.4

What is a Program Made of?

What is a Program Made of?

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



Program 1-1

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



Key Words

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

Key Words

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



Programmer-Defined Identifiers

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



Operators

- Orange Used to perform operations on data
- Orange Many types of operators:
 - Orange Arithmetic - ex: +, -, *, /
 - Orange Assignment – ex: =
- Orange Some operators in Program1-1:

<< >> = *



Operators

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



Punctuation

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

Punctuation

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



Syntax

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



Variables

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

Variable Definitions

- To create a variable in a program you must write a variable definition (also called a variable declaration)
- Here is the statement from Program 1-1 that defines the variables:

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



Variable Definitions

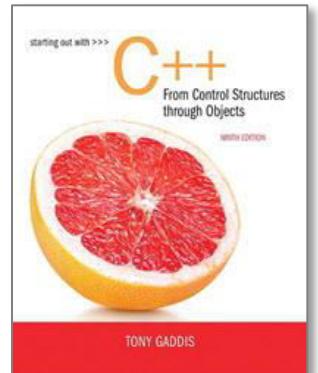
- There are many different types of data, which you will learn about in this course.
- A variable holds a specific type of data.
- The variable definition specifies the type of data a variable can hold, and the variable name.

Variable Definitions

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

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

- The word **double** specifies that the variables can hold double-precision floating point numbers. (You will learn more about that in Chapter 2)



1.5

Input, Processing, and Output



Input, Processing, and Output

Three steps that a program typically performs:

1) Gather input data:

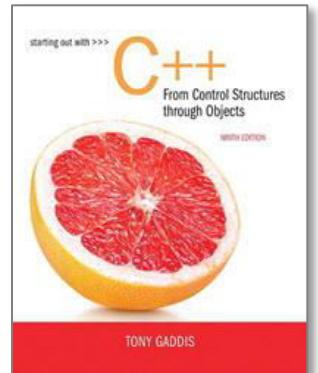
- from keyboard
- from files on disk drives

2) Process the input data

3) Display the results as output:

- send it to the screen
- write to a file





1.6

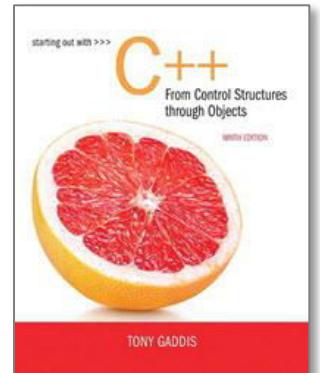
The Programming Process



The Programming Process

1. Clearly define what the program is to do.
2. Visualize the program running on the computer.
3. Use design tools such as a hierarchy chart, flowcharts, or pseudocode to create a model of the program.
4. Check the model for logical errors.
5. Type the code, save it, and compile it.
6. Correct any errors found during compilation. Repeat Steps 5 and 6 as many times as necessary.
7. Run the program with test data for input.
8. Correct any errors found while running the program.
Repeat Steps 5 through 8 as many times as necessary.
9. Validate the results of the program.





1.7

Procedural and Object-Oriented Programming

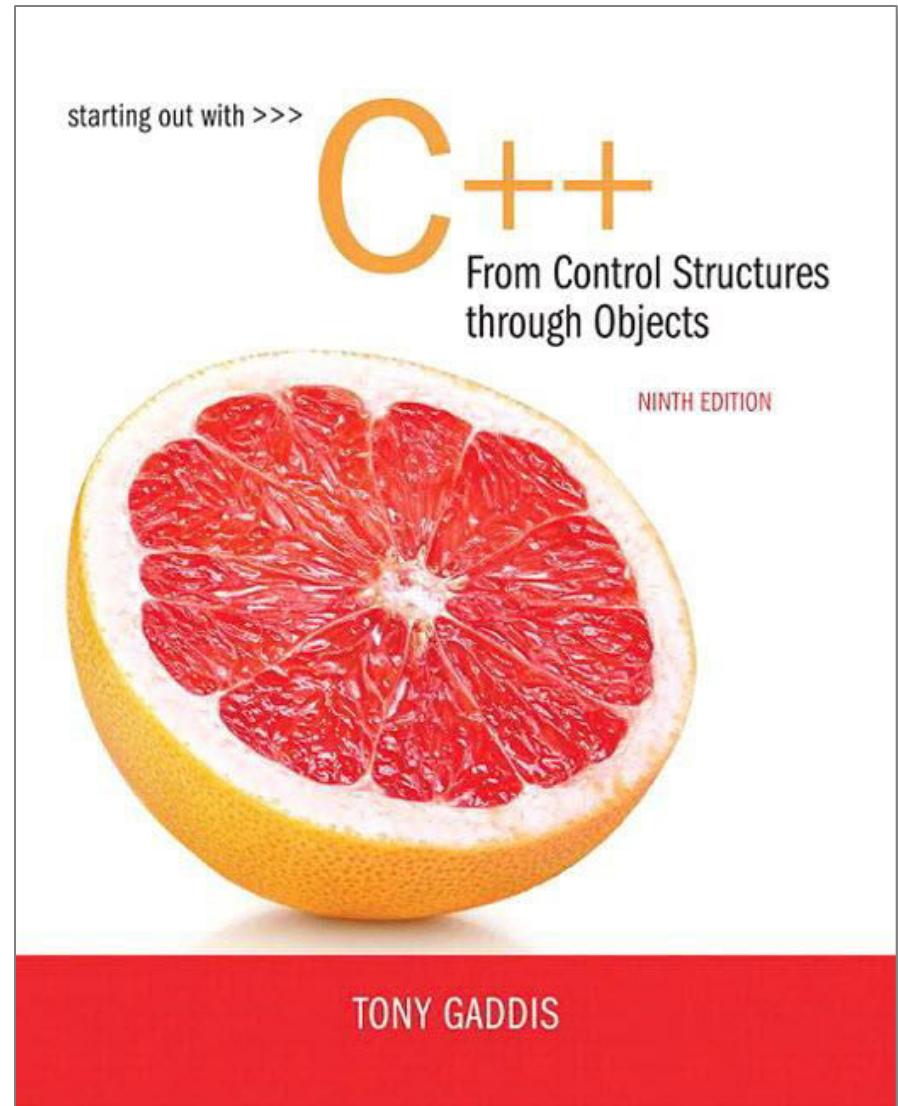


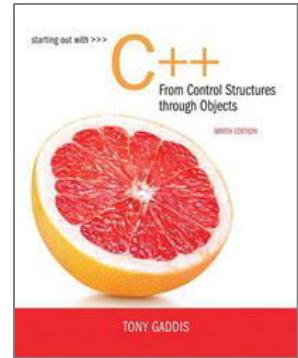
Procedural and Object-Oriented Programming

- Procedural programming: focus is on the process. Procedures/functions are written to process data.
- Object-Oriented programming: focus is on objects, which contain data and the means to manipulate the data. Messages sent to objects to perform operations.

Chapter 2:

Introduction to C++





2.1

The Parts of a C++ Program



The Parts of a C++ Program

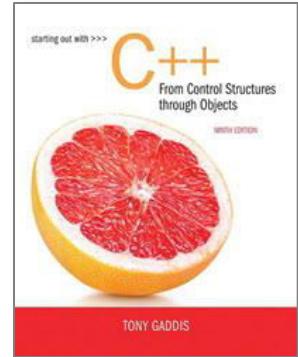
```
// sample C++ program ← comment
#include <iostream> ← preprocessor directive
using namespace std; ← which namespace to use
int main() ← beginning of function named main
{ ← beginning of block for main
    cout << "Hello, there!"; ← output statement
    ↑
    return 0; ← string literal
    ← Send 0 to operating system
} ← end of block for main
```



Special Characters

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





2.2

The cout Object



The cout Object

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

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



The cout Object

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

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

Or:

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

The cout Object

- This produces one line of output:

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



The endl Manipulator

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

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



The endl Manipulator

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



The endl Manipulator

- You do NOT put quotation marks around **endl**
- The last character in **endl** is a lowercase L, not the number 1.

endl ← This is a lowercase L



The \n Escape Sequence

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

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

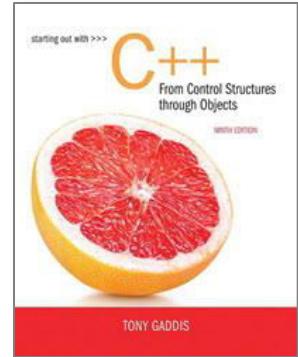
Notice that the `\n` is INSIDE
the string.



The \n Escape Sequence

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





2.3

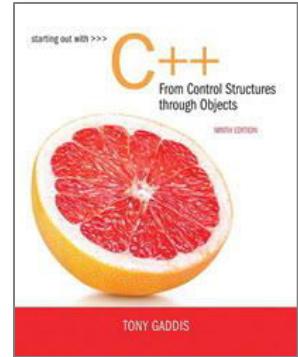
The #include Directive



The #include Directive

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





2.4

Variables and Literals



Pearson

Copyright © 2018, 2015, 2012, 2009 Pearson Education, Inc. All rights reserved.

Variables and Literals

● Variable: a storage location in memory

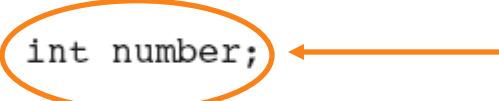
- Has a name and a type of data it can hold
- Must be defined before it can be used:

```
int item;
```



Variable Definition in Program 2-7

Program 2-7

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

Program Output

The value in number is 5



Pearson

Copyright © 2018, 2015, 2012, 2009 Pearson Education, Inc. All rights reserved.

Literals

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

"hello, there" (string literal)

12 (integer literal)



Integer Literal in Program 2-9

Program 2-9

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

Program Output

Today we sold 20 bushels of apples.



String Literals in Program 2-9

Program 2-9

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

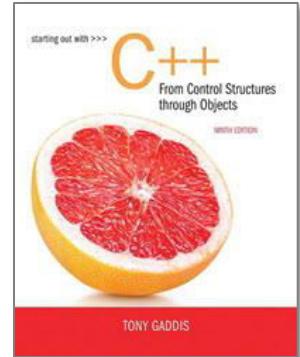
These are string literals

The diagram consists of two orange arrows. One arrow points from the text 'These are string literals' to the string literal 'Today we sold ' located on line 10. Another arrow points from the same text to the string literal ' bushels of apples.' also located on line 10.

Program Output

Today we sold 20 bushels of apples.





2.5

Identifiers



Identifiers

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



C++ Key Words

Table 2-4 The C++ Key Words

alignas	const	for	private	throw
alignof	constexpr	friend	protected	true
and	const_cast	goto	public	try
and_eq	continue	if	register	typedef
asm	decltype	inline	reinterpret_cast	typeid
auto	default	int	return	typename
bitand	delete	long	short	union
bitor	do	mutable	signed	unsigned
bool	double	namespace	sizeof	using
break	dynamic_cast	new	static	virtual
case	else	noexcept	static_assert	void
catch	enum	not	static_cast	volatile
char	explicit	not_eq	struct	wchar_t
char16_t	export	nullptr	switch	while
char32_t	extern	operator	template	xor
class	false	or	this	xor_eq
compl	float	or_eq	thread_local	

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



Variable Names

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

itemsOrdered

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

Identifier Rules

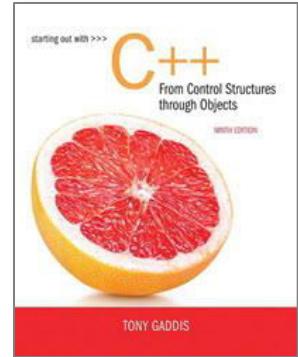
- The first character of an identifier must be an alphabetic character or and underscore (_),
- After the first character you may use alphabetic characters, numbers, or underscore characters.
- Upper- and lowercase characters are distinct



Valid and Invalid Identifiers

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





2.6

Integer Data Types



Integer Data Types

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

Table 2-6 Integer Data Types

Data Type	Typical Size	Typical Range
short int	2 bytes	-32,768 to +32,767
unsigned short int	2 bytes	0 to +65,535
int	4 bytes	-2,147,483,648 to +2,147,483,647
unsigned int	4 bytes	0 to 4,294,967,295
long int	4 bytes	-2,147,483,648 to +2,147,483,647
unsigned long int	4 bytes	0 to 4,294,967,295
long long int	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long int	8 bytes	0 to 18,446,744,073,709,551,615



Defining Variables

- Variables of the same type can be defined

- On separate lines:

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

- On the same line:

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

- Variables of different types must be in different definitions

Integer Types in Program 2-10

```
1 // This program has variables of several of the integer types.  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main()  
6 {  
7     int checking;  
8     unsigned int miles;  
9     long diameter;  
10  
11     checking = -20;  
12     miles = 4276;  
13     diameter = 100000;  
14     cout << "We have made a long journey of " << miles;  
15     cout << " miles.\n";  
16     cout << "Our checking account balance is " << checking;  
17     cout << "\nThe galaxy is about " << diameter;  
18     cout << " light years in diameter.\n";  
19     return 0;  
20 }
```

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



Integer Literals

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

```
itemsOrdered = 15;
```

In this code, 15 is an integer literal.



Integer Literals in Program 2-10

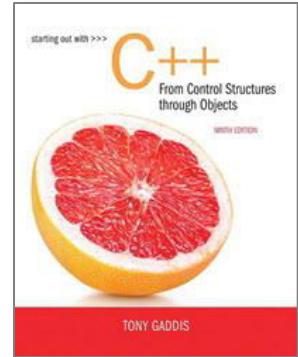
```
1 // This program has variables of several of the integer types.  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main()  
6 {  
7     int checking;  
8     unsigned int miles;  
9     long diameter;  
10    checking = -20;  
11    miles = 4276;  
12    diameter = 100000;  
13    cout << "We have made a long journey of " << miles;  
14    cout << " miles.\n";  
15    cout << "Our checking account balance is " << checking;  
16    cout << "\nThe galaxy is about " << diameter;  
17    cout << " light years in diameter.\n";  
18    return 0;  
19 }  
20 }
```

Integer Literals



Integer Literals

- Integer literals are stored in memory as ints by default
- To store an integer constant in a long memory location, put ‘L’ at the end of the number: 1234L
- To store an integer constant in a long long memory location, put ‘LL’ at the end of the number: 324LL
- Constants that begin with ‘0’ (zero) are base 8: 075
- Constants that begin with ‘0x’ are base 16: 0x75A



2.7

The `char` Data Type



The `char` Data Type

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

CODE:

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

MEMORY:

letter

67

Character Literals

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

'A'



Character Literals in Program 2-14

Program 2-14

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

Program Output

A
B



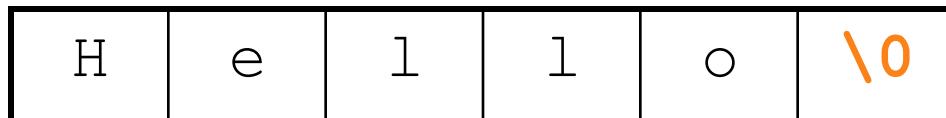
Character Strings

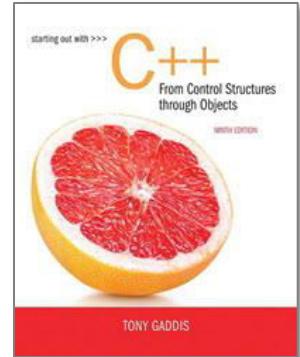
- Orange circle icon A series of characters in consecutive memory locations:

"Hello"

- Orange circle icon Stored with the null terminator, \0, at the end:

- Orange circle icon Comprised of the characters between the " "





2.8

The C++ string Class



The C++ string Class

- Orange Special data type supports working with strings

```
#include <string>
```

- Orange Can define string variables in programs:

```
string firstName, lastName;
```

- Orange Can receive values with assignment operator:

```
firstName = "George";
```

```
lastName = "Washington";
```

- Orange Can be displayed via cout

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



The string class in Program 2-15

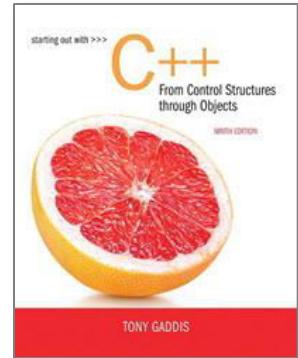
Program 2-15

```
1 // This program demonstrates the string class.  
2 #include <iostream>  
3 #include <string> // Required for the string class.  
4 using namespace std;  
5  
6 int main()  
7 {  
8     string movieTitle;  
9  
10    movieTitle = "Wheels of Fury";  
11    cout << "My favorite movie is " << movieTitle << endl;  
12    return 0;  
13 }
```

Program Output

My favorite movie is Wheels of Fury





2.9

Floating-Point Data Types



Floating-Point Data Types

- The floating-point data types are:

`float`

`double`

`long double`

- They can hold real numbers such as:

12.45

-3.8

- Stored in a form similar to scientific notation

- All floating-point numbers are signed

Floating-Point Data Types

Table 2-8 Floating Point Data Types on PCs

Data Type	Key Word	Description
Single precision	<code>float</code>	4 bytes. Numbers between $\pm 3.4\text{E-}38$ and $\pm 3.4\text{E}38$
Double precision	<code>double</code>	8 bytes. Numbers between $\pm 1.7\text{E-}308$ and $\pm 1.7\text{E}308$
Long double precision	<code>long double*</code>	8 bytes. Numbers between $\pm 1.7\text{E-}308$ and $\pm 1.7\text{E}308$



Floating-Point Literals

- Can be represented in

- Fixed point (decimal) notation:

31.4159

0.0000625

- E notation:

3.14159E1

6.25e-5

- Are double by default

- Can be forced to be float (3.14159f) or long double (0.0000625L)

Floating-Point Data Types in Program 2-16

Program 2-16

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

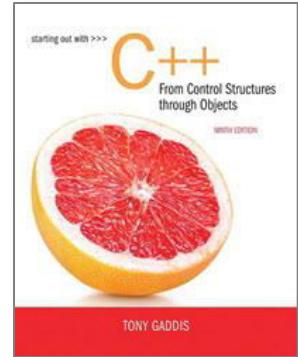
Program Output

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



Pearson

Copyright © 2018, 2015, 2012, 2009 Pearson Education, Inc. All rights reserved.



2.10

The `bool` Data Type



The `bool` Data Type

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

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

```
bool finished = false;
```

1

0

Boolean Variables in Program 2-17

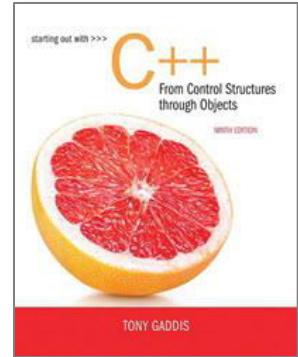
Program 2-17

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

Program Output

```
1  
0
```





2.11

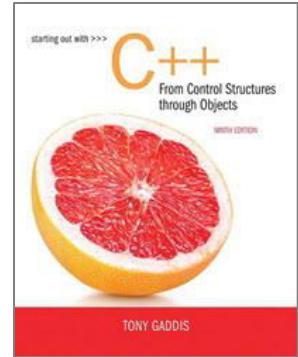
Determining the Size of a Data Type



Determining the Size of a Data Type

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

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



2.12

Variable Assignments and Initialization



Pearson

Copyright © 2018, 2015, 2012, 2009 Pearson Education, Inc. All rights reserved.

Variable Assignments and Initialization

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

```
item = 12;
```

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



Assignment

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

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



Variable Initialization

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

```
int length = 12;
```

- Can initialize some or all variables:

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



Variable Initialization in Program 2-19

Program 2-19

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

Program Output

Month 2 has 28 days.



Declaring Variables With the `auto` Key Word

- C++ 11 introduces an alternative way to define variables, using the `auto` key word and an initialization value. Here is an example:

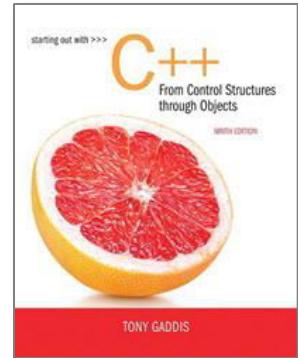
```
auto amount = 100; ← int
```

- The `auto` key word tells the compiler to determine the variable's data type from the initialization value.

```
auto interestRate= 12.0; ← double
```

```
auto stockCode = 'D'; ← char
```

```
auto customerNum = 459L; ← long
```



2.13

Scope



Scope

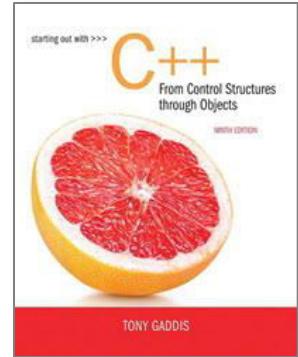
- The scope of a variable: the part of the program in which the variable can be accessed
- A variable cannot be used before it is defined

Variable Out of Scope in Program 2-20

Program 2-20

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





2.14

Arithmetic Operators



Arithmetic Operators

- Used for performing numeric calculations
- C++ has unary, binary, and ternary operators:
 - unary (1 operand) -5
 - binary (2 operands) $13 - 7$
 - ternary (3 operands) $\text{exp1} ? \text{exp2} : \text{exp3}$

Binary Arithmetic Operators

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



Arithmetic Operators in Program 2-21

Program 2-21

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

Program Output

Wages for this week are \$1007.8



Pearson

Copyright © 2010, 2013, 2012, 2009 Pearson Education, Inc. All rights reserved.

A Closer Look at the / Operator

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

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

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

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

```
cout << 13 / 5.0;    // displays 2.6
```

```
cout << 91.0 / 7;    // displays 13.0
```



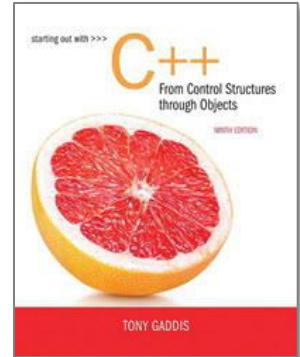
A Closer Look at the % Operator

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

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

 % requires integers for both operands

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



2.15

Comments



Comments

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

Single-Line Comments

- Begin with `//` through to the end of line:

```
int length = 12; // length in  
inches
```

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

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



Multi-Line Comments

- Begin with `/*`, end with `*/`

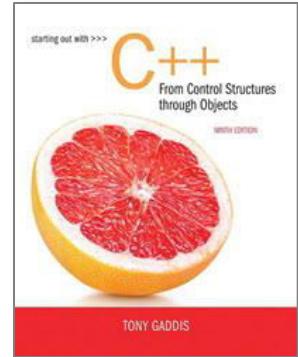
- Can span multiple lines:

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

- Can begin and end on the same line:

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





2.16

Named Constants



Named Constants

- Named constant (constant variable): variable whose content cannot be changed during program execution
- Used for representing constant values with descriptive names:

```
const double TAX_RATE = 0.0675;
```

```
const int NUM_STATES = 50;
```

- Often named in uppercase letters



Named Constants in Program 2-28

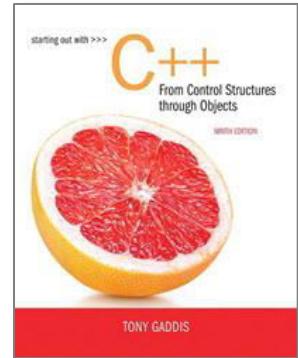
Program 2-28

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

Program Output

The circumference is: 31.4159





2.17

Programming Style



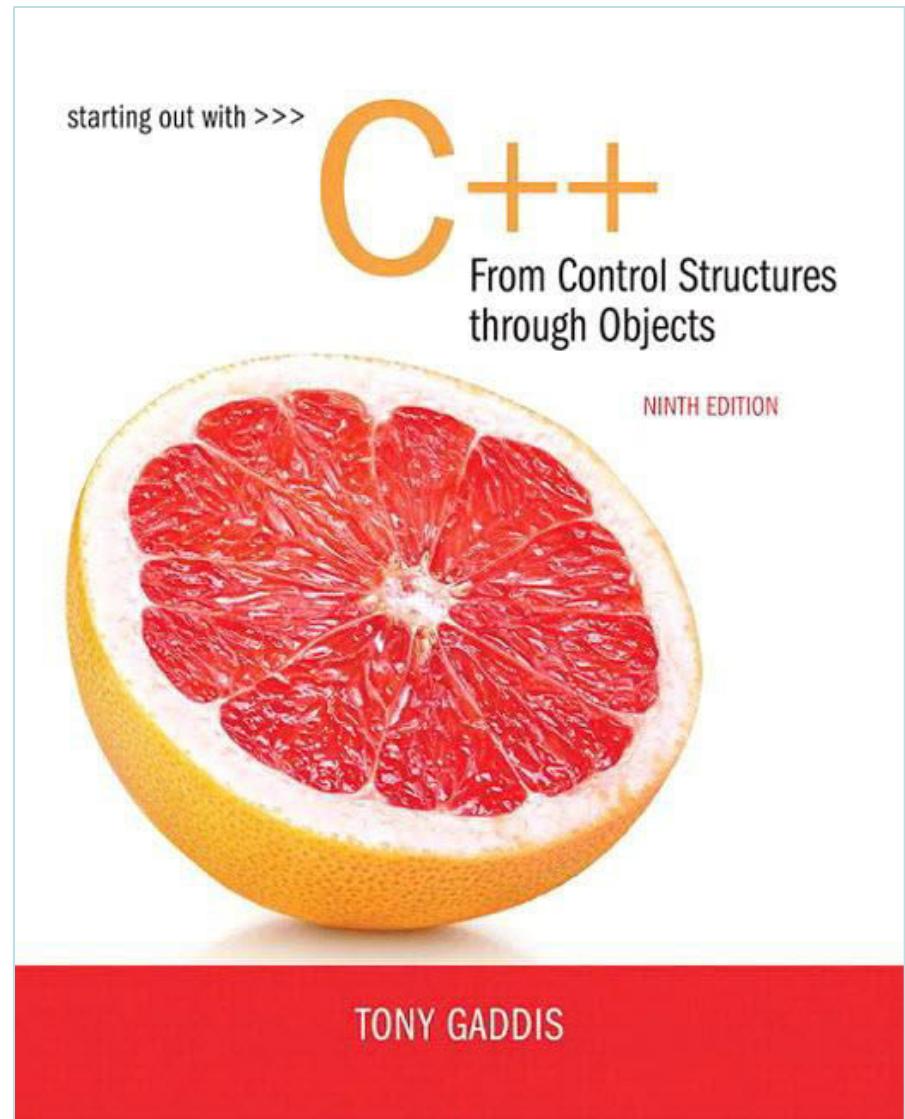
Programming Style

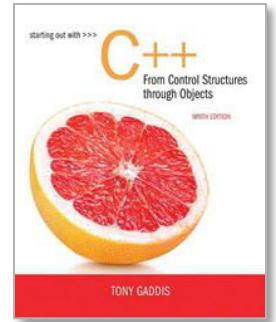
- The visual organization of the source code
- Includes the use of spaces, tabs, and blank lines
- Does not affect the syntax of the program
- Affects the readability of the source code



Chapter 3:

Expressions and Interactivity





3.1

The `cin` Object

The `cin` Object

- Standard input object
- Like `cout`, requires `iostream` file
- Used to read input from keyboard
- Information retrieved from `cin` with `>>`
- Input is stored in one or more variables

The `cin` Object in Program 3-1

Program 3-1

```
1 // This program asks the user to enter the length and width of
2 // a rectangle. It calculates the rectangle's area and displays
3 // the value on the screen.
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     int length, width, area;
10
11    cout << "This program calculates the area of a ";
12    cout << "rectangle.\n";
13    cout << "What is the length of the rectangle? ";
14    cin >> length;
15    cout << "What is the width of the rectangle? ";
16    cin >> width;
17    area = length * width;
18    cout << "The area of the rectangle is " << area << ".\n";
19    return 0;
20 }
```

Program Output with Example Input Shown in Bold

This program calculates the area of a rectangle.

What is the length of the rectangle? **10 [Enter]**

What is the width of the rectangle? **20 [Enter]**

The area of the rectangle is 200.

The `cin` Object

- `cin` converts data to the type that matches the variable:

```
int height;  
cout << "How tall is the room? ";  
cin >> height;
```

Displaying a Prompt

- A prompt is a message that instructs the user to enter data.
- You should always use **cout** to display a prompt before each **cin** statement.

```
cout << "How tall is the room? ";
cin >> height;
```

The `cin` Object

- Orange Can be used to input more than one value:

```
cin >> height >> width;
```

- Orange Multiple values from keyboard must be separated by spaces
- Orange Order is important: first value entered goes to first variable, etc.

The `cin` Object Gathers Multiple Values in Program 3-2

Program 3-2

```
1 // This program asks the user to enter the length and width of
2 // a rectangle. It calculates the rectangle's area and displays
3 // the value on the screen.
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     int length, width, area;
10
11    cout << "This program calculates the area of a ";
12    cout << "rectangle.\n";
13    cout << "Enter the length and width of the rectangle ";
14    cout << "separated by a space.\n";
15    cin >> length >> width;
16    area = length * width;
17    cout << "The area of the rectangle is " << area << endl;
18    return 0;
19 }
```

Program Output with Example Input Shown in Bold

This program calculates the area of a rectangle.

Enter the length and width of the rectangle separated by a space.

10 20 [Enter]

The area of the rectangle is 200



The `cin` Object Reads Different Data Types in Program 3-3

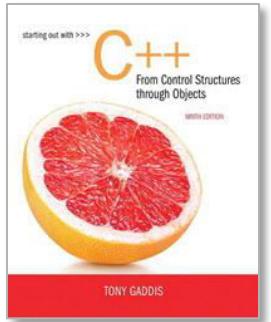
Program 3-3

```
1 // This program demonstrates how cin can read multiple values
2 // of different data types.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int whole;
9     double fractional;
10    char letter;
11
12    cout << "Enter an integer, a double, and a character: ";
13    cin >> whole >> fractional >> letter;
14    cout << "Whole: " << whole << endl;
15    cout << "Fractional: " << fractional << endl;
16    cout << "Letter: " << letter << endl;
17    return 0;
18 }
```

Program Output with Example Input Shown in Bold

```
Enter an integer, a double, and a character: 4 5.7 b [Enter]
Whole: 4
Fractional: 5.7
Letter: b
```





3.2

Mathematical Expressions

Mathematical Expressions

- Orange circle icon: Can create complex expressions using multiple mathematical operators
- Orange circle icon: An expression can be a literal, a variable, or a mathematical combination of constants and variables
- Orange circle icon: Can be used in assignment, cout, other statements:

```
area = 2 * PI * radius;  
cout << "border is: " << 2*(l+w);
```

Order of Operations

In an expression with more than one operator, evaluate in this order:

- (unary negation), in order, left to right
- * / %, in order, left to right
- + -, in order, left to right

In the expression $2 + 2 * 2 - 2$

evaluate second evaluate first evaluate third

Order of Operations

Table 3-2 Some Simple Expressions and Their Values

Expression	Value
5 + 2 * 4	13
10 / 2 - 3	2
8 + 12 * 2 - 4	28
4 + 17 % 2 - 1	4
6 - 3 * 2 + 7 - 1	6

Associativity of Operators

- (unary negation) associates right to left
- * , / , % , + , – associate right to left
- parentheses () can be used to override the order of operations:

$$2 + 2 * 2 - 2 = 4$$

$$(2 + 2) * 2 - 2 = 6$$

$$2 + 2 * (2 - 2) = 2$$

$$(2 + 2) * (2 - 2) = 0$$

Grouping with Parentheses

Table 3-4 More Simple Expressions and Their Values

Expression	Value
(5 + 2) * 4	28
10 / (5 - 3)	5
8 + 12 * (6 - 2)	56
(4 + 17) % 2 - 1	0
(6 - 3) * (2 + 7) / 3	9

Algebraic Expressions

- Orange circle icon: Multiplication requires an operator:

Area = lw is written as `Area = l * w;`

- Orange circle icon: There is no exponentiation operator:

Area = s² is written as `Area = pow(s, 2);`

- Orange circle icon: Parentheses may be needed to maintain order of operations:

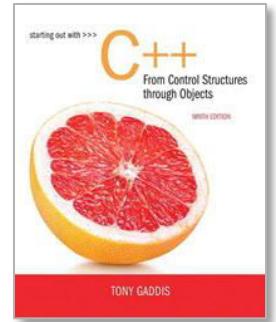
$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

is written as
`m = (y2-y1) / (x2-x1);`

Algebraic Expressions

Table 3-5 Algebraic and C++ Multiplication Expressions

Algebraic Expression	Operation	C++ Equivalent
$6B$	6 times B	<code>6 * B</code>
$(3)(12)$	3 times 12	<code>3 * 12</code>
$4xy$	4 times x times y	<code>4 * x * y</code>



3.3

When You Mix Apples with Oranges: Type Conversion

When You Mix Apples with Oranges: Type Conversion

- Operations are performed between operands of the same type.
- If not of the same type, C++ will convert one to be the type of the other
- This can impact the results of calculations.

Hierarchy of Types

Highest: long double
double
float
unsigned long
long
unsigned int
Lowest: int

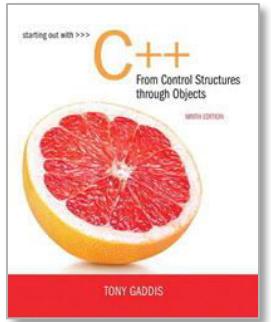
Ranked by largest number they can hold

Type Coercion

- Type Coercion: automatic conversion of an operand to another data type
- Promotion: convert to a higher type
- Demotion: convert to a lower type

Coercion Rules

- 1) char, short, unsigned short automatically promoted to int
- 2) When operating on values of different data types, the lower one is promoted to the type of the higher one.
- 3) When using the = operator, the type of expression on right will be converted to type of variable on left

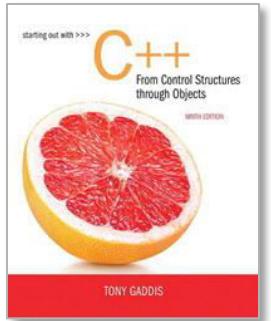


3.4

Overflow and Underflow

Overflow and Underflow

- Orange circle icon: Occurs when assigning a value that is too large (overflow) or too small (underflow) to be held in a variable
- Orange circle icon: Variable contains value that is ‘wrapped around’ set of possible values
- Orange circle icon: Different systems may display a warning/error message, stop the program, or continue execution using the incorrect value



3.5

Type Casting

Type Casting

- Orange Used for manual data type conversion
- Orange Useful for floating point division using ints:

```
double m;  
m = static_cast<double>(y2-y1)  
      / (x2-x1);
```
- Orange Useful to see int value of a char variable:

```
char ch = 'C';  
cout << ch << " is "  
     << static_cast<int>(ch);
```

Type Casting in Program 3-9

Program 3-9

```
1 // This program uses a type cast to avoid integer division.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int books;          // Number of books to read
8     int months;         // Number of months spent reading
9     double perMonth;   // Average number of books per month
10
11    cout << "How many books do you plan to read? ";
12    cin >> books;
13    cout << "How many months will it take you to read them? ";
14    cin >> months;
15    perMonth = static_cast<double>(books) / months;
16    cout << "That is " << perMonth << " books per month.\n";
17    return 0;
18 }
```

Program Output with Example Input Shown in Bold

How many books do you plan to read? **30** [Enter]

How many months will it take you to read them? **7** [Enter]

That is 4.28571 books per month.



C-Style and Prestandard Type Cast Expressions

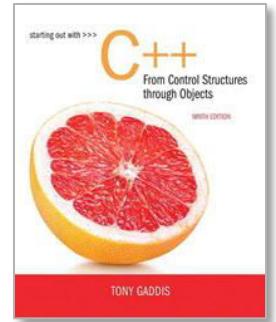
- C-Style cast: data type name in ()

```
cout << ch << " is " << (int)ch;
```

- Prestandard C++ cast: value in ()

```
cout << ch << " is " << int(ch);
```

- Both are still supported in C++, although static_cast is preferred



3.6

Multiple Assignment and Combined Assignment

Multiple Assignment and Combined Assignment

- The = can be used to assign a value to multiple variables:

```
x = y = z = 5;
```

- Value of = is the value that is assigned
- Associates right to left:

```
x = (y = (z = 5));
```

The diagram illustrates the associativity of the assignment operator. Three orange arrows point from the text "value is 5" to the three assignment operators (=) in the expression `x = (y = (z = 5));`. The first arrow points to the innermost assignment `z = 5`, the second to the middle assignment `y = (z = 5)`, and the third to the outermost assignment `x = (y = (z = 5))`.

Combined Assignment

- Look at the following statement:

```
sum = sum + 1;
```

This adds 1 to the variable **sum**.

Other Similar Statements

Table 3-8 (Assume $x = 6$)

Statement	What It Does	Value of x After the Statement
$x = x + 4;$	Adds 4 to x	10
$x = x - 3;$	Subtracts 3 from x	3
$x = x * 10;$	Multiplies x by 10	60
$x = x / 2;$	Divides x by 2	3
$x = x \% 4$	Makes x the remainder of $x / 4$	2

Combined Assignment

- The combined assignment operators provide a shorthand for these types of statements.
- The statement

```
sum = sum + 1;
```

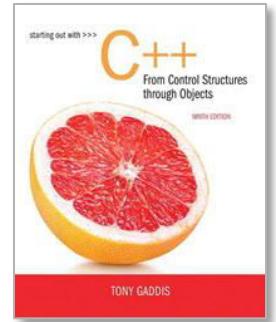
is equivalent to

```
sum += 1;
```

Combined Assignment Operators

Table 3-9

Operator	Example Usage	Equivalent to
<code>+=</code>	<code>x += 5;</code>	<code>x = x + 5;</code>
<code>-=</code>	<code>y -= 2;</code>	<code>y = y - 2;</code>
<code>*=</code>	<code>z *= 10;</code>	<code>z = z * 10;</code>
<code>/=</code>	<code>a /= b;</code>	<code>a = a / b;</code>
<code>%=</code>	<code>c %= 3;</code>	<code>c = c % 3;</code>



3.7

Formatting Output

Formatting Output

- Can control how output displays for numeric, string data:
 - size
 - position
 - number of digits
- Requires `iomanip` header file

Stream Manipulators

- Used to control how an output field is displayed
- Some affect just the next value displayed:
 - `setw(x)`: print in a field at least x spaces wide. Use more spaces if field is not wide enough

The `setw` Stream Manipulator in Program 3-13

Program 3-13

```
1 // This program displays three rows of numbers.
2 #include <iostream>
3 #include <iomanip>      // Required for setw
4 using namespace std;
5
6 int main()
7 {
8     int num1 = 2897, num2 = 5,     num3 = 837,
9         num4 = 34,    num5 = 7,     num6 = 1623,
10        num7 = 390,   num8 = 3456, num9 = 12;
11
12    // Display the first row of numbers
13    cout << setw(6) << num1 << setw(6)
14        << num2 << setw(6) << num3 << endl;
15
16    // Display the second row of numbers
17    cout << setw(6) << num4 << setw(6)
18        << num5 << setw(6) << num6 << endl;
19
20    // Display the third row of numbers
21    cout << setw(6) << num7 << setw(6)
22        << num8 << setw(6) << num9 << endl;
23    return 0;
24 }
```

Continued...



The `setw` Stream Manipulator in Program 3-13

Program Output

2897	5	837
34	7	1623
390	3456	12



Stream Manipulators

- Some affect values until changed again:
 - fixed: use decimal notation for floating-point values
 - setprecision(x): when used with fixed, print floating-point value using x digits after the decimal. Without fixed, print floating-point value using x significant digits
 - showpoint: always print decimal for floating-point values

More Stream Manipulators in Program 3-17

Program 3-17

```
1 // This program asks for sales amounts for 3 days. The total
2 // sales are calculated and displayed in a table.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     double day1, day2, day3, total;
10
11     // Get the sales for each day.
12     cout << "Enter the sales for day 1: ";
13     cin >> day1;
14     cout << "Enter the sales for day 2: ";
15     cin >> day2;
16     cout << "Enter the sales for day 3: ";
17     cin >> day3;
18
19     // Calculate the total sales.
20     total = day1 + day2 + day3;
21 }
```

Continued...



More Stream Manipulators in Program 3-17

```
22     // Display the sales amounts.  
23     cout << "\nSales Amounts\n";  
24     cout << "-----\n";  
25     cout << setprecision(2) << fixed;  
26     cout << "Day 1: " << setw(8) << day1 << endl;  
27     cout << "Day 2: " << setw(8) << day2 << endl;  
28     cout << "Day 3: " << setw(8) << day3 << endl;  
29     cout << "Total: " << setw(8) << total << endl;  
30     return 0;  
31 }
```

Program Output with Example Input Shown in Bold

Enter the sales for day 1: **1321.87**

Enter the sales for day 2: **1869.26**

Enter the sales for day 3: **1403.77**

Sales Amounts

Day 1: 1321.87

Day 2: 1869.26

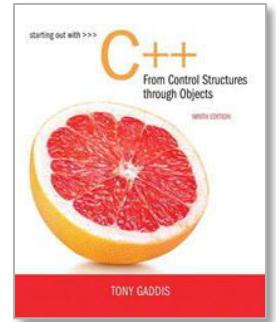
Day 3: 1403.77

Total: 4594.90

Stream Manipulators

Table 3-12

Stream Manipulator	Description
<code>setw(<i>n</i>)</code>	Establishes a print field of <i>n</i> spaces.
<code>fixed</code>	Displays floating-point numbers in fixed point notation.
<code>showpoint</code>	Causes a decimal point and trailing zeroes to be displayed, even if there is no fractional part.
<code>setprecision(<i>n</i>)</code>	Sets the precision of floating-point numbers.
<code>left</code>	Causes subsequent output to be left justified.
<code>right</code>	Causes subsequent output to be right justified.



3.8

Working with Characters and string Objects

Working with Characters and **string** Objects

- Using `cin` with the `>>` operator to input strings can cause problems:
- It passes over and ignores any leading *whitespace characters* (*spaces, tabs, or line breaks*)
- To work around this problem, you can use a C++ function named `getline`.

Using `getline` in Program 3-19

Program 3-19

```
1 // This program demonstrates using the getline function
2 // to read character data into a string object.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string name;
10    string city;
11
12    cout << "Please enter your name: ";
13    getline(cin, name);
14    cout << "Enter the city you live in: ";
15    getline(cin, city);
16
17    cout << "Hello, " << name << endl;
18    cout << "You live in " << city << endl;
19    return 0;
20 }
```

Program Output with Example Input Shown in Bold

Please enter your name: **Kate Smith** [Enter]

Enter the city you live in: **Raleigh** [Enter]

Hello, Kate Smith

You live in Raleigh



Working with Characters and **string** Objects

- To read a single character:

- Use `cin`:

```
char ch;  
cout << "Strike any key to continue";  
cin >> ch;
```

Problem: will skip over blanks, tabs, <CR>

- Use `cin.get()`:

```
cin.get(ch);
```

Will read the next character entered, even whitespace

Using `cin.get()` in Program 3-21

Program 3-21

```
1 // This program demonstrates three ways
2 // to use cin.get() to pause a program.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     char ch;
9
10    cout << "This program has paused. Press Enter to continue.";
11    cin.get(ch);
12    cout << "It has paused a second time. Please press Enter again.";
13    ch = cin.get();
14    cout << "It has paused a third time. Please press Enter again.";
15    cin.get();
16    cout << "Thank you!";
17    return 0;
18 }
```

Program Output with Example Input Shown in Bold

This program has paused. Press Enter to continue. **[Enter]**
It has paused a second time. Please press Enter again. **[Enter]**
It has paused a third time. Please press Enter again. **[Enter]**
Thank you!



Working with Characters and **string** Objects

- Orange Mixing `cin >>` and `cin.get()` in the same program can cause input errors that are hard to detect
- Orange To skip over unneeded characters that are still in the keyboard buffer, use `cin.ignore()`:

```
cin.ignore(); // skip next char
cin.ignore(10, '\n'); // skip the next
                     // 10 char. or until a '\n'
```

string Member Functions and Operators

- To find the length of a string:

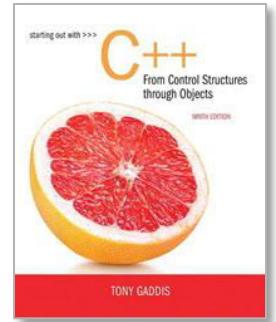
```
string state = "Texas";  
int size = state.length();
```

- To concatenate (join) multiple strings:

```
greeting2 = greeting1 + name1;  
greeting1 = greeting1 + name2;
```

Or using the `+=` combined assignment operator:

```
greeting1 += name2;
```



3.9

More Mathematical Library Functions

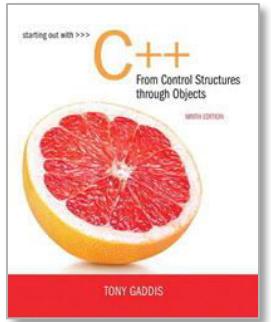
More Mathematical Library Functions

- Require cmath header file
- Take double as input, return a double
- Commonly used functions:

sin	Sine
cos	Cosine
tan	Tangent
sqrt	Square root
log	Natural (e) log
abs	Absolute value (takes and returns an int)

More Mathematical Library Functions

- These require `cstdlib` header file
- `rand()` : returns a random number (`int`) between 0 and the largest `int` the computer holds. Yields same sequence of numbers each time program is run.
- `srand(x)` : initializes random number generator with `unsigned int x`



3.10

Hand Tracing a Program

Hand Tracing a Program

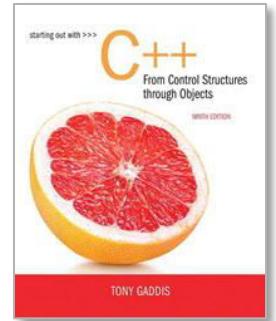
- Orange Hand trace a program: act as if you are the computer, executing a program:
 - Orange step through and ‘execute’ each statement, one-by-one
 - Orange record the contents of variables after statement execution, using a hand trace chart (table)
- Orange Useful to locate logic or mathematical errors

Program 3-27 with Hand Trace Chart

Program 3-27 (with hand trace chart filled)

```
1 // This program asks for three numbers, then
2 // displays the average of the numbers.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     double num1, num2, num3, avg;
9     cout << "Enter the first number: ";
10    cin >> num1;
11    cout << "Enter the second number: ";
12    cin >> num2;
13    cout << "Enter the third number: ";
14    cin >> num3;
15    avg = num1 + num2 + num3 / 3;
16    cout << "The average is " << avg << endl;
17 }
```

num1	num2	num3	avg
?	?	?	?
?	?	?	?
10	?	?	?
10	?	?	?
10	20	?	?
10	20	?	?
10	20	30	?
10	20	30	40
10	20	30	40



3.11

A Case Study

A Case Study

- General Crates, Inc. builds custom-designed wooden crates.
- You have been asked to write a program that calculates the:
 - Volume (in cubic feet)
 - Cost
 - Customer price
 - Profit of any crate GCI builds

Variables

Table 3-14

Constant or Variable	Description
<code>COST_PER_CUBIC_FOOT</code>	A named constant, declared as a <code>double</code> and initialized with the value 0.23. This represents the cost to build a crate, per cubic foot.
<code>CHARGE_PER_CUBIC_FOOT</code>	A named constant, declared as a <code>double</code> and initialized with the value 0.5. This represents the amount charged for a crate, per cubic foot.
<code>length</code>	A <code>double</code> variable to hold the length of the crate, which is input by the user.
<code>width</code>	A <code>double</code> variable to hold the width of the crate, which is input by the user.
<code>height</code>	A <code>double</code> variable to hold the height of the crate, which is input by the user.
<code>volume</code>	A <code>double</code> variable to hold the volume of the crate. The value stored in this variable is calculated.
<code>cost</code>	A <code>double</code> variable to hold the cost of building the crate. The value stored in this variable is calculated.
<code>charge</code>	A <code>double</code> variable to hold the amount charged to the customer for the crate. The value stored in this variable is calculated.
<code>profit</code>	A <code>double</code> variable to hold the profit GCI makes from the crate. The value stored in this variable is calculated.

Program Design

The program must perform the following general steps:

Step 1:

 Ask the user to enter the dimensions of the crate

Step 2:

 Calculate:

 the crate's volume

 the cost of building the crate

 the customer's charge

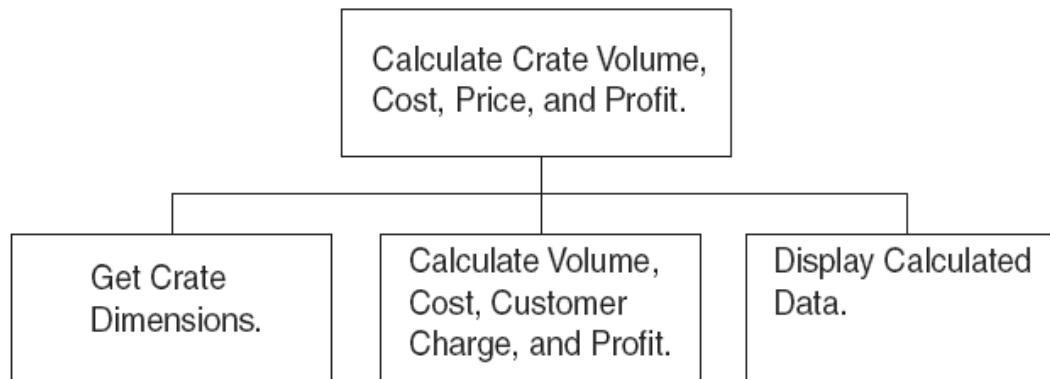
 the profit made

Step 3:

 Display the data calculated in Step 2.

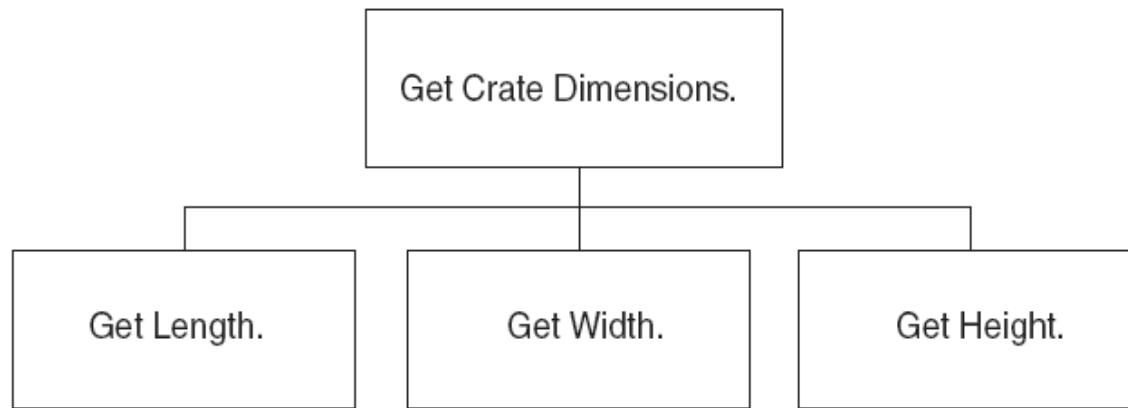
General Hierarchy Chart

Figure 3-7



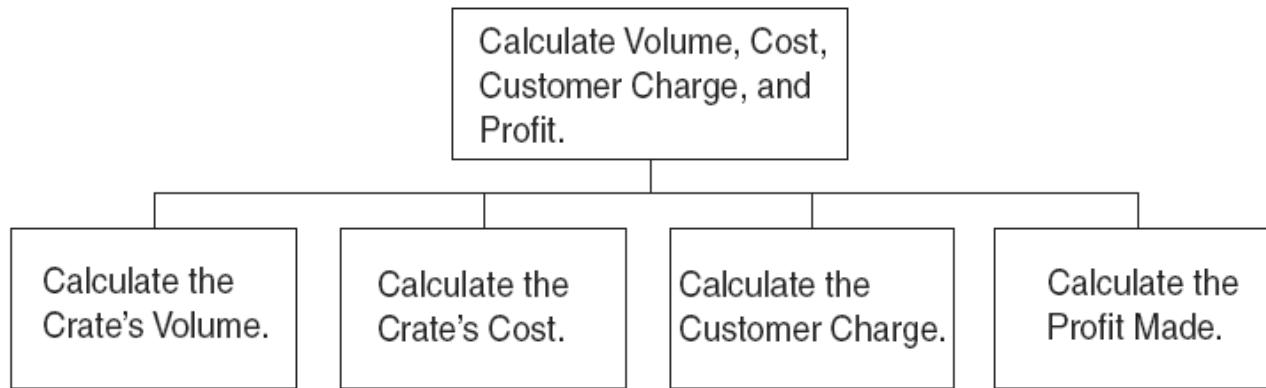
Get Crate Dimensions

Figure 3-8



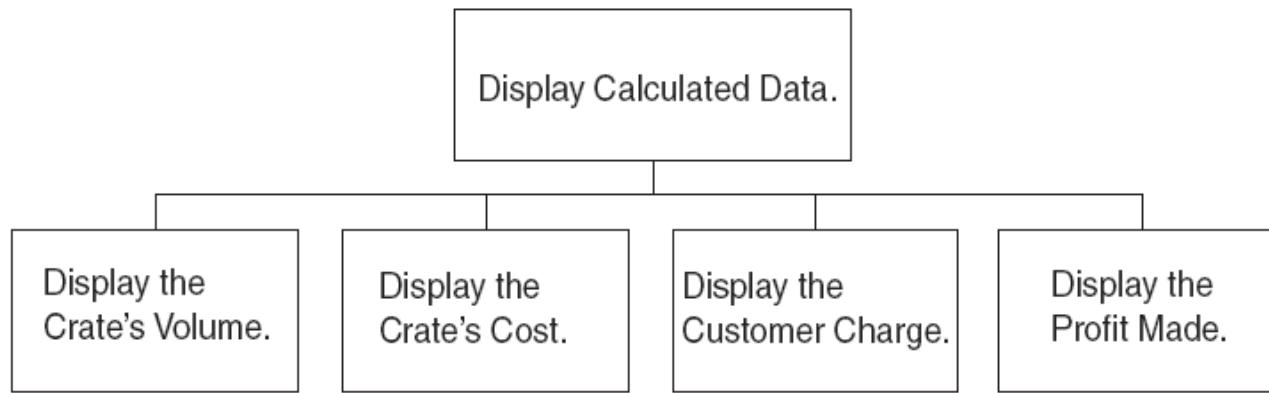
Calculate Volume, Cost, Customer Charge, and Profit

Figure 3-9



Display Calculated Data

Figure 3-10



Psuedocode

Ask the user to input the crate's length.

Ask the user to input the crate's width.

Ask the user to input the crate's height.

Calculate the crate's volume.

Calculate the cost of building the crate.

Calculate the customer's charge for the crate.

Calculate the profit made from the crate.

Display the crate's volume.

Display the cost of building the crate.

Display the customer's charge for the crate.

Display the profit made from the crate.

Calculations

The following formulas will be used to calculate the crate's volume, cost, charge, and profit:

$$\text{volume} = \text{length} \times \text{width} \times \text{height}$$

$$\text{cost} = \text{volume} \times 0.23$$

$$\text{charge} = \text{volume} \times 0.5$$

$$\text{profit} = \text{charge} - \text{cost}$$

The Program

Program 3-28

```
1 // This program is used by General Crates, Inc. to calculate
2 // the volume, cost, customer charge, and profit of a crate
3 // of any size. It calculates this data from user input, which
4 // consists of the dimensions of the crate.
5 #include <iostream>
6 #include <iomanip>
7 using namespace std;
8
9 int main()
10 {
11     // Constants for cost and amount charged
12     const double COST_PER_CUBIC_FOOT = 0.23;
13     const double CHARGE_PER_CUBIC_FOOT = 0.5;
14
15     // Variables
16     double length,      // The crate's length
17           width,       // The crate's width
18           height,      // The crate's height
19           volume,      // The volume of the crate
20           cost,        // The cost to build the crate
21           charge,      // The customer charge for the crate
22           profit;      // The profit made on the crate
23
24     // Set the desired output formatting for numbers.
25     cout << setprecision(2) << fixed << showpoint;
26 }
```

Continued...



The Program

```
27     // Prompt the user for the crate's length, width, and height
28     cout << "Enter the dimensions of the crate (in feet):\n";
29     cout << "Length: ";
30     cin >> length;
31     cout << "Width: ";
32     cin >> width;
33     cout << "Height: ";
34     cin >> height;
35
36     // Calculate the crate's volume, the cost to produce it,
37     // the charge to the customer, and the profit.
38     volume = length * width * height;
39     cost = volume * COST_PER_CUBIC_FOOT;
40     charge = volume * CHARGE_PER_CUBIC_FOOT;
41     profit = charge - cost;
42
43     // Display the calculated data.
44     cout << "The volume of the crate is ";
45     cout << volume << " cubic feet.\n";
46     cout << "Cost to build: $" << cost << endl;
47     cout << "Charge to customer: $" << charge << endl;
48     cout << "Profit: $" << profit << endl;
49
50 }
```

Continued...

The Program

Program Output with Example Input Shown in Bold

Enter the dimensions of the crate (in feet):

Length: **10** [Enter]

Width: **8** [Enter]

Height: **4** [Enter]

The volume of the crate is 320.00 cubic feet.

Cost to build: \$73.60

Charge to customer: \$160.00

Profit: \$86.40

Program Output with Different Example Input Shown in Bold

Enter the dimensions of the crate (in feet):

Length: **12.5** [Enter]

Width: **10.5** [Enter]

Height: **8** [Enter]

The volume of the crate is 1050.00 cubic feet.

Cost to build: \$241.50

Charge to customer: \$525.00

Profit: \$283.50

Chapter 4:

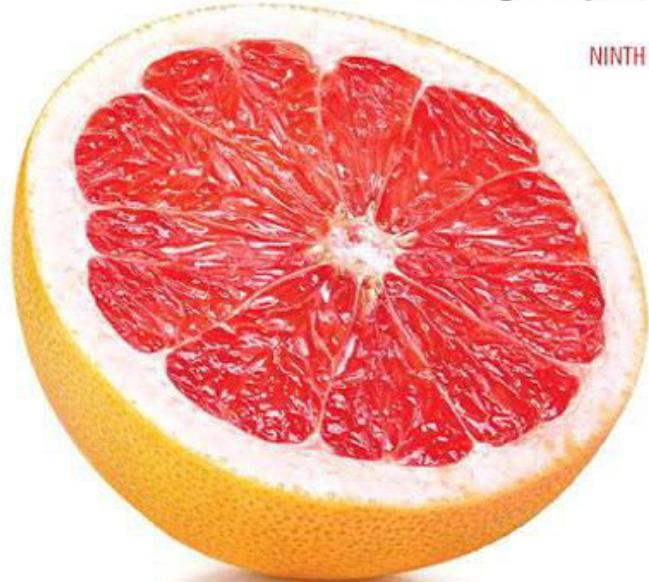
Making Decisions

starting out with >>>

C++

From Control Structures
through Objects

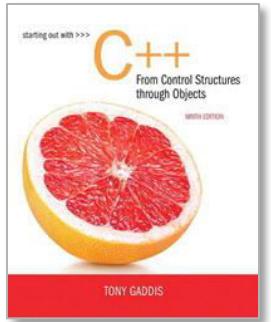
NINTH EDITION



TONY GADDIS



Pearson Copyright © 2018, 2015, 2012, 2009 Pearson Education, Inc. All rights reserved.



4.1

Relational Operators



Relational Operators

- Used to compare numbers to determine relative order
- Operators:

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



Relational Expressions

- Orange Boolean expressions – true or false
- Orange Examples:

`12 > 5` is true

`7 <= 5` is false

`if x is 10, then`

`x == 10` is true,

`x != 8` is true, and

`x == 8` is false



Relational Expressions

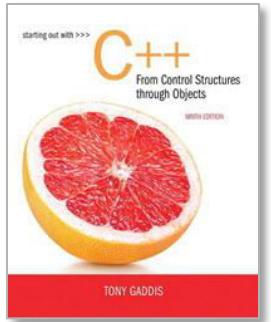
- Orange icon: Can be assigned to a variable:

```
result = x <= y;
```

- Orange icon: Assigns 0 for false, 1 for true

- Orange icon: Do not confuse = and ==





4.2

The if Statement

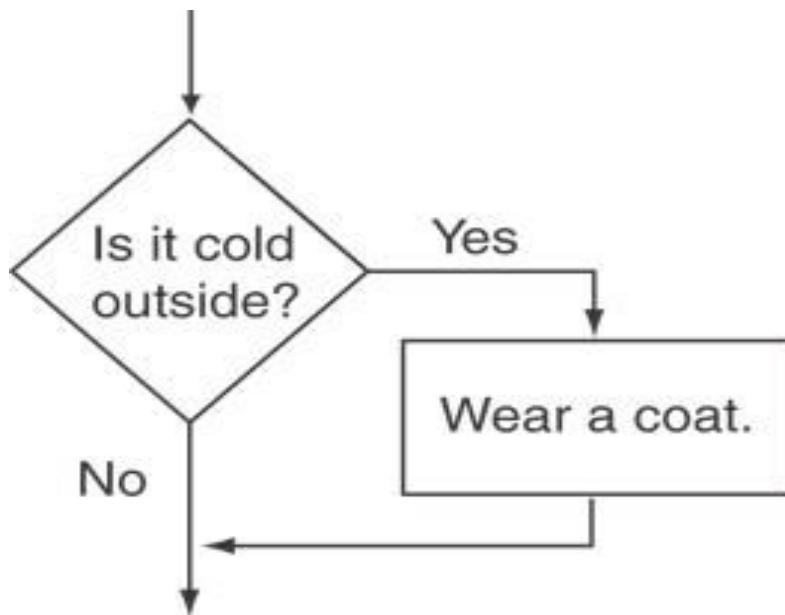


The if Statement

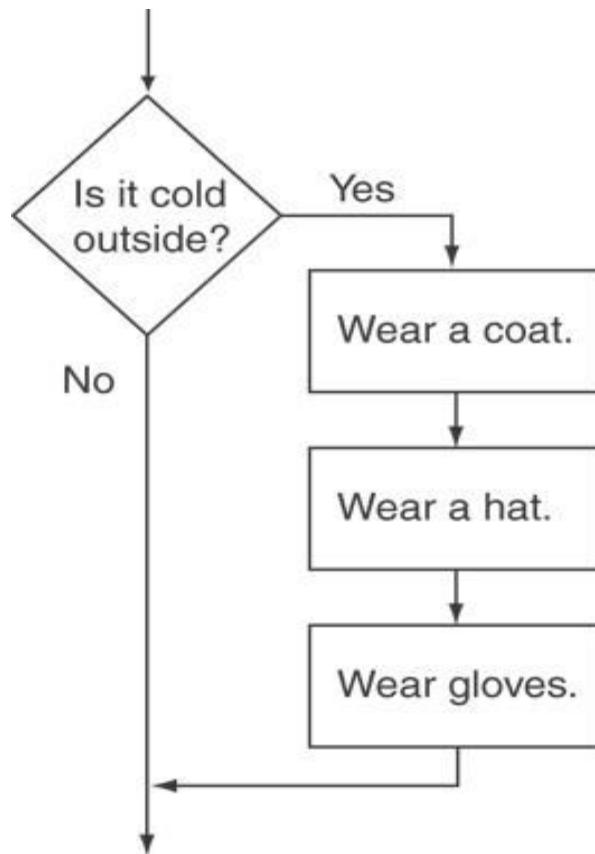
- Allows statements to be conditionally executed or skipped over
- Models the way we mentally evaluate situations:
 - "If it is raining, take an umbrella."
 - "If it is cold outside, wear a coat."



Flowchart for Evaluating a Decision



Flowchart for Evaluating a Decision



The if Statement

- General Format:

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



The if Statement-What Happens

To evaluate:

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

- If the *expression* is true, then *statement* is executed.
- If the *expression* is false, then *statement* is skipped.



if Statement in Program 4-2

Program 4-2

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

Continued...



if Statement in Program 4-2

Program 4-2 *(continued)*

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

Program Output with Example Input Shown in Bold

Enter 3 test scores and I will average them: **80 90 70 [Enter]**

Your average is 80.0

Program Output with Other Example Input Shown in Bold

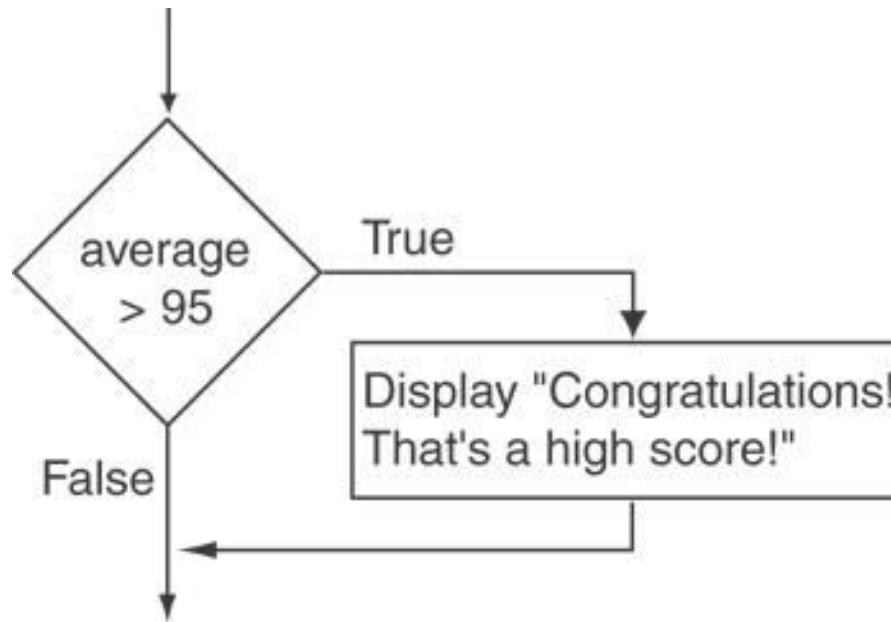
Enter 3 test scores and I will average them: **100 100 100 [Enter]**

Your average is 100.0

Congratulations! That's a high score!



Flowchart for Program 4-2 Lines 21 and 22



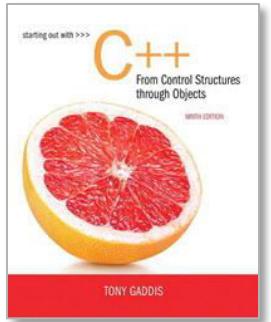
if Statement Notes

- Orange circle icon: Do not place ; after *(expression)*
- Orange circle icon: Place *statement;* on a separate line after *(expression)*, indented:

```
if (score > 90)  
    grade = 'A';
```

- Orange circle icon: Be careful testing floats and doubles for equality
- Orange circle icon: 0 is false; any other value is true





4.3

Expanding the `if` Statement



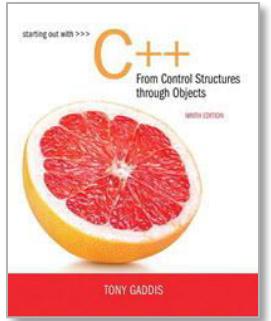
Expanding the `if` Statement

- Orange circle icon To execute more than one statement as part of an `if` statement, enclose them in `{ }`:

```
if (score > 90)
{
    grade = 'A';
    cout << "Good Job! \n";
}
```

- Orange circle icon `{ }` creates a block of code





4.4

The if/else Statement



The `if/else` statement

- Orange Provides two possible paths of execution
- Orange Performs one statement or block if the *expression* is true, otherwise performs another statement or block.



The **if/else** statement

General Format:

```
if (expression)
    statement1; // or block
else
    statement2; // or block
```



if/else-What Happens

To evaluate:

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

- If the *expression* is true, then *statement1* is executed and *statement2* is skipped.
- If the *expression* is false, then *statement1* is skipped and *statement2* is executed.



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

Program 4-8

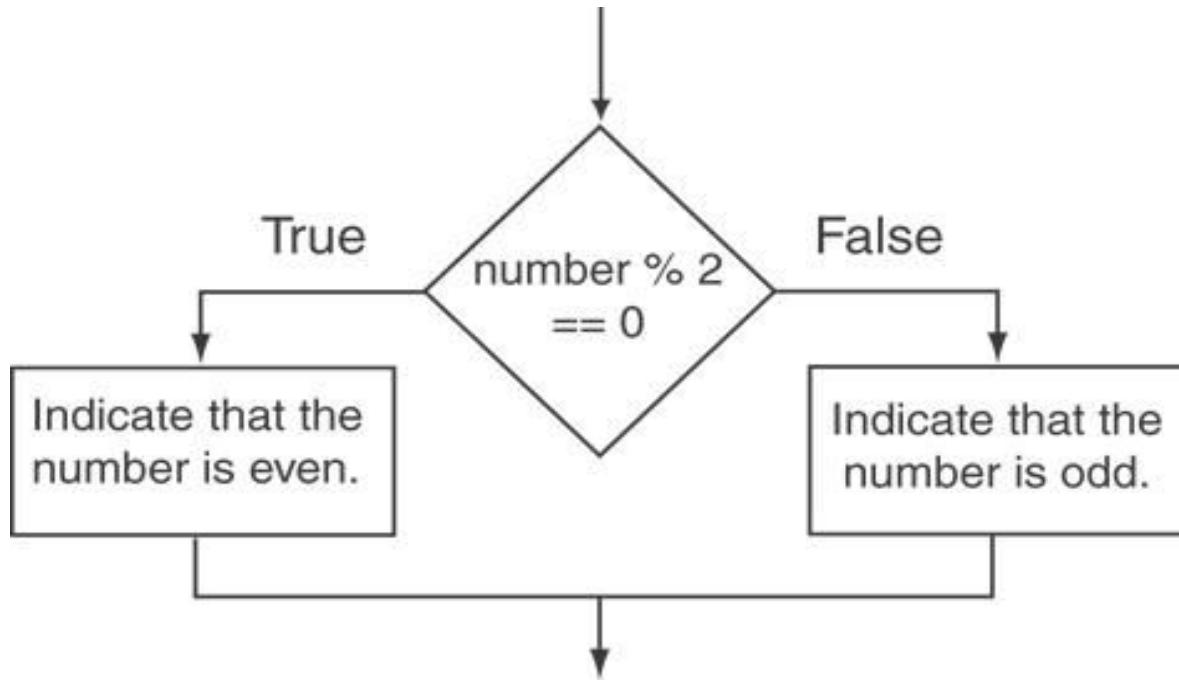
```
1 // This program uses the modulus operator to determine
2 // if a number is odd or even. If the number is evenly divisible
3 // by 2, it is an even number. A remainder indicates it is odd.
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     int number;
10
11    cout << "Enter an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

Program Output with Example Input Shown in Bold

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



Flowchart for Program 4-8 Lines 14 through 18



Testing the Divisor in Program 4-9

Program 4-9

```
1 // This program asks the user for two numbers, num1 and num2.  
2 // num1 is divided by num2 and the result is displayed.  
3 // Before the division operation, however, num2 is tested  
4 // for the value 0. If it contains 0, the division does not  
5 // take place.  
6 #include <iostream>  
7 using namespace std;  
8  
9 int main()  
10 {  
11     double num1, num2, quotient;  
12 }
```

Continued...



Testing the Divisor in Program 4-9

Program 4-9 *(continued)*

```
13     // Get the first number.  
14     cout << "Enter a number: ";  
15     cin >> num1;  
16  
17     // Get the second number.  
18     cout << "Enter another number: ";  
19     cin >> num2;  
20  
21     // If num2 is not zero, perform the division.  
22     if (num2 == 0)  
23     {  
24         cout << "Division by zero is not possible.\n";  
25         cout << "Please run the program again and enter\n";  
26         cout << "a number other than zero.\n";  
27     }  
28     else  
29     {  
30         quotient = num1 / num2;  
31         cout << "The quotient of " << num1 << " divided by "  
32         cout << num2 << " is " << quotient << ".\n";  
33     }  
34     return 0;  
35 }
```

Program Output with Example Input Shown in Bold

(When the user enters 0 for num2)

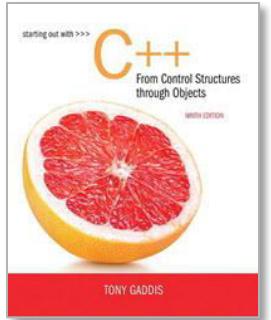
Enter a number: **10 [Enter]**

Enter another number: **0 [Enter]**

Division by zero is not possible.

Please run the program again and enter
a number other than zero.





4.5

Nested `if` Statements

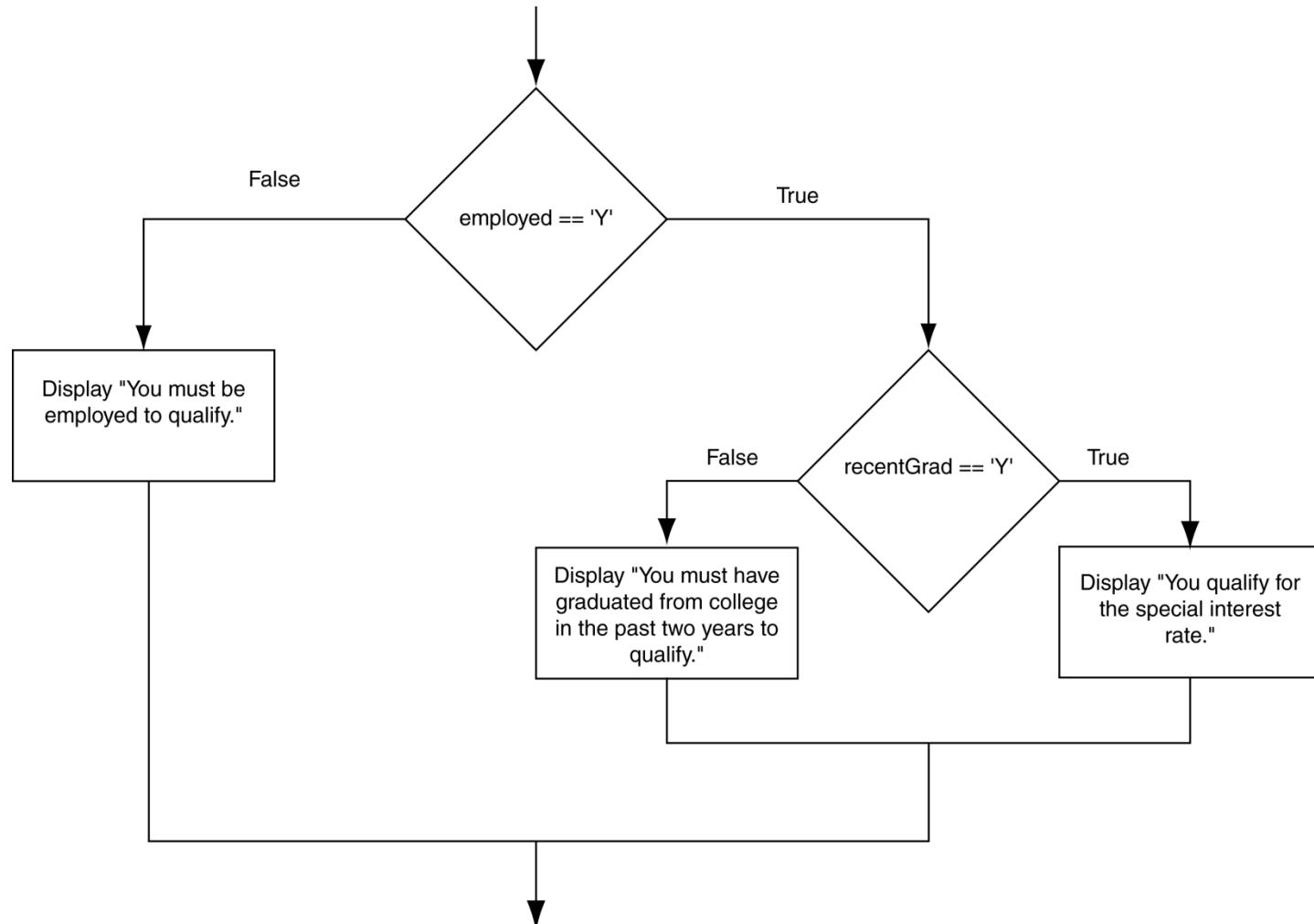


Nested if Statements

- An if statement that is nested inside another if statement
- Nested if statements can be used to test more than one condition



Flowchart for a Nested if Statement



Nested if Statements

From Program 4-10

```
20     // Determine the user's loan qualifications.  
21     if (employed == 'Y')  
22     {  
23         if (recentGrad == 'Y') //Nested if  
24         {  
25             cout << "You qualify for the special "  
26             cout << "interest rate.\n";  
27         }  
28     }
```



Nested if Statements

Another example, from Program 4-1

```
20     // Determine the user's loan qualifications.  
21     if (employed == 'Y')  
22     {  
23         if (recentGrad == 'Y') // Nested if  
24         {  
25             cout << "You qualify for the special "  
26             cout << "interest rate.\n";  
27         }  
28     else // Not a recent grad, but employed  
29     {  
30         cout << "You must have graduated from "  
31         cout << "college in the past two\n";  
32         cout << "years to qualify.\n";  
33     }  
34 }  
35 else // Not employed  
36 {  
37     cout << "You must be employed to qualify.\n";  
38 }
```



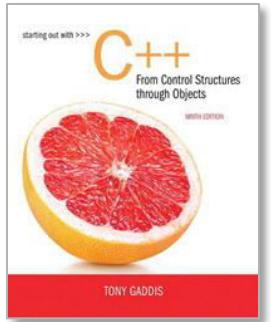
Use Proper Indentation!

```
if (employed == 'Y')
{
    if (recentGrad == 'Y') // Nested if
    {
        cout << "You qualify for the special ";
        cout << "interest rate.\n";
    }
    else // Not a recent grad, but employed
    {
        cout << "You must have graduated from ";
        cout << "college in the past two\n";
        cout << "years to qualify.\n";
    }
}
else // Not employed
{
    cout << "You must be employed to qualify.\n";
}
```

This if and else go together.

This if and else go together.





4.6

The if/else if Statement



The `if/else if` Statement

- Tests a series of conditions until one is found to be true
- Often simpler than using nested `if/else` statements
- Can be used to model thought processes such as:

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



if/else if Format

```
if (expression)
    statement1; // or block
else if (expression)
    statement2; // or block
.
.
// other else ifs
.
else if (expression)
    statementn; // or block
```



The if/else if Statement in Program 4-13

```
21     // Determine the letter grade.  
22     if (testScore >= A_SCORE)  
23         cout << "Your grade is A.\n";  
24     else if (testScore >= B_SCORE)  
25         cout << "Your grade is B.\n";  
26     else if (testScore >= C_SCORE)  
27         cout << "Your grade is C.\n";  
28     else if (testScore >= D_SCORE)  
29         cout << "Your grade is D.\n";  
30     else  
31         cout << "Your grade is F.\n";
```



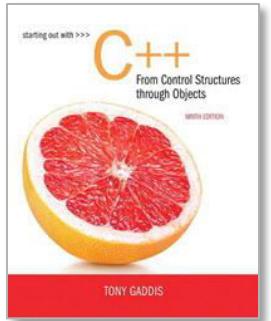
Using a Trailing else to Catch Errors in Program 4-14

- The trailing else clause is optional, but it is best used to catch errors.

```
21     // Determine the letter grade.  
22     if (testScore >= A_SCORE)  
23         cout << "Your grade is A.\n";  
24     else if (testScore >= B_SCORE)  
25         cout << "Your grade is B.\n";  
26     else if (testScore >= C_SCORE)  
27         cout << "Your grade is C.\n";  
28     else if (testScore >= D_SCORE)  
29         cout << "Your grade is D.\n";  
30     else if (testScore >= 0)  
31         cout << "Your grade is F.\n";  
32     else  
33         cout << "Invalid test score.\n";
```

This trailing
else
catches
invalid test
scores





4.7

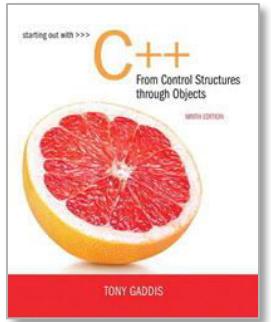
Flags



Flags

- Variable that signals a condition
- Usually implemented as a `bool` variable
- Can also be an integer
 - The value `0` is considered `false`
 - Any nonzero value is considered `true`
- As with other variables in functions, must be assigned an initial value before it is used





4.8

Logical Operators



Logical Operators

- Used to create relational expressions from other relational expressions
- Operators, meaning, and explanation:

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



Logical Operators-Examples

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

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



The logical && operator in Program 4-15

```
21     // Determine the user's loan qualifications.  
22     if (employed == 'Y' && recentGrad == 'Y')  
23     {  
24         cout << "You qualify for the special "  
25             << "interest rate.\n";  
26     }  
27     else  
28     {  
29         cout << "You must be employed and have\n"  
30             << "graduated from college in the\n"  
31             << "past two years to qualify.\n";  
32     }
```



The logical || Operator in Program

4-16

```
23 // Determine the user's loan qualifications.  
24 if (income >= MIN_INCOME || years > MIN_YEARS)  
25     cout << "You qualify.\n";  
26 else  
27 {  
28     cout << "You must earn at least $"  
29             << MIN_INCOME << " or have been "  
30             << "employed more than " << MIN_YEARS  
31             << " years.\n";  
32 }
```



The logical ! Operator in Program

4-17

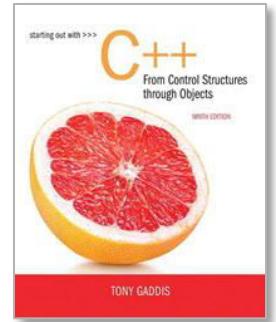
```
23 // Determine the user's loan qualifications.  
24 if (!(income >= MIN_INCOME || years > MIN_YEARS))  
25 {  
26     cout << "You must earn at least $"  
27         << MIN_INCOME << " or have been "  
28         << "employed more than " << MIN_YEARS  
29         << " years.\n";  
30 }  
31 else  
32     cout << "You qualify.\n";
```



Logical Operator-Notes

- ! has highest precedence, followed by &&, then ||
- If the value of an expression can be determined by evaluating just the sub-expression on left side of a logical operator, then the sub-expression on the right side will not be evaluated (*short circuit evaluation*)





4.9

Checking Numeric Ranges with Logical Operators



Checking Numeric Ranges with Logical Operators

- Used to test to see if a value falls **inside** a range:

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

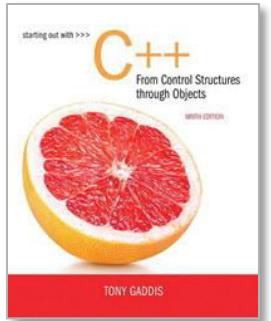
- Can also test to see if value falls **outside** of range:

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

- Cannot use mathematical notation:

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





4.10

Menus



Menus

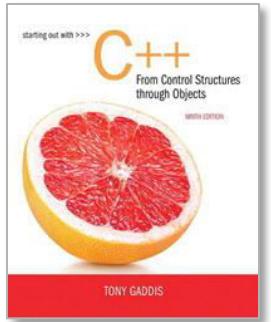
- Menu-driven program: program execution controlled by user selecting from a list of actions
- Menu: list of choices on the screen
- Menus can be implemented using if/else if statements



Menu-Driven Program Organization

- Display list of numbered or lettered choices for actions
- Prompt user to make selection
- Test user selection in *expression*
 - if a match, then execute code for action
 - if not, then go on to next *expression*





4.11

Validating User Input



Validating User Input

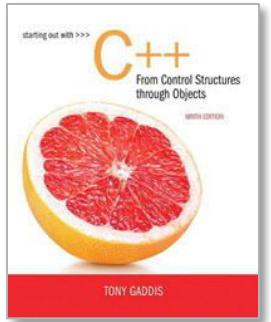
- Input validation: inspecting input data to determine whether it is acceptable
- Bad output will be produced from bad input
- Can perform various tests:
 - Range
 - Reasonableness
 - Valid menu choice
 - Divide by zero



Input Validation in Program 4-19

```
16     int testScore; // To hold a numeric test score
17
18     // Get the numeric test score.
19     cout << "Enter your numeric test score and I will\n"
20         << "tell you the letter grade you earned: ";
21     cin >> testScore;
22
23     // Validate the input and determine the grade.
24     if (testScore >= MIN_SCORE && testScore <= MAX_SCORE)
25     {
26         // Determine the letter grade.
27         if (testScore >= A_SCORE)
28             cout << "Your grade is A.\n";
29         else if (testScore >= B_SCORE)
30             cout << "Your grade is B.\n";
31         else if (testScore >= C_SCORE)
32             cout << "Your grade is C.\n";
33         else if (testScore >= D_SCORE)
34             cout << "Your grade is D.\n";
35         else
36             cout << "Your grade is F.\n";
37     }
38     else
39     {
40         // An invalid score was entered.
41         cout << "That is an invalid score. Run the program\n"
42             << "again and enter a value in the range of\n"
43                 << MIN_SCORE << " through " << MAX_SCORE << ".\n";
44     }
```





4.12

Comparing Characters and Strings



Comparing Characters

- Characters are compared using their ASCII values
- 'A' < 'B'
 - The ASCII value of 'A' (65) is less than the ASCII value of 'B'(66)
- '1' < '2'
 - The ASCII value of '1' (49) is less than the ASCII value of '2' (50)
- Lowercase letters have higher ASCII codes than uppercase letters, so 'a' > 'Z'



Relational Operators Compare Characters in Program 4-20

```
10 // Get a character from the user.  
11 cout << "Enter a digit or a letter: ";  
12 ch = cin.get();  
13  
14 // Determine what the user entered.  
15 if (ch >= '0' && ch <= '9')  
16     cout << "You entered a digit.\n";  
17 else if (ch >= 'A' && ch <= 'Z')  
18     cout << "You entered an uppercase letter.\n";  
19 else if (ch >= 'a' && ch <= 'z')  
20     cout << "You entered a lowercase letter.\n";  
21 else  
22     cout << "That is not a digit or a letter.\n";
```



Comparing **string** Objects

- Like characters, strings are compared using their ASCII values

```
string name1 = "Mary";  
string name2 = "Mark";
```

```
name1 > name2 // true  
name1 <= name2 // false  
name1 != name2 // true
```

```
name1 < "Mary Jane" // true
```

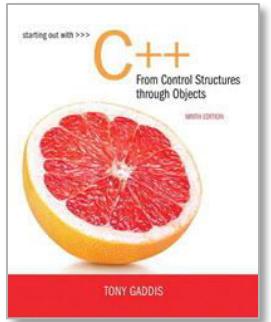
The characters in each string must match before they are equal



Relational Operators Compare Strings in Program 4-21

```
26     // Determine and display the correct price
27     if (partNum == "S-29A")
28         cout << "The price is $" << PRICE_A << endl;
29     else if (partNum == "S-29B")
30         cout << "The price is $" << PRICE_B << endl;
31     else
32         cout << partNum << " is not a valid part number.\n";
```





4.13

The Conditional Operator



The Conditional Operator

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

```
x<0 ? y=10 : z=20;
```

First Expression:
Expression to be tested

2nd Expression:
Executes if first expression is true

3rd Expression:
Executes if the first expression is false

The Conditional Operator

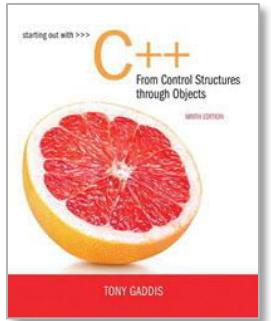
- The value of a conditional expression is
 - The value of the second expression if the first expression is true
 - The value of the third expression if the first expression is false
- Parentheses () may be needed in an expression due to precedence of conditional operator



The Conditional Operator in Program 4-22

```
1 // This program calculates a consultant's charges at $50
2 // per hour, for a minimum of 5 hours. The ?: operator
3 // adjusts hours to 5 if less than 5 hours were worked.
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;
7
8 int main()
9 {
10    const double PAY_RATE = 50.0;    // Hourly pay rate
11    const int MIN_HOURS = 5;        // Minimum billable hours
12    double hours,                 // Hours worked
13        charges;                  // Total charges
14
15    // Get the hours worked.
16    cout << "How many hours were worked? ";
17    cin >> hours;
18
19    // Determine the hours to charge for.
20    hours = hours < MIN_HOURS ? MIN_HOURS : hours;
21
22    // Calculate and display the charges.
23    charges = PAY_RATE * hours;
24    cout << fixed << showpoint << setprecision(2)
25        << "The charges are $" << charges << endl;
26    return 0;
27 }
```





4.14

The switch Statement



The `switch` Statement

- Used to select among statements from several alternatives
- In some cases, can be used instead of `if/else if` statements



switch Statement Format

```
switch (expression) //integer  
{  
    case exp1: statement1;  
    case exp2: statement2;  
    . . .  
    case expn: statementn;  
    default:     statementn+1;  
}
```



The **switch** Statement in Program

4-23

Program 4-23

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

Program Output with Example Input Shown in Bold

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

Program Output with Example Input Shown in Bold

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



`switch` Statement Requirements

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



switch Statement-How it Works

- 1) *expression* is evaluated
- 2) The value of *expression* is compared against *exp₁* through *exp_n*.
- 3) If *expression* matches value *exp_i*, the program branches to the statement following *exp_i* and continues to the end of the **switch**
- 4) If no matching value is found, the program branches to the statement after **default**:



break Statement

- Used to exit a `switch statement`
- If it is left out, the program "falls through" the remaining statements in the `switch statement`



break and default statements in Program 4-25

Program 4-25

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

Continued...



break and default statements in Program 4-25

Program Output with Example Input Shown in Bold

Our TVs come in three models:

The 100, 200, and 300. Which do you want? **100 [Enter]**

That model has the following features:

 Remote control.

Program Output with Example Input Shown in Bold

Our TVs come in three models:

The 100, 200, and 300. Which do you want? **200 [Enter]**

That model has the following features:

 Stereo sound.

 Remote control.

Program Output with Example Input Shown in Bold

Our TVs come in three models:

The 100, 200, and 300. Which do you want? **300 [Enter]**

That model has the following features:

 Picture-in-a-picture.

 Stereo sound.

 Remote control.

Program Output with Example Input Shown in Bold

Our TVs come in three models:

The 100, 200, and 300. Which do you want? **500 [Enter]**

That model has the following features:

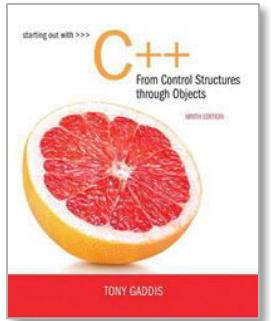
You can only choose the 100, 200, or 300.



Using `switch` in Menu Systems

- `switch` statement is a natural choice for menu-driven program:
 - display the menu
 - then, get the user's menu selection
 - use user input as `expr` in `switch` statement
 - use menu choices as `expr` in `case` statements





4.15

More About Blocks and Scope



More About Blocks and Scope

- Scope of a variable is the block in which it is defined, from the point of definition to the end of the block
- Usually defined at beginning of function
- May be defined close to first use



Inner Block Variable Definition in Program 4-29

```
16     if (income >= MIN_INCOME)
17     {
18         // Get the number of years at the current job.
19         cout << "How many years have you worked at "
20             << "your current job? ";
21         int years;      // Variable definition
22         cin >> years;
23
24         if (years > MIN_YEARS)
25             cout << "You qualify.\n";
26         else
27         {
28             cout << "You must have been employed for\n"
29                 << "more than " << MIN_YEARS
30                 << " years to qualify.\n";
31         }
32     }
```



Variables with the Same Name

- Variables defined inside { } have local or block scope
- When inside a block within another block, can define variables with the same name as in the outer block.
 - When in inner block, outer definition is not available
 - Not a good idea



Two Variables with the Same Name in Program 4-30

Program 4-30

```
1 // This program uses two variables with the name number.  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main()  
6 {  
7     // Define a variable named number.  
8     int number;  
9  
10    cout << "Enter a number greater than 0: ";  
11    cin >> number;  
12    if (number > 0)  
13    {  
14        int number; // Another variable named number.  
15        cout << "Now enter another number: ";  
16        cin >> number;  
17        cout << "The second number you entered was "  
18            << number << endl;  
19    }  
20    cout << "Your first number was " << number << endl;  
21    return 0;  
22 }
```

Program Output with Example Input Shown in Bold

Enter a number greater than 0: **2 [Enter]**

Now enter another number: **7 [Enter]**

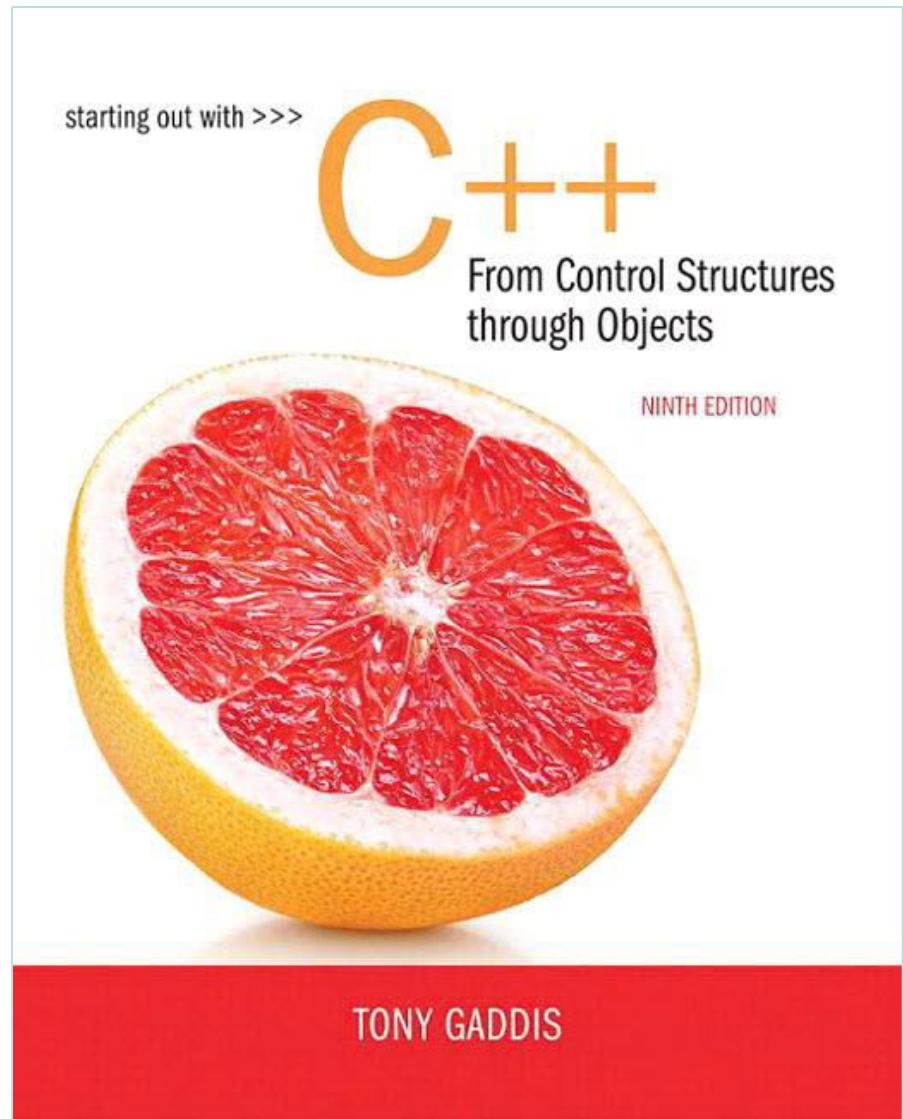
The second number you entered was **7**

Your first number was **2**



Chapter 5:

Loops and Files



The Increment and Decrement Operators

- `++` is the increment operator.

It adds one to a variable.

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

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

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



The Increment and Decrement Operators

- -- is the decrement operator.

It subtracts one from a variable.

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

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

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



Increment and Decrement Operators in Program 5-1

Program 5-1

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

Continued...



Increment and Decrement Operators in Program 5-1

Program 5-1 *(continued)*

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

Program Output

The variable num is 4

I will now increment num.

Now the variable num is 5

I will increment num again.

Now the variable num is 6

I will now decrement num.

Now the variable num is 5

I will decrement num again.

Now the variable num is 4



Prefix vs. Postfix

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



Prefix vs. Postfix - Examples

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



Notes on Increment and Decrement

- Orange icon: Can be used in expressions:

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

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

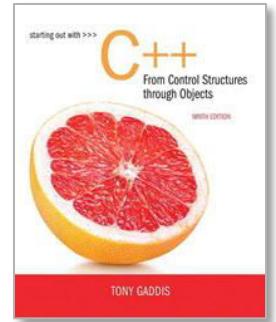
```
result = (num1 + num2) ++;
```

- Orange icon: Can be used in relational expressions:

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

pre- and post-operations will cause different comparisons





5.2

Introduction to Loops: The `while` Loop



Introduction to Loops: The `while` Loop

- Loop: a control structure that causes a statement or statements to repeat
- General format of the `while` loop:

while (*expression*)

statement;

- *statement*; can also be a block of statements enclosed in { }



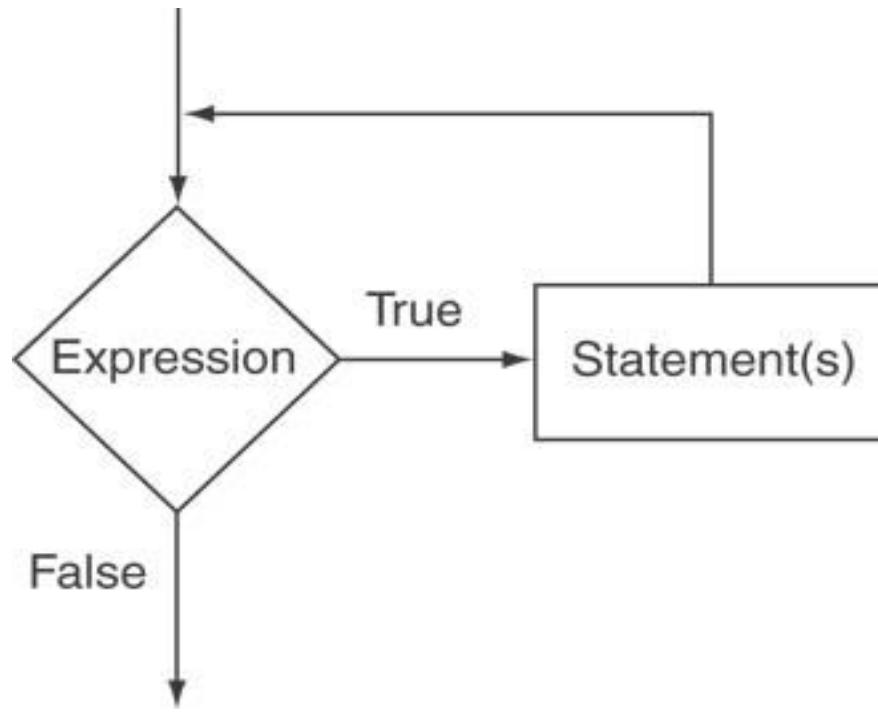
The while Loop – How It Works

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

- *expression* is evaluated
 - if true, then *statement* is executed, and *expression* is evaluated again
 - if false, then the loop is finished and program statements following *statement* execute



The Logic of a while Loop



The while loop in Program 5-3

Program 5-3

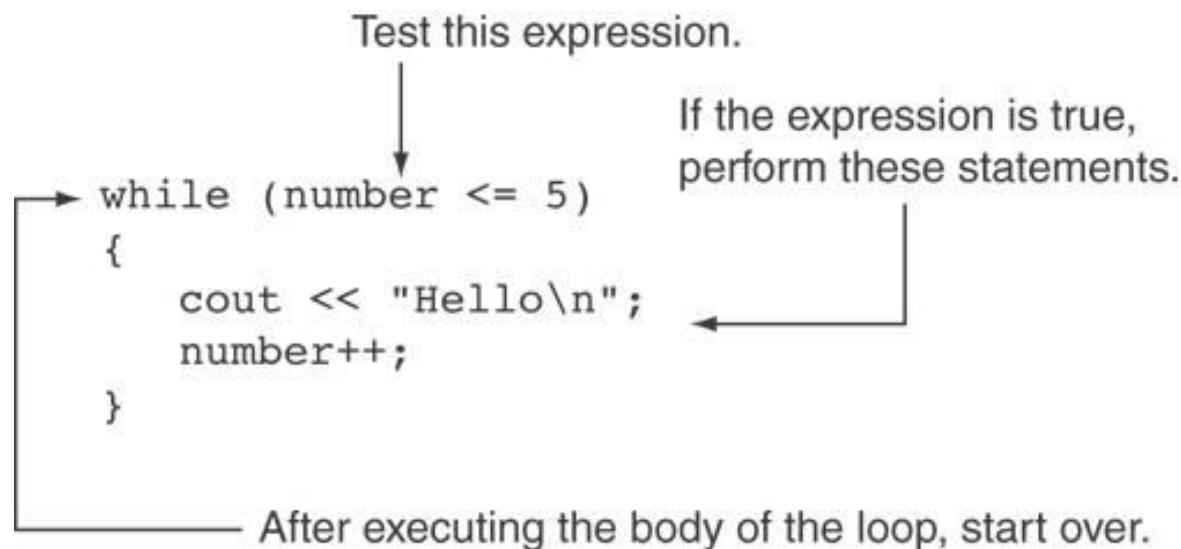
```
1 // This program demonstrates a simple while loop.  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main()  
6 {  
7     int number = 1;  
8  
9     while (number <= 5)  
10    {  
11        cout << "Hello\n";  
12        number++;  
13    }  
14    cout << "That's all!\n";  
15    return 0;  
16 }
```

Program Output

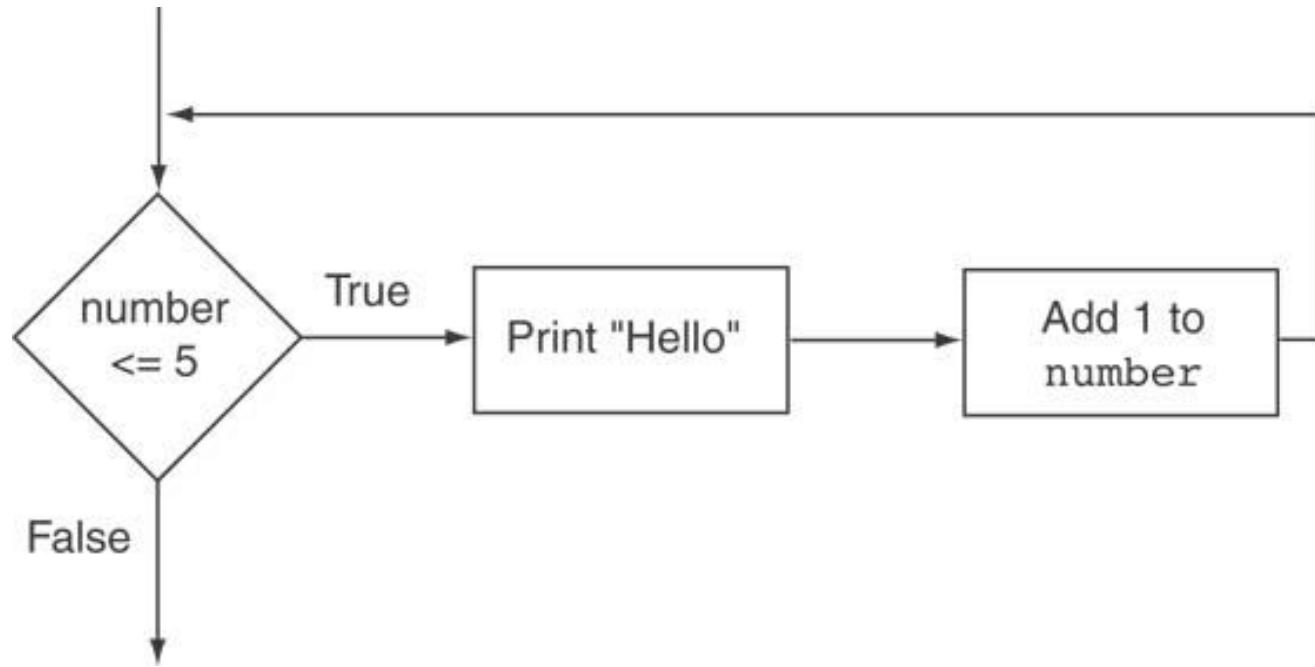
```
Hello  
Hello  
Hello  
Hello  
Hello  
That's all!
```



How the `while` Loop in Program 5-3 Lines 9 through 13 Works



Flowchart of the while Loop in Program 5-3



The `while` Loop is a Pretest Loop

expression is evaluated *before* the loop executes. The following loop will never execute:

```
int number = 6;  
while (number <= 5)  
{  
    cout << "Hello\n";  
    number++;  
}
```



Watch Out for Infinite Loops

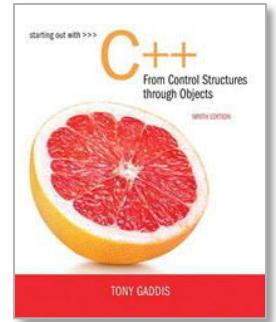
- The loop must contain code to make *expression* become false
- Otherwise, the loop will have no way of stopping
- Such a loop is called an *infinite loop*, because it will repeat an infinite number of times



Example of an Infinite Loop

```
int number = 1;  
while (number <= 5)  
{  
    cout << "Hello\n";  
}
```





5.3

Using the `while` Loop for Input Validation



Using the `while` Loop for Input Validation

- Input validation is the process of inspecting data that is given to the program as input and determining whether it is valid.
- The while loop can be used to create input routines that reject invalid data, and repeat until valid data is entered.



Using the `while` Loop for Input Validation

- Here's the general approach, in pseudocode:

Read an item of input.
While the input is invalid
 Display an error message.
 Read the input again.
End While

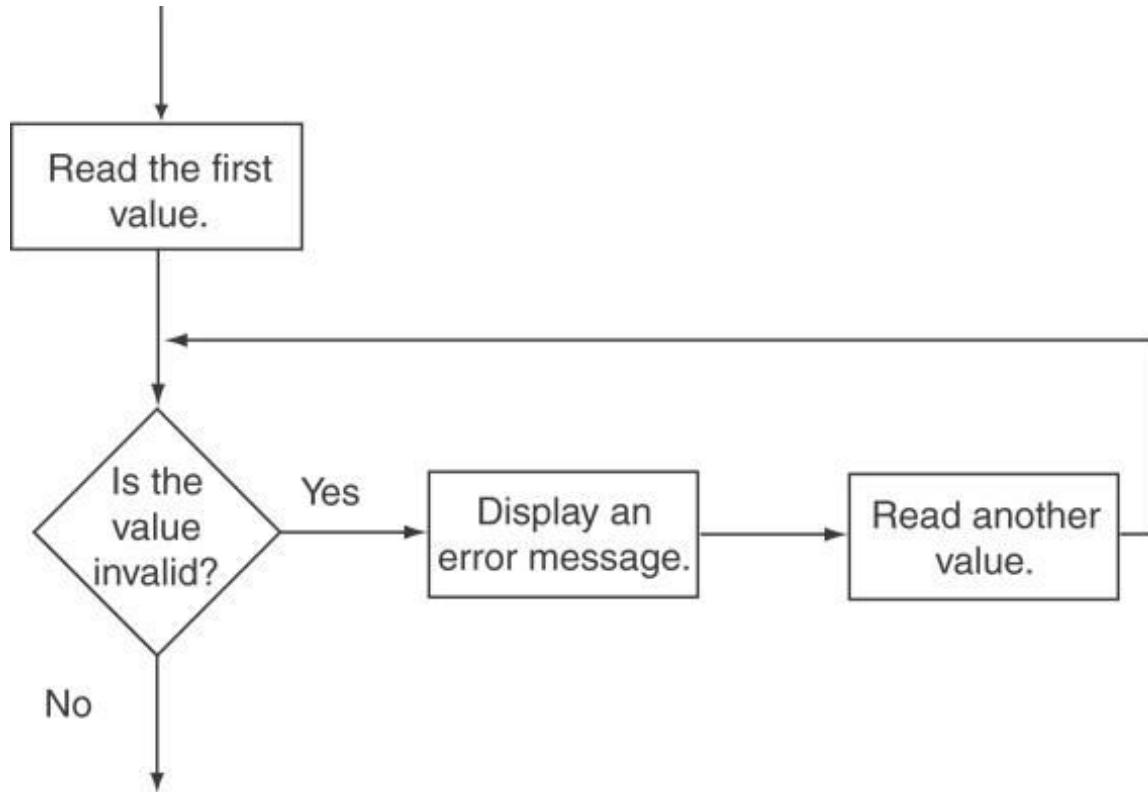


Input Validation Example

```
cout << "Enter a number less than 10: ";
cin >> number;
while (number >= 10)
{
    cout << "Invalid Entry!"
        << "Enter a number less than 10: ";
    cin >> number;
}
```



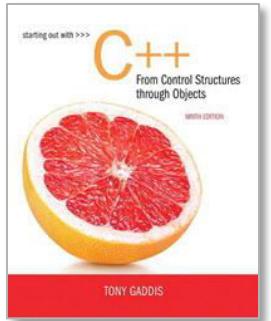
Flowchart for Input Validation



Input Validation in Program 5-5

```
20 // Get the number of players per team.  
21 cout << "How many players do you wish per team? ";  
22 cin >> teamPlayers;  
23  
24 // Validate the input.  
25 while (teamPlayers < MIN_PLAYERS || teamPlayers > MAX_PLAYERS)  
26 {  
27     // Explain the error.  
28     cout << "You should have at least " << MIN_PLAYERS  
29         << " but no more than " << MAX_PLAYERS << " per team.\n";  
30  
31     // Get the input again.  
32     cout << "How many players do you wish per team? ";  
33     cin >> teamPlayers;  
34 }  
35  
36 // Get the number of players available.  
37 cout << "How many players are available? ";  
38 cin >> players;  
39  
40 // Validate the input.  
41 while (players <= 0)  
42 {  
43     // Get the input again.  
44     cout << "Please enter 0 or greater: ";  
45     cin >> players;  
46 }
```





5.4

Counters



Counters

- Orange Counter: a variable that is incremented or decremented each time a loop repeats
- Orange Can be used to control execution of the loop (also known as the loop control variable)
- Orange Must be initialized before entering loop



A Counter Variable Controls the Loop in Program 5-6

Program 5-6

```
1 // This program displays a list of numbers and
2 // their squares.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     const int MIN_NUMBER = 1,      // Starting number to square
9                  MAX_NUMBER = 10;    // Maximum number to square
10
11    int num = MIN_NUMBER;        // Counter
12
13    cout << "Number Number Squared\n";
14    cout << "-----\n";
```

Continued...



A Counter Variable Controls the Loop in Program 5-6

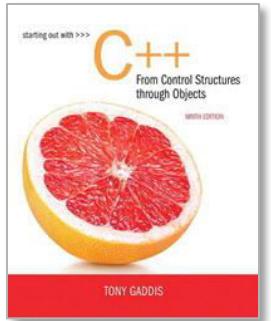
```
15     while (num <= MAX_NUMBER)
16     {
17         cout << num << "\t\t" << (num * num) << endl;
18         num++; //Increment the counter.
19     }
20     return 0;
21 }
```

Program Output

Number Number Squared

Number	Number Squared
<hr/>	
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100





5.5

The do-while Loop



The do-while Loop

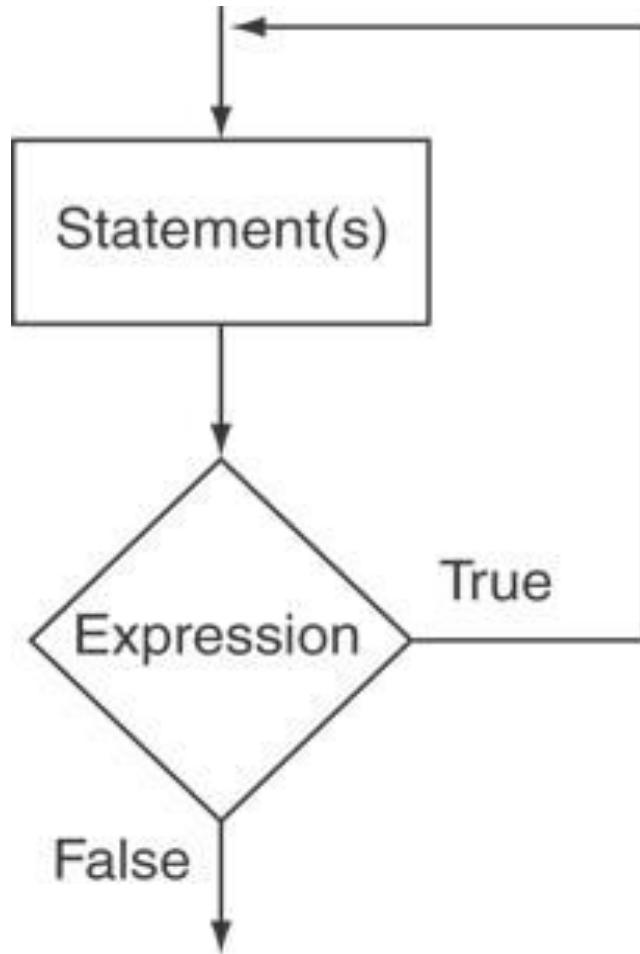
- do-while: a posttest loop – execute the loop, then test the expression
- General Format:

```
do  
    statement; // or block in { }  
    while (expression);
```

- Note that a semicolon is required after (expression)



The Logic of a do-while Loop



An Example do-while Loop

```
int x = 1;  
do  
{  
    cout << x << endl;  
} while(x < 0);
```

Although the test expression is false, this loop will execute one time because do-while is a posttest loop.



A do-while Loop in Program 5-7

Program 5-7

```
1 // This program averages 3 test scores. It repeats as
2 // many times as the user wishes.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int score1, score2, score3; // Three scores
9     double average;           // Average score
10    char again;              // To hold Y or N input
11
12    do
13    {
14        // Get three scores.
15        cout << "Enter 3 scores and I will average them: ";
16        cin >> score1 >> score2 >> score3;
17
18        // Calculate and display the average.
19        average = (score1 + score2 + score3) / 3.0;
20        cout << "The average is " << average << ".\n";
21
22        // Does the user want to average another set?
23        cout << "Do you want to average another set? (Y/N) ";
24        cin >> again;
25    } while (again == 'Y' || again == 'y');
26    return 0;
27 }
```

Continued...



A do-while Loop in Program 5-7

Program Output with Example Input Shown in Bold

Enter 3 scores and I will average them: **80 90 70 [Enter]**

The average is 80.

Do you want to average another set? (Y/N) **y [Enter]**

Enter 3 scores and I will average them: **60 75 88 [Enter]**

The average is 74.3333.

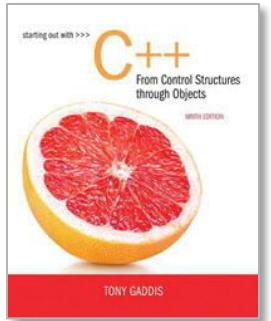
Do you want to average another set? (Y/N) **n [Enter]**



do-while Loop Notes

- Loop always executes at least once
- Execution continues as long as *expression* is true, stops repetition when *expression* becomes false
- Useful in menu-driven programs to bring user back to menu to make another choice
(see Program 5-8 on pages 245-246)





5.6

The for Loop



The for Loop

- Useful for counter-controlled loop

- General Format:

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

- No semicolon after the update expression or after the)



for Loop - Mechanics

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

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

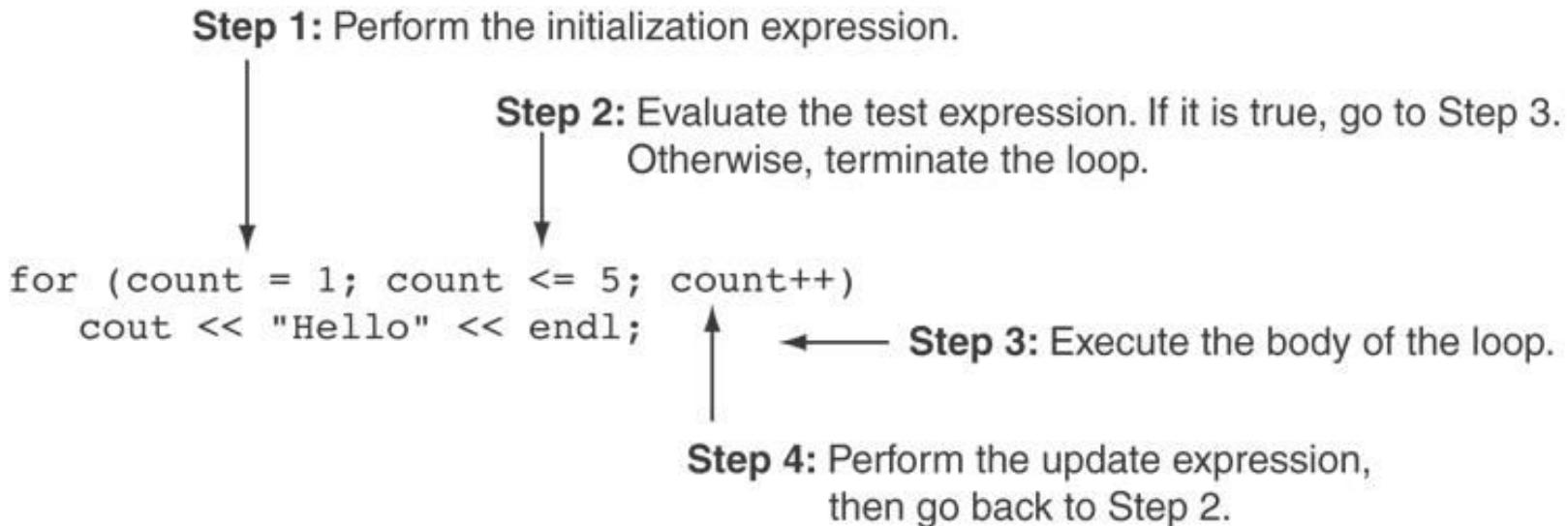


for Loop - Example

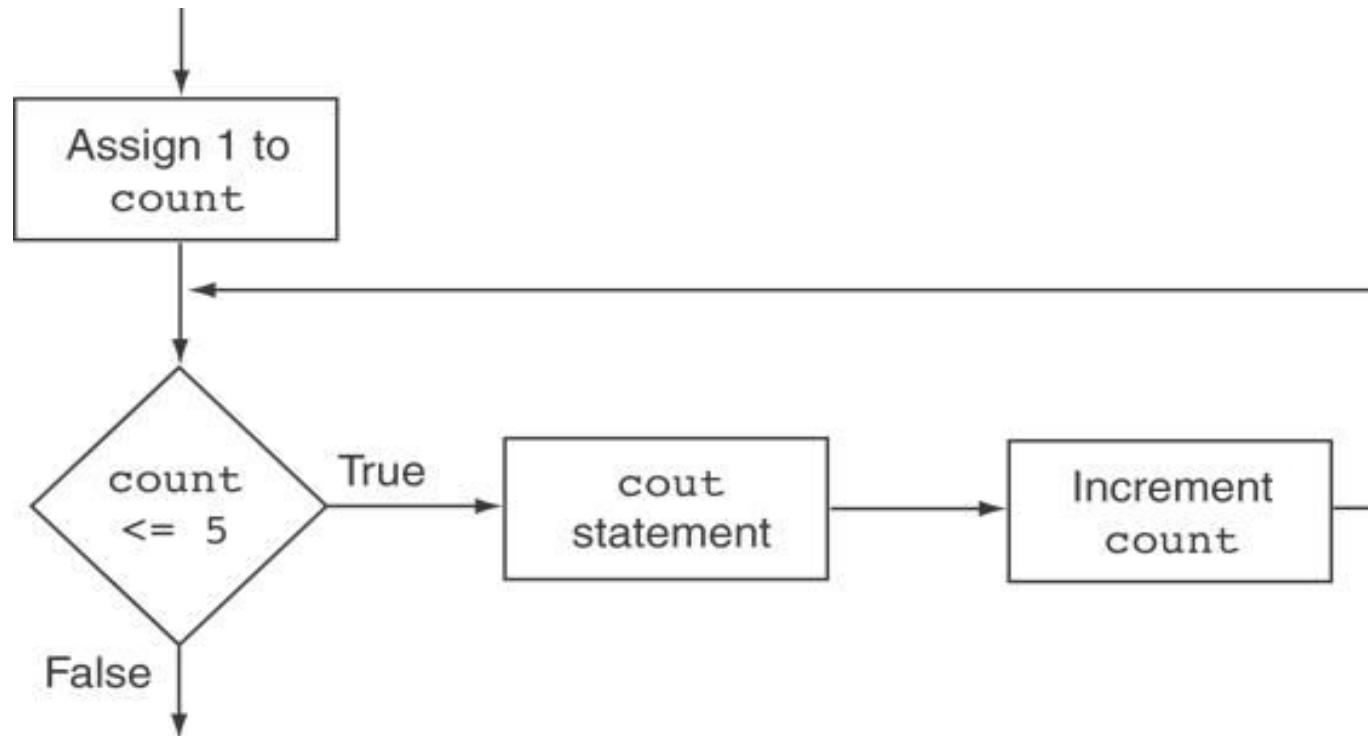
```
int count;  
  
for (count = 1; count <= 5; count++)  
    cout << "Hello" << endl;
```



A Closer Look at the Previous Example



Flowchart for the Previous Example



A for Loop in Program 5-9

Program 5-9

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

Continued...



A for Loop in Program 5-9

Program Output

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

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



A Closer Look at Lines 15 through 16 in Program 5-9

Step 1: Perform the initialization expression.

Step 2: Evaluate the test expression.
If it is true, go to Step 3.
Otherwise, terminate the loop.

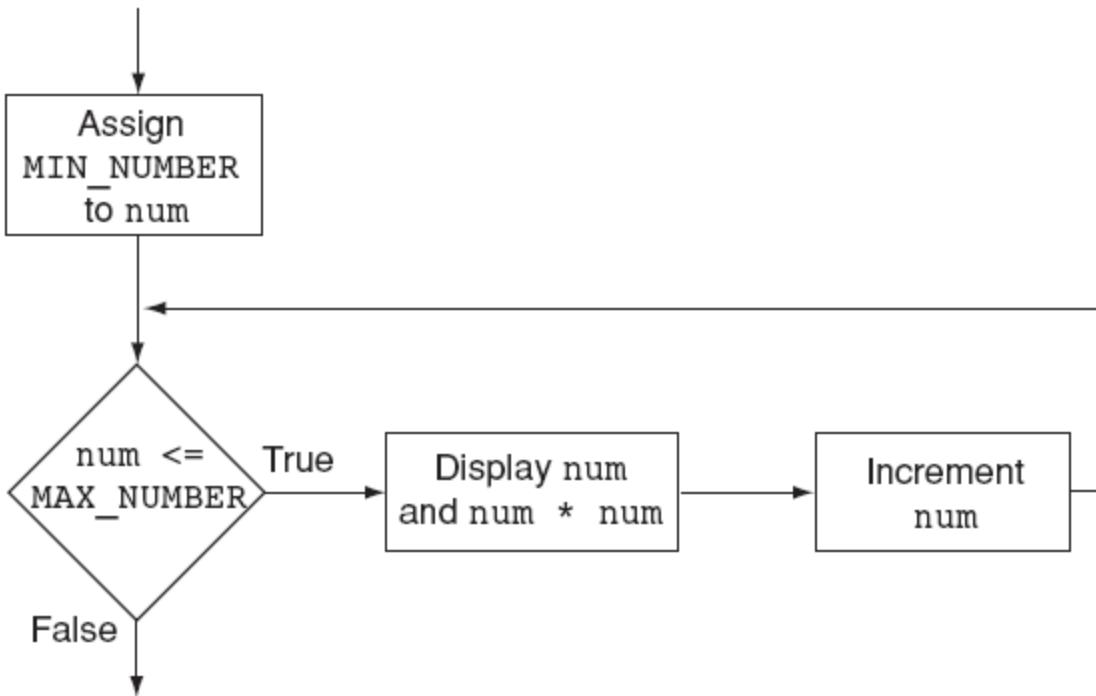
Step 4: Perform the update expression, then go back to Step 2.

```
for (num = MIN_NUMBER; num <= MAX_NUMBER; num++)  
    cout << num << "\t\t" << (num * num) << endl;
```

Step 3: Execute the body of the loop.



Flowchart for Lines 15 through 16 in Program 5-9



When to Use the `for` Loop

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



The for Loop is a Pretest Loop

- The for loop tests its test expression before each iteration, so it is a pretest loop.
- The following loop will never iterate:

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



for Loop - Modifications

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

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

Initialization Expression

for Loop - Modifications

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

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

Test Expression



for Loop - Modifications

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

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



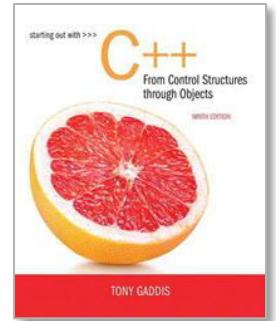
for Loop - Modifications

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

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

The scope of the variable num is the for loop.





5.7

Keeping a Running Total



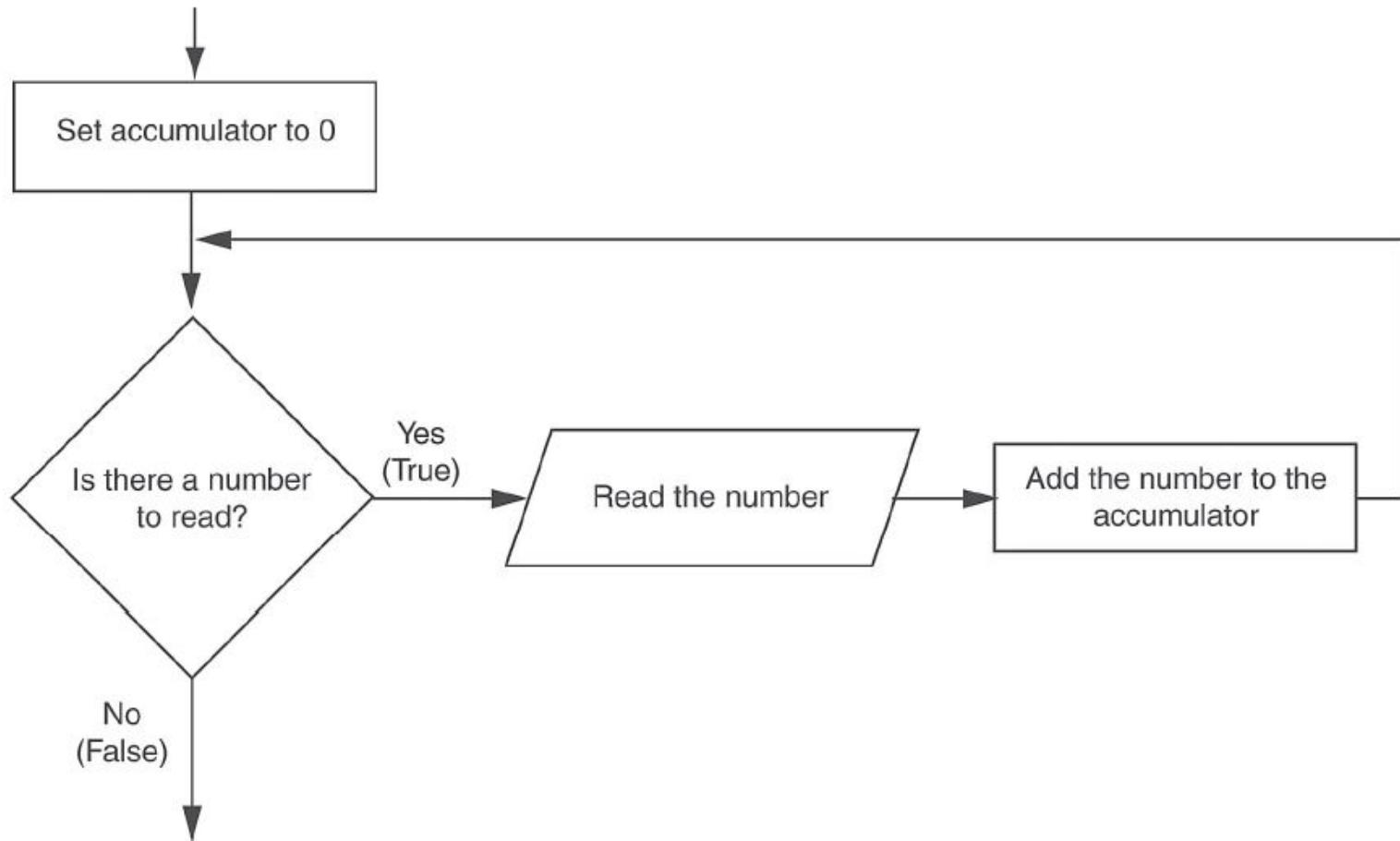
Keeping a Running Total

- orange running total: accumulated sum of numbers from each repetition of loop
- orange accumulator: variable that holds running total

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



Logic for Keeping a Running Total



A Running Total in Program 5-12

Program 5-12

```
1 // This program takes daily sales amounts over a period of time
2 // and calculates their total.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     int days;           // Number of days
10    double total = 0.0; // Accumulator, initialized with 0
11
12    // Get the number of days.
13    cout << "For how many days do you have sales amounts? ";
```

Continued...



A Running Total in Program 5-12

```
14     cin >> days;  
15  
16     // Get the sales for each day and accumulate a total.  
17     for (int count = 1; count <= days; count++)  
18     {  
19         double sales;  
20         cout << "Enter the sales for day " << count << ": ";  
21         cin >> sales;  
22         total += sales; // Accumulate the running total.  
23     }  
24  
25     // Display the total sales.  
26     cout << fixed << showpoint << setprecision(2);  
27     cout << "The total sales are $" << total << endl;  
28     return 0;  
29 }
```

Program Output with Example Input Shown in Bold

For how many days do you have sales amounts? **5**

Enter the sales for day 1: **489.32**

Enter the sales for day 2: **421.65**

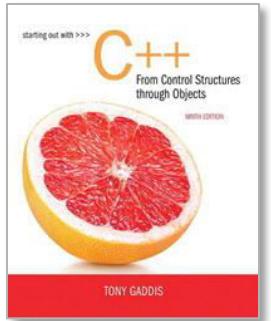
Enter the sales for day 3: **497.89**

Enter the sales for day 4: **532.37**

Enter the sales for day 5: **506.92**

The total sales are \$2448.15





5.8

Sentinels



Sentinels

- sentinel: value in a list of values that indicates end of data
- Special value that cannot be confused with a valid value, e.g., -999 for a test score
- Used to terminate input when user may not know how many values will be entered



A Sentinel in Program 5-13

Program 5-13

```
1 // This program calculates the total number of points a
2 // soccer team has earned over a series of games. The user
3 // enters a series of point values, then -1 when finished.
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     int game = 1,      // Game counter
10    points,          // To hold a number of points
11    total = 0;        // Accumulator
12
13    cout << "Enter the number of points your team has earned\n";
14    cout << "so far in the season, then enter -1 when finished.\n\n";
15    cout << "Enter the points for game " << game << ": ";
16    cin >> points;
17
18    while (points != -1)
19    {
20        total += points;
21        game++;
22        cout << "Enter the points for game " << game << ": ";
23        cin >> points;
24    }
25    cout << "\nThe total points are " << total << endl;
26    return 0;
27 }
```

Continued...



A Sentinel in Program 5-13

Program Output with Example Input Shown in Bold

Enter the number of points your team has earned so far in the season, then enter -1 when finished.

Enter the points for game 1: **7 [Enter]**

Enter the points for game 2: **9 [Enter]**

Enter the points for game 3: **4 [Enter]**

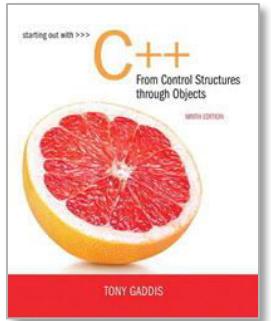
Enter the points for game 4: **6 [Enter]**

Enter the points for game 5: **8 [Enter]**

Enter the points for game 6: **-1 [Enter]**

The total points are 34





5.9

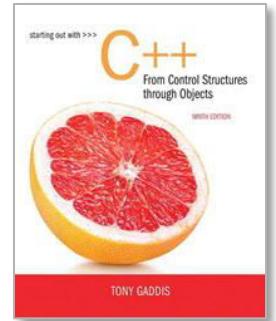
Deciding Which Loop to Use



Deciding Which Loop to Use

- The `while` loop is a conditional pretest loop
 - Iterates as long as a certain condition exists
 - Validating input
 - Reading lists of data terminated by a sentinel
- The `do-while` loop is a conditional posttest loop
 - Always iterates at least once
 - Repeating a menu
- The `for` loop is a pretest loop
 - Built-in expressions for initializing, testing, and updating
 - Situations where the exact number of iterations is known





5.10

Nested Loops



Nested Loops

- A nested loop is a loop inside the body of another loop
- Inner (inside), outer (outside) loops:

```
for (row=1; row<=3; row++) //outer  
  for (col=1; col<=3; col++) //inner  
    cout << row * col << endl;
```



Nested for Loop in Program 5-14

```
26 // Determine each student's average score.  
27 for (int student = 1; student <= numStudents; student++)  
28 {  
29     total = 0;          // Initialize the accumulator.  
30     for (int test = 1; test <= numTests; test++)  
31     {  
32         double score;  
33         cout << "Enter score " << test << " for ";  
34         cout << "student " << student << ": ";  
35         cin >> score;  
36         total += score;  
37     }  
38     average = total / numTests;  
39     cout << "The average score for student " << student;  
40     cout << " is " << average << ".\n\n";  
41 }
```

Inner Loop

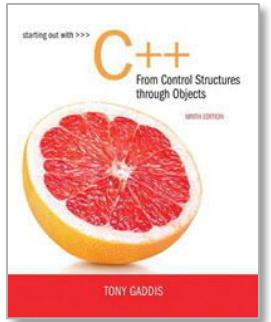
Outer Loop



Nested Loops - Notes

- Orange circle icon: Inner loop goes through all repetitions for each repetition of outer loop
- Orange circle icon: Inner loop repetitions complete sooner than outer loop
- Orange circle icon: Total number of repetitions for inner loop is product of number of repetitions of the two loops.





5.11

Using Files for Data Storage



Using Files for Data Storage

- Can use files instead of keyboard, monitor screen for program input, output
- Allows data to be retained between program runs
- Steps:
 - Open the file
 - Use the file (read from, write to, or both)
 - Close the file



Files: What is Needed

- Use `fstream` header file for file access
- File stream types:

`ifstream` for input from a file

`ofstream` for output to a file

`fstream` for input from or output to a file

- Define file stream objects:

```
ifstream infile;
```

```
ofstream outfile;
```



Opening Files

- Create a link between file name (outside the program) and file stream object (inside the program)
- Use the `open` member function:

```
infile.open("inventory.dat");  
outfile.open("report.txt");
```

- Filename may include drive, path info.
- Output file will be created if necessary; existing file will be erased first
- Input file must exist for `open` to work



Testing for File Open Errors

- Can test a file stream object to detect if an open operation failed:

```
infile.open("test.txt");  
if (!infile)  
{  
    cout << "File open failure!";  
}
```

- Can also use the fail member function



Using Files

- Orange icon: Can use output file object and << to send data to a file:

```
outfile << "Inventory report";
```

- Orange icon: Can use input file object and >> to copy data from file to variables:

```
infile >> partNum;
```

```
infile >> qtyInStock >>  
qtyOnOrder;
```



Using Loops to Process Files

- The stream extraction operator `>>` returns true when a value was successfully read, false otherwise
- Can be tested in a `while` loop to continue execution as long as values are read from the file:

```
while (inputFile >> number) ...
```



Closing Files

- Orange Use the `close` member function:

```
infile.close();
```

```
outfile.close();
```

- Orange Don't wait for operating system to close files at program end:

- Orange may be limit on number of open files

- Orange may be buffered output data waiting to send to file



Letting the User Specify a Filename

- In many cases, you will want the user to specify the name of a file for the program to open.
- In C++ 11, you can pass a `string` object as an argument to a file stream object's `open` member function.



Letting the User Specify a Filename in Program 5-24

Program 5-24

```
1 // This program lets the user enter a filename.
2 #include <iostream>
3 #include <string>
4 #include <fstream>
5 using namespace std;
6
7 int main()
8 {
9     ifstream inputFile;
10    string filename;
11    int number;
12
13    // Get the filename from the user.
14    cout << "Enter the filename: ";
15    cin >> filename;
16
17    // Open the file.
18    inputFile.open(filename);
19
20    // If the file successfully opened, process it.
21    if (inputFile)
```

Continued...



Letting the User Specify a Filename in Program 5-24

```
22     {
23         // Read the numbers from the file and
24         // display them.
25         while (inputFile >> number)
26         {
27             cout << number << endl;
28         }
29
30         // Close the file.
31         inputFile.close();
32     }
33     else
34     {
35         // Display an error message.
36         cout << "Error opening the file.\n";
37     }
38     return 0;
39 }
```

Program Output with Example Input Shown in Bold

```
Enter the filename: ListOfNumbers.txt [Enter]
100
200
300
400
500
600
700
```



Using the `c_str` Member Function in Older Versions of C++

- Prior to C++ 11, the `open` member function requires that you pass the name of the file as a null-terminated string, which is also known as a C-string.
- *String literals are stored in memory as null-terminated C-strings, but string objects are not.*



Using the `c_str` Member Function in Older Versions of C++

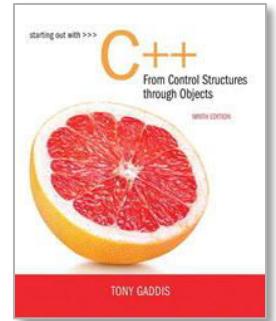
- `string` objects have a member function named `c_str`
 - It returns the contents of the object formatted as a null-terminated C-string.
 - Here is the general format of how you call the `c_str` function:

```
stringObject.c_str()
```

- Line 18 in Program 5-24 could be rewritten in the following manner:

```
inputFile.open(filename.c_str());
```





5.12

Breaking and Continuing a Loop



Breaking Out of a Loop

- Orange circle icon: Can use `break` to terminate execution of a loop
- Orange circle icon: Use sparingly if at all – makes code harder to understand and debug
- Orange circle icon: When used in an inner loop, terminates that loop only and goes back to outer loop



The `continue` Statement

- Can use `continue` to go to end of loop and prepare for next repetition
 - while, do-while loops: go to test, repeat loop if test passes
 - for loop: perform update step, then test, then repeat loop if test passes
- Use sparingly – like `break`, can make program logic hard to follow

Chapter 6:

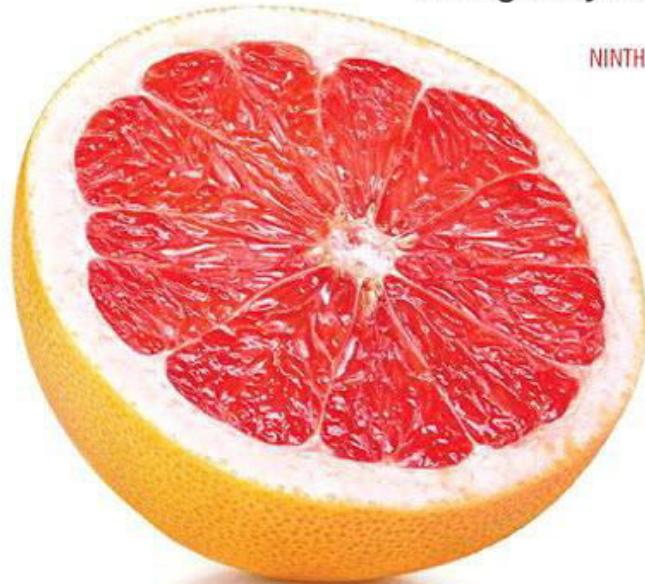
Functions

starting out with >>>

C++

From Control Structures
through Objects

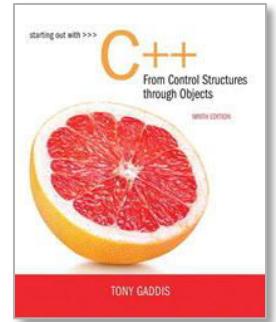
NINTH EDITION



TONY GADDIS



Pearson Copyright © 2018, 2015, 2012, 2009 Pearson Education, Inc. All rights reserved.



6.1

Modular Programming



Modular Programming

- Modular programming: breaking a program up into smaller, manageable functions or modules
- Function: a collection of statements to perform a task
- Motivation for modular programming:
 - Improves maintainability of programs
 - Simplifies the process of writing programs



This program has one long, complex function containing all of the statements necessary to solve a problem.



```
int main()
{
    statement;
    statement;
}
```

In this program the problem has been divided into smaller problems, each of which is handled by a separate function.



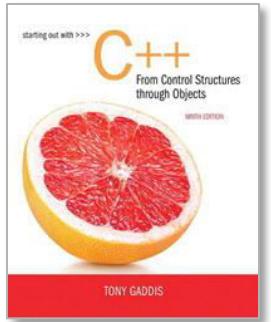
```
int main()
{
    statement;
    statement;
    statement;
}

void function2()
{
    statement;
    statement;
    statement;
}

void function3()
{
    statement;
    statement;
    statement;
}

void function4()
{
    statement;
    statement;
    statement;
}
```





6.2

Defining and Calling Functions



Defining and Calling Functions

- Function call: statement causes a function to execute
- Function definition: statements that make up a function



Function Definition

- Definition includes:

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



Function Definition

```
Return type      Parameter list (This one is empty)  
↓             ↓  
int main ()  
 {           Function body  
     cout << "Hello World\n";  
     return 0;  
 }
```

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



Function Return Type

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

```
int main()
```

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

```
void printHeading()
{
    cout << "Monthly Sales\n";
}
```



Calling a Function

- To call a function, use the function name followed by () and ;

```
printHeading();
```

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



Functions in Program 6-1

Program 6-1

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

Program Output

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



Flow of Control in Program 6-1

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

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

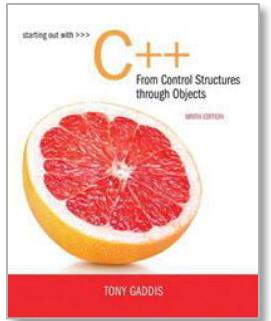
The diagram illustrates the flow of control between the `main()` and `displayMessage()` functions. It consists of two rectangular boxes. The top box contains the `displayMessage()` function definition, which outputs the string "Hello from the function displayMessage.\n". The bottom box contains the `main()` function definition, which outputs "Hello from main.\n", calls the `displayMessage()` function, and then outputs "Back in function main again.\n". Blue arrows indicate the flow of control: one arrow points from the start of `main()` to the start of `displayMessage()`, and another arrow points back from the end of `displayMessage()` to the end of `main()`.



Calling Functions

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





6.3

Function Prototypes



Function Prototypes

- Ways to notify the compiler about a function before a call to the function:
 - Place function definition before calling function's definition
 - Use a function prototype (function declaration) – like the function definition without the body
 - Header: void printHeading()
 - Prototype: void printHeading();



Function Prototypes in Program 6-5

Program 6-5

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

(Program Continues)



Function Prototypes in Program 6-5

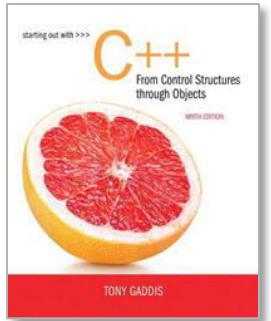
```
18 //*****  
19 // Definition of function first.      *  
20 // This function displays a message.  *  
21 //*****  
22  
23 void first()  
24 {  
25     cout << "I am now inside the function first.\n";  
26 }  
27  
28 //*****  
29 // Definition of function second.      *  
30 // This function displays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



Prototype Notes

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





6.4

Sending Data into a Function



Sending Data into a Function

- Orange icon: Can pass values into a function at time of call:

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

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



A Function with a Parameter Variable

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

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



Function with a Parameter in Program 6-6

Program 6-6

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

(Program Continues)



Function with a Parameter in Program 6-6

Program 6-6 *(continued)*

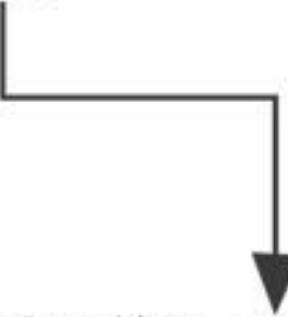
```
16 //*****  
17 // Definition of function displayValue. *  
18 // It uses an integer parameter whose value is displayed. *  
19 //*****  
20  
21 void displayValue(int num)  
22 {  
23     cout << "The value is " << num << endl;  
24 }
```

Program Output

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



Function with a Parameter in Program 6-6

```
displayValue(5);  
  
void displayValue(int num)  
{  
    cout << "The value is " << num << endl;  
}
```

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



Other Parameter Terminology

- A parameter can also be called a formal parameter or a formal argument
- An argument can also be called an actual parameter or an actual argument



Parameters, Prototypes, and Function Headers

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

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



Function Call Notes

- Value of argument is copied into parameter when the function is called
- A parameter's scope is the function which uses it
- Function can have multiple parameters
- There must be a data type listed in the prototype () and an argument declaration in the function header () for each parameter
- Arguments will be promoted/demoted as necessary to match parameters



Passing Multiple Arguments

When calling a function and passing multiple arguments:

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



Passing Multiple Arguments in Program 6-8

Program 6-8

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

(Program Continues)



Passing Multiple Arguments in Program 6-8

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

Program Output with Example Input Shown in Bold

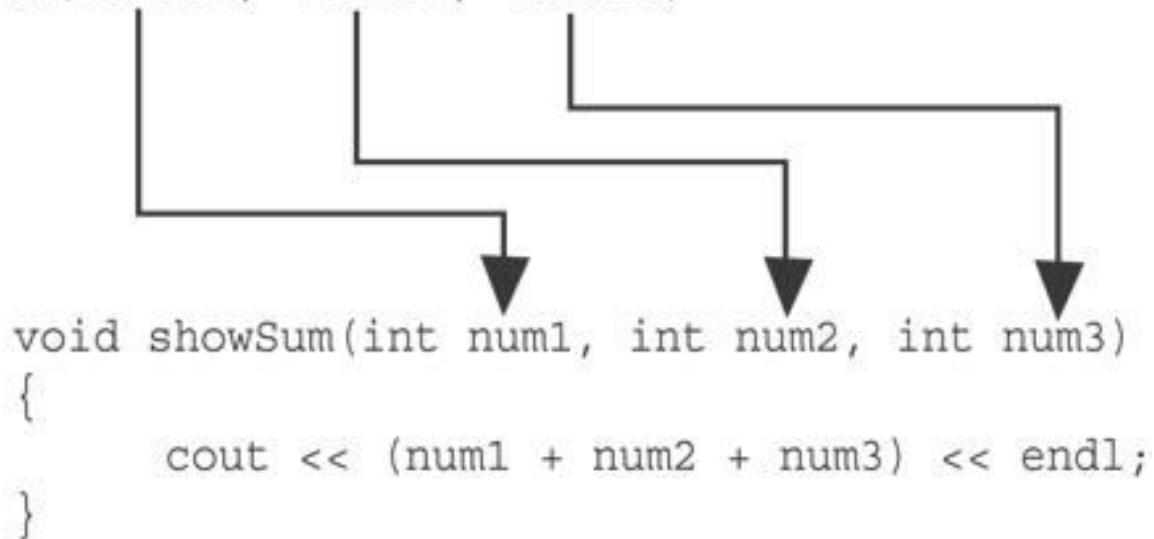
Enter three integers and I will display their sum: **4 8 7 [Enter]**

19



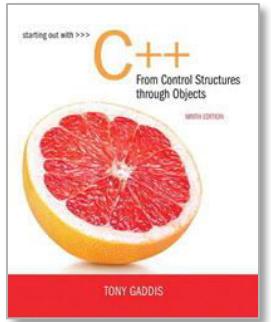
Passing Multiple Arguments in Program 6-8

Function Call → showSum(value1, value2, value3)



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





6.5

Passing Data by Value



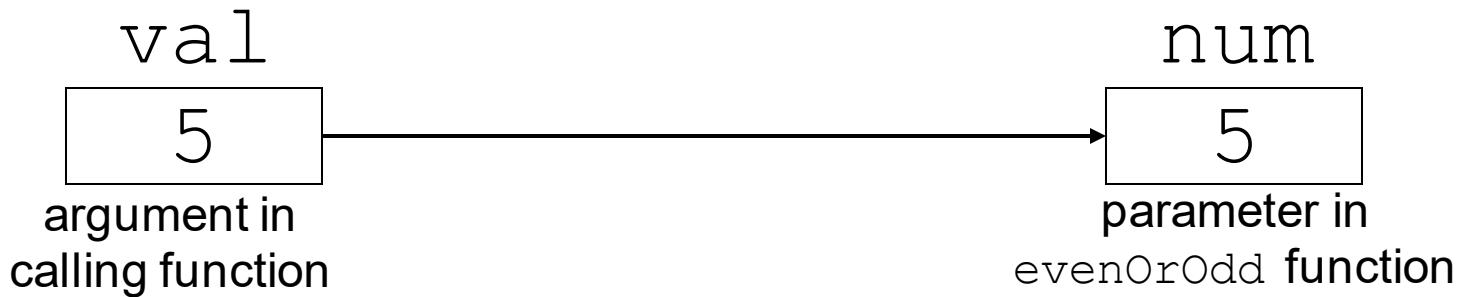
Passing Data by Value

- Pass by value: when an argument is passed to a function, its value is copied into the parameter.
- Changes to the parameter in the function do not affect the value of the argument

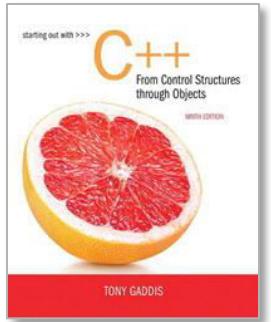


Passing Information to Parameters by Value

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



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



6.6

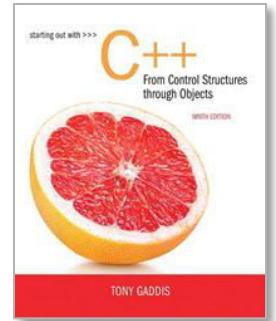
Using Functions in Menu-Driven Programs



Using Functions in Menu-Driven Programs

- Functions can be used
 - to implement user choices from menu
 - to implement general-purpose tasks:
 - Higher-level functions can call general-purpose functions, minimizing the total number of functions and speeding program development time
- See *Program 6-10 in the book*





6.7

The return Statement



The return Statement

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



Performing Division in Program 6-11

Program 6-11

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

(Program Continues)



Performing Division in Program 6-11

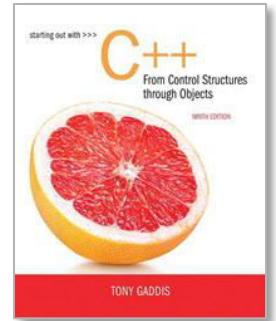
```
20 //*****  
21 // Definition of function divide. *  
22 // Uses two parameters: arg1 and arg2. The function divides arg1*  
23 // by arg2 and shows the result. If arg2 is zero, however, the *  
24 // function returns. *  
25 //*****  
26  
27 void divide(double arg1, double arg2)  
28 {  
29   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

Program Output with Example Input Shown in Bold

Enter two numbers and I will divide the first number by the second number: **12 0 [Enter]**

Sorry, I cannot divide by zero.





6.8

Returning a Value From a Function



Returning a Value From a Function

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

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



Returning a Value From a Function

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

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



A Value-Returning Function

Return Type

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

Value Being Returned



A Value-Returning Function

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

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



Function Returning a Value in Program 6-12

Program 6-12

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

(Program Continues)



Function Returning a Value in Program 6-12

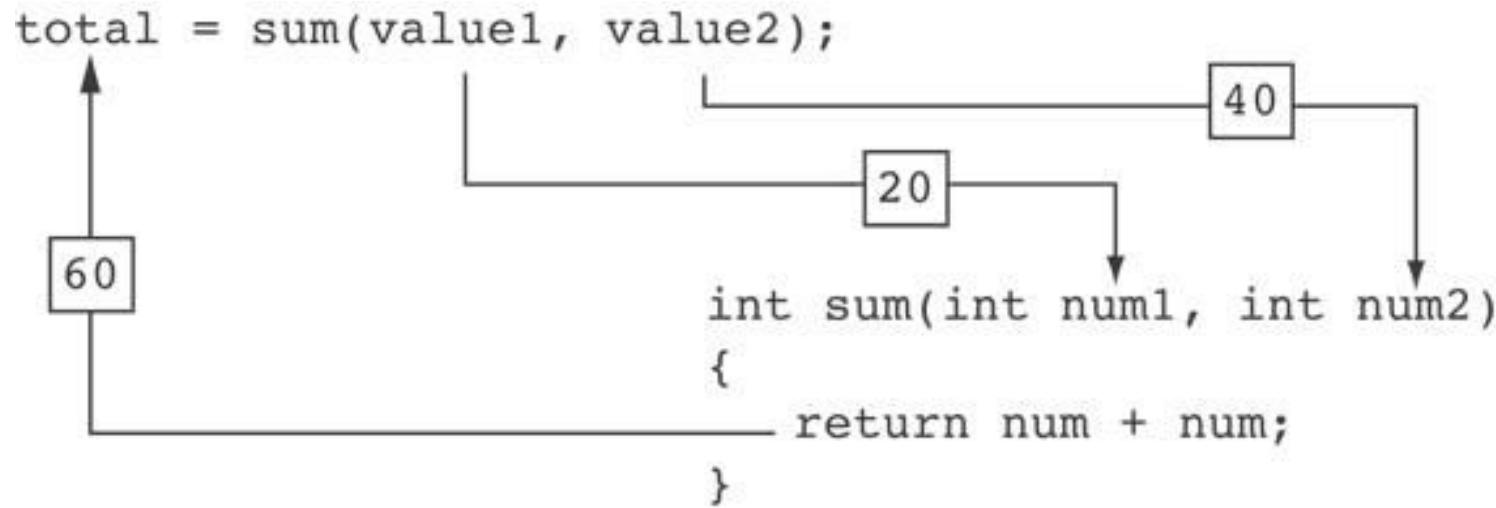
```
24
25 //*****
26 // Definition of function sum. This function returns *
27 // the sum of its two parameters. *
28 //*****
29
30 int sum(int num1, int num2)
31 {
32     return num1 + num2;
33 }
```

Program Output

The sum of 20 and 40 is 60



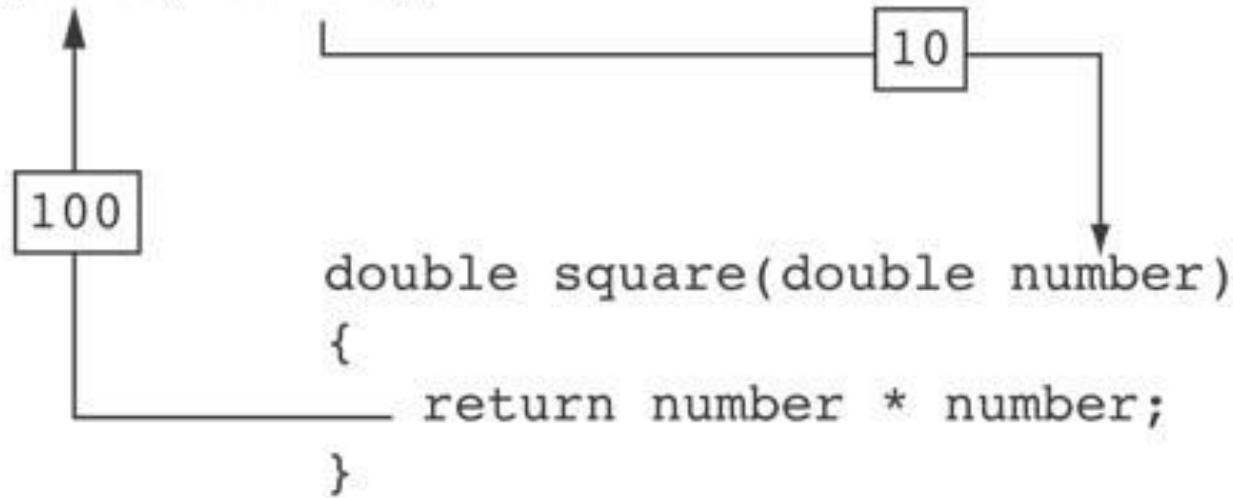
Function Returning a Value in Program 6-12



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

Another Example from Program 6-13

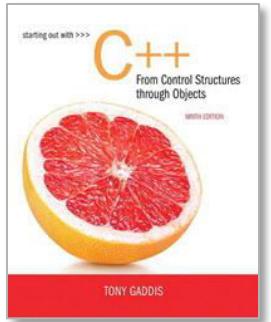
```
area = PI * square(radius);
```



Returning a Value From a Function

- The prototype and the definition must indicate the data type of return value (not void)
- Calling function should use return value:
 - assign it to a variable
 - send it to cout
 - use it in an expression





6.9

Returning a Boolean Value



Returning a Boolean Value

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



Returning a Boolean Value in Program 6-15

Program 6-15

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

(Program Continues)



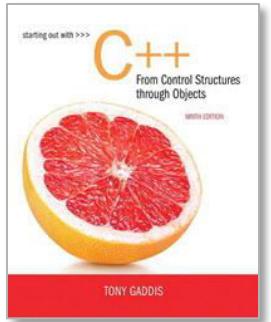
Returning a Boolean Value in Program 6-15

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

Program Output with Example Input Shown in Bold

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





6.10

Local and Global Variables



Local and Global Variables

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



Local Variables in Program 6-16

Program 6-16

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

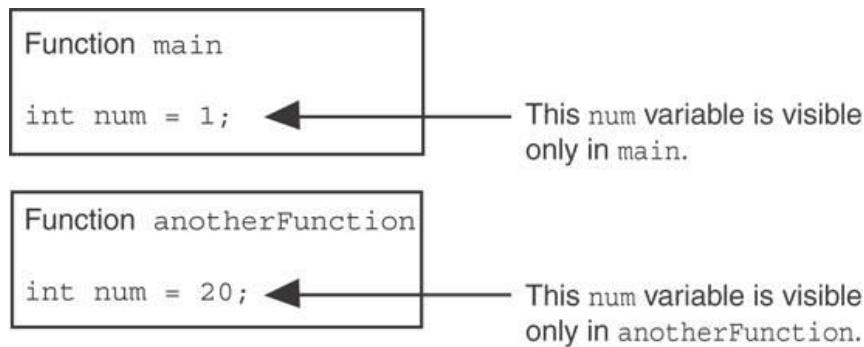


Local Variables in Program 6-16

Program Output

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

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



Local Variable Lifetime

- A function's local variables exist only while the function is executing. This is known as the *lifetime* of a local variable.
- When the function begins, its local variables and its parameter variables are created in memory, and when the function ends, the local variables and parameter variables are destroyed.
- This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.



Global Variables and Global Constants

- A global variable is any variable defined outside all the functions in a program.
- The scope of a global variable is the portion of the program from the variable definition to the end.
- This means that a global variable can be accessed by *all* functions that are defined after the global variable is defined.



Global Variables and Global Constants

- Orange You should avoid using global variables because they make programs difficult to debug.
- Orange Any global that you create should be *global constants*.



Global Constants in Program 6-19

Program 6-19

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

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



Global Constants in Program 6-19

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

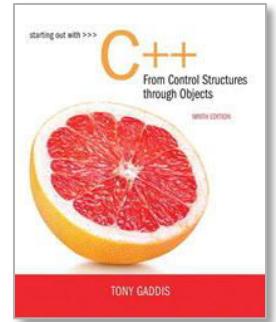
```
29      // Get overtime pay, if any.  
30      if (hours > BASE_HOURS)  
31          overtime = getOvertimePay(hours);  
  
56      // Determine base pay.  
57      if (hoursWorked > BASE_HOURS)  
58          basePay = BASE_HOURS * PAY_RATE;  
59      else  
60          basePay = hoursWorked * PAY_RATE;  
  
75      // Determine overtime pay.  
76      if (hoursWorked > BASE_HOURS)  
77      {  
78          overtimePay = (hoursWorked - BASE_HOURS) *  
79                      PAY_RATE * OT_MULTIPLIER;  
80      }  
81  }
```



Initializing Local and Global Variables

- Local variables are not automatically initialized. They must be initialized by programmer.
- Global variables (not constants) are automatically initialized to 0 (numeric) or NULL (character) when the variable is defined.





6.11

Static Local Variables



Static Local Variables

- Orange icon: Local variables only exist while the function is executing. When the function terminates, the contents of local variables are lost.
- Orange icon: static local variables retain their contents between function calls.
- Orange icon: static local variables are defined and initialized only the first time the function is executed. 0 is the default initialization value.



Local Variables Do Not Retain Values Between Function calls in Program 6-21

Program 6-21

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

(Program Continues)



Local Variables Do Not Retain Values Between Function calls in Program 6-21

Program 6-21 *(continued)*

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

Program Output

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

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



A Different Approach, Using a Static Variable in Program 6-22

Program 6-22

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

(Program Continues)



A Different Approach, Using a Static Variable in Program 6-22

Program 6-22 *(continued)*

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

Program Output

statNum is 0 ← statNum is automatically initialized to 0. Notice that it retains its value between function calls.
statNum is 1
statNum is 2
statNum is 3
statNum is 4



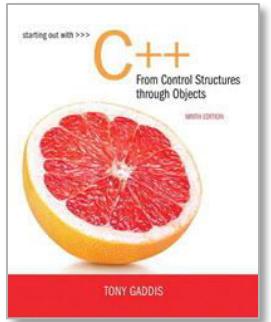
If you do initialize a local static variable, the initialization only happens once. See Program 6-23.

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

Program Output

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





6.12

Default Arguments



Default Arguments

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

- Must be a constant declared in prototype:

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

- Can be declared in header if no prototype

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

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

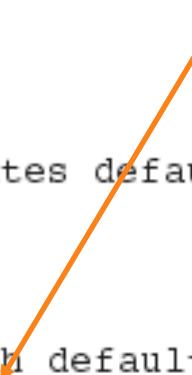


Default Arguments in Program 6-24

Default arguments specified in the prototype

Program 6-24

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



(Program Continues)



Default Arguments in Program 6-24

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

Program Output

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



Default Arguments

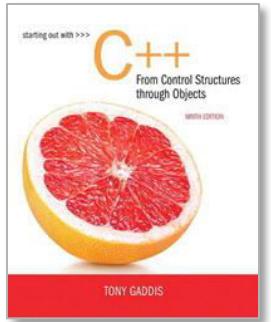
- If not all parameters to a function have default values, the defaultless ones are declared first in the parameter list:

```
int getSum(int, int=0, int=0); // OK  
int getSum(int, int=0, int); // NO
```

- When an argument is omitted from a function call, all arguments after it must also be omitted:

```
sum = getSum(num1, num2); // OK  
sum = getSum(num1, , num3); // NO
```





6.13

Using Reference Variables as Parameters



Using Reference Variables as Parameters

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



Passing by Reference

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

```
void getDimensions(int&, int&);
```
- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters *by reference*

Passing a Variable By Reference in Program 6-25

Program 6-25

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

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

Here we are passing value by reference.

(Program Continues)



Passing a Variable By Reference in Program 6-25

The & also appears here in the function header.

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

Program Output

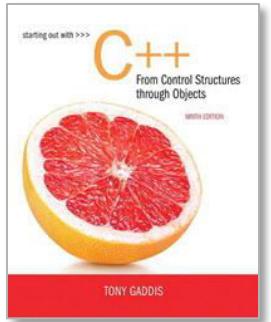
```
In main, value is 4  
Now calling doubleNum...  
Now back in main. value is 8
```



Reference Variable Notes

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





6.14

Overloading Functions



Overloading Functions

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



Function Overloading Examples

Using these overloaded functions,

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

the compiler will use them as follows:

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



Function Overloading in Program 6-27

Program 6-27

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

Passing an int

Passing a double

(Program Continues)



Function Overloading in Program 6-27

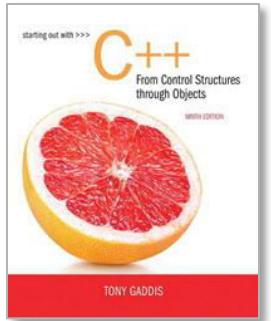
```
26 //*****
27 // Definition of overloaded function square. *
28 // This function uses an int parameter, number. It returns the *
29 // square of number as an int. *
30 //*****
31
32 int square(int number)
33 {
34     return number * number;
35 }
36
37 //*****
38 // Definition of overloaded function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

Program Output with Example Input Shown in Bold

Enter an integer and a floating-point value: **12 4.2 [Enter]**

Here are their squares: 144 and 17.64





6.15

The `exit()` Function



The exit () Function

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



The exit () Function

- Example:

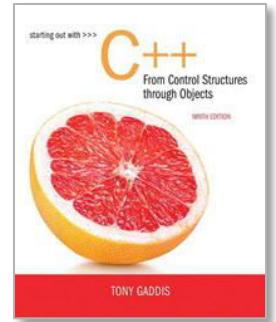
```
exit(0);
```

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

```
exit(EXIT_SUCCESS);
```

```
exit(EXIT_FAILURE);
```





6.16

Stubs and Drivers



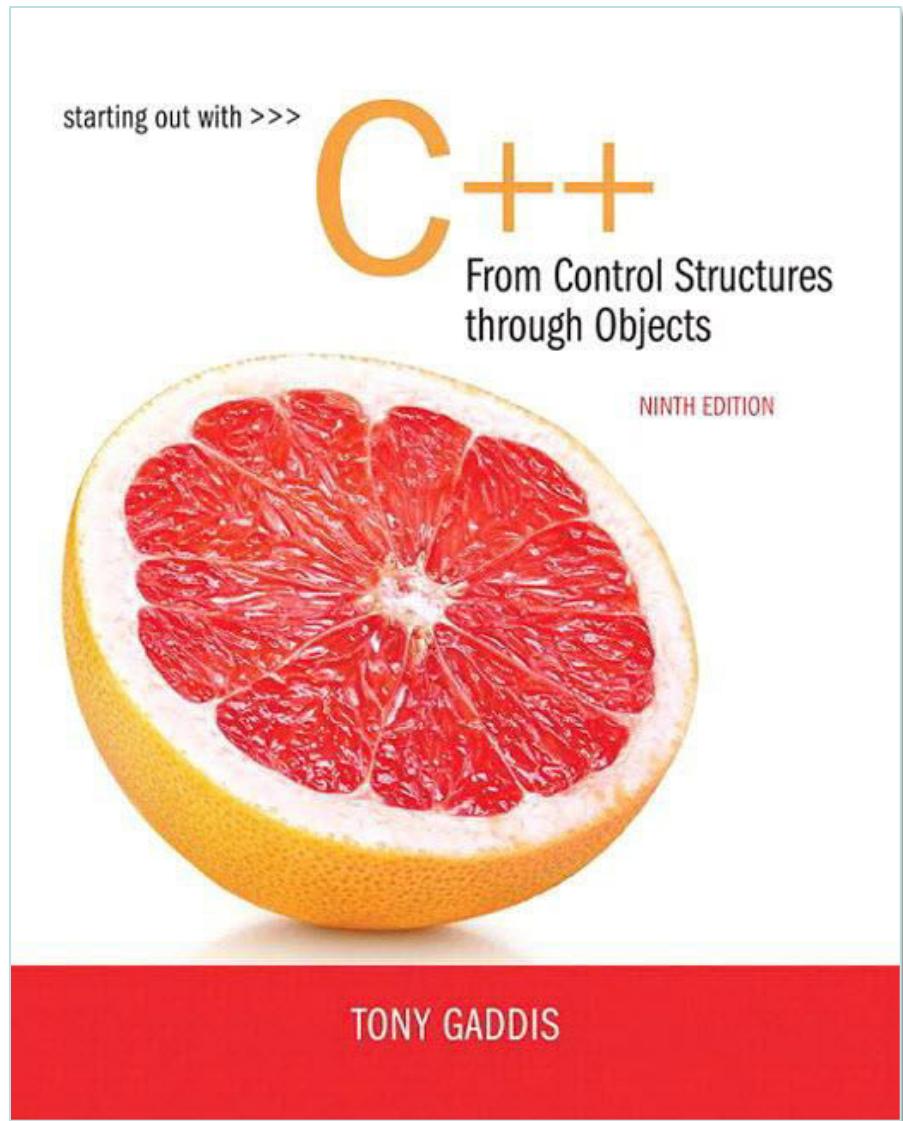
Stubs and Drivers

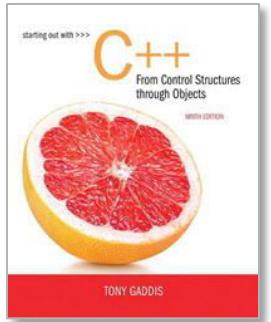
- Useful for testing and debugging program and function logic and design
- Stub: A dummy function used in place of an actual function
 - Usually displays a message indicating it was called. May also display parameters
- Driver: A function that tests another function by calling it
 - Various arguments are passed and return values are tested



Chapter 7:

Arrays and Vectors





7.1

Arrays Hold Multiple Values



Arrays Hold Multiple Values

- Array: variable that can store multiple values of the same type
- Values are stored in adjacent memory locations
- Declared using [] operator:

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

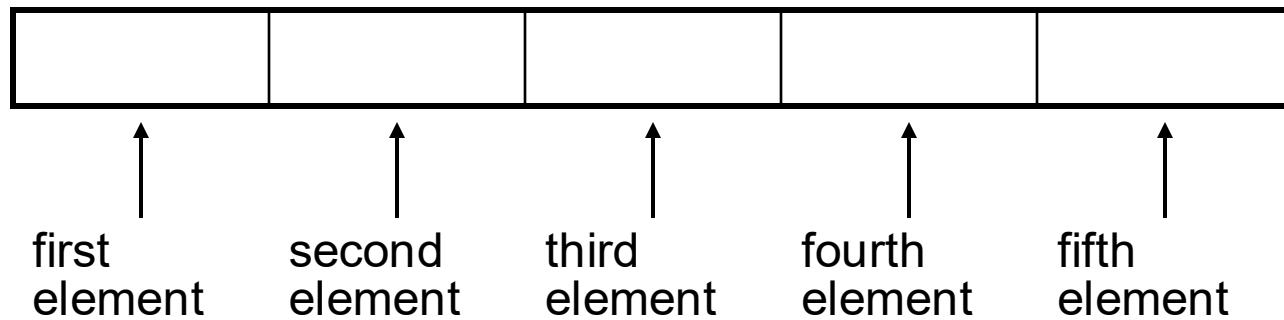


Array - Memory Layout

- The definition:

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

allocates the following memory:



Array Terminology

In the definition `int tests[5];`

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



Array Terminology

- The size of an array is:
 - the total number of bytes allocated for it
 - $(\text{number of elements}) * (\text{number of bytes for each element})$
- Examples:
 - int tests[5] is an array of 20 bytes, assuming 4 bytes for an int
 - long double measures[10] is an array of 80 bytes, assuming 8 bytes for a long double



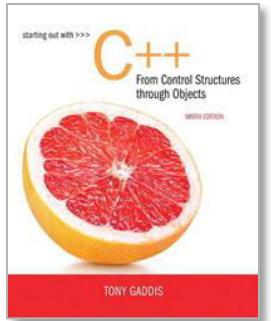
Size Declarators

- Named constants are commonly used as size declarators.

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

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





7.2

Accessing Array Elements



Accessing Array Elements

- Each element in an array is assigned a unique *subscript*.
- Subscripts start at 0

subscripts:

0

1

2

3

4



Accessing Array Elements

- The last element's subscript is $n-1$ where n is the number of elements in the array.

subscripts:

0

1

2

3

4



Accessing Array Elements

- Array elements can be used as regular variables:

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

- Arrays must be accessed via individual elements:

```
cout << tests; // not legal
```



Accessing Array Elements in Program 7-1

Program 7-1

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

(Program Continues)



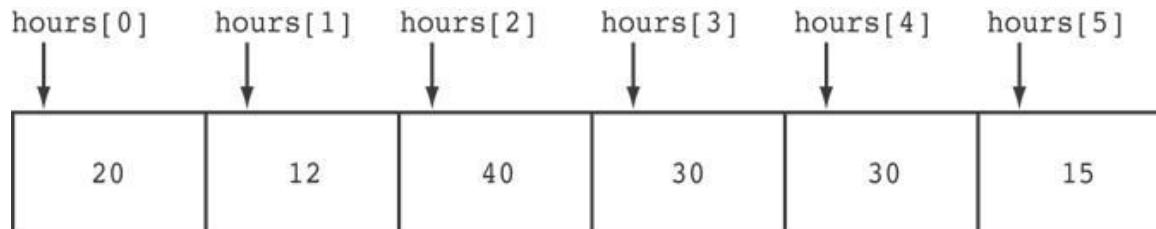
Accessing Array Elements in Program 7-1

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

Program Output with Example Input Shown in Bold

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

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



Accessing Array Contents

- Orange icon: Can access element with a constant or literal subscript:

```
cout << tests[3] << endl;
```

- Orange icon: Can use integer expression as subscript:

```
int i = 5;
```

```
cout << tests[i] << endl;
```



Using a Loop to Step Through an Array

- Example – The following code defines an array, numbers, and assigns 99 to each element:

```
const int ARRAY_SIZE = 5;  
int numbers [ARRAY_SIZE];  
  
for (int count = 0; count < ARRAY_SIZE; count++)  
    numbers[count] = 99;
```



A Closer Look At the Loop

The variable count starts at 0, which is the first valid subscript value.

```
for (count = 0; count < ARRAY_SIZE; count++)  
    numbers[count] = 99;
```

The loop ends when the variable count reaches 5, which is the first invalid subscript value.

The variable count is incremented after each iteration.



Default Initialization

- Global array → all elements initialized to 0 by default
- Local array → all elements *uninitialized* by default



Array Initialization

- Arrays can be initialized with an initialization list:

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

- The values are stored in the array in the order in which they appear in the list.
- The initialization list cannot exceed the array size.



Code From Program 7-3

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

Program Output

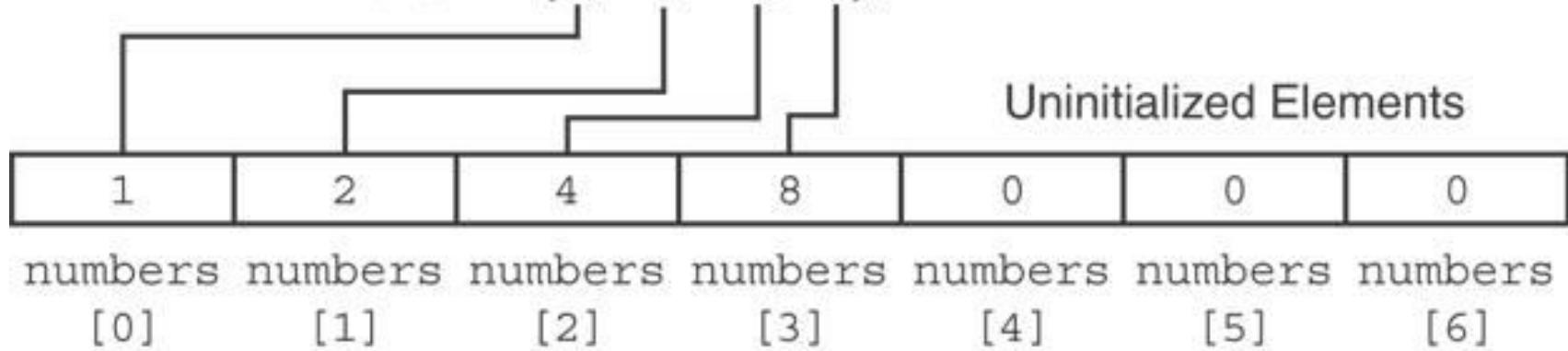
```
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```



Partial Array Initialization

- If array is initialized with fewer initial values than the size declarator, the remaining elements will be set to 0 :

```
int numbers[7] = {1, 2, 4, 8};
```



Implicit Array Sizing

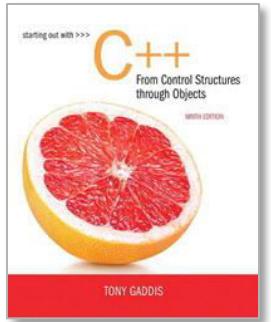
- Orange icon: Can determine array size by the size of the initialization list:

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

12	17	15	11
----	----	----	----

- Orange icon: Must use either array size declarator or initialization list at array definition





7.3

No Bounds Checking in C++



No Bounds Checking in C++

- When you use a value as an array subscript, C++ does not check it to make sure it is a *valid* subscript.
- In other words, you can use subscripts that are beyond the bounds of the array.



Code From Program 7-9

- The following code defines a three-element array, and then writes five values to it!

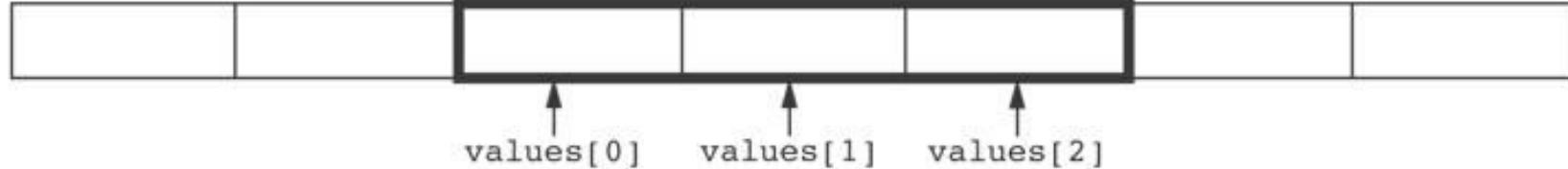
```
9     const int SIZE = 3;    // Constant for the array size
10    int values[SIZE];      // An array of 3 integers
11    int count;              // Loop counter variable
12
13    // Attempt to store five numbers in the 3-element array.
14    cout << "I will store 5 numbers in a 3-element array!\n";
15    for (count = 0; count < 5; count++)
16        values[count] = 100;
```



What the Code Does

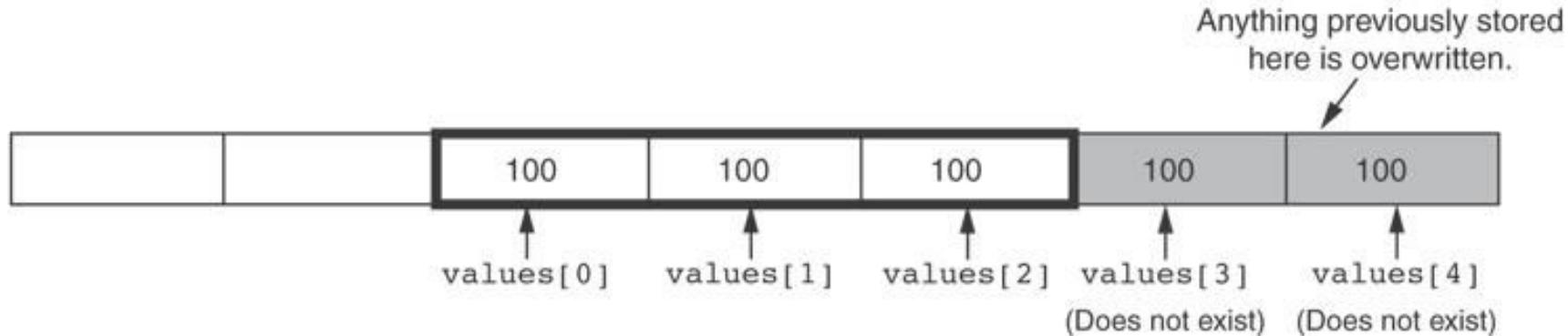
The way the `values` array is set up in memory.
The outlined area represents the array.

Memory outside the array
(Each block = 4 bytes)



Memory outside the array
(Each block = 4 bytes)

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



No Bounds Checking in C++

- Be careful not to use invalid subscripts.
- Doing so can corrupt other memory locations, crash program, or lock up computer, and cause elusive bugs.

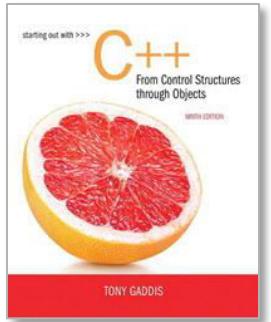


Off-By-One Errors

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

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





7.4

The Range-Based for Loop



The Range-Based for Loop

- C++ 11 provides a specialized version of the `for` loop that, in many circumstances, simplifies array processing.
- *The range-based for loop is a loop that iterates once for each element in an array.*
- *Each time the loop iterates, it copies an element from the array to a built-in variable, known as the range variable.*
- The range-based `for` loop automatically knows the number of elements in an array.
 - You do not have to use a counter variable.
 - You do not have to worry about stepping outside the bounds of the array.



The Range-Based for Loop

- Here is the general format of the range-based for loop:

```
for (dataType rangeVariable : array)
    statement;
```

- dataType* is the data type of the range variable.
- rangeVariable* is the name of the range variable. This variable will receive the value of a different array element during each loop iteration.
- array* is the name of an array on which you wish the loop to operate.
- statement* is a statement that executes during a loop iteration. If you need to execute more than one statement in the loop, enclose the statements in a set of braces.



The range-based for loop in Program 7-10

```
1 // This program demonstrates the range-based for loop.  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main()  
6 {  
7     // Define an array of integers.  
8     int numbers[] = { 10, 20, 30, 40, 50 };  
9  
10    // Display the values in the array.  
11    for (int val : numbers)  
12        cout << val << endl;  
13  
14    return 0;  
15 }
```



Modifying an Array with a Range-Based for Loop

- As the range-based `for` loop executes, its range variable contains only a copy of an array element.
- You cannot use a range-based `for` loop to modify the contents of an array unless you declare the range variable as a reference.
- To declare the range variable as a reference variable, simply write an ampersand (`&`) in front of its name in the loop header.
- Program 7-12 demonstrates



Modifying an Array with a Range-Based for Loop in Program 7-12

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

// Get values for the array.
for (int &val : numbers)
{
    cout << "Enter an integer value: ";
    cin >> val;
}

// Display the values in the array.
cout << "Here are the values you entered:\n";
for (int val : numbers)
    cout << val << endl;
```



Modifying an Array with a Range-Based for Loop

You can use the `auto` key word with a reference range variable. For example, the code in lines 12 through 16 in Program 7-12 could have been written like this:

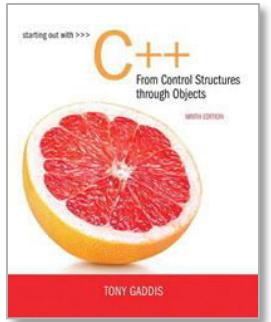
```
for (auto &val : numbers)
{
    cout << "Enter an integer value: ";
    cin >> val;
}
```



The Range-Based for Loop versus the Regular for Loop

- The range-based for loop can be used in any situation where you need to step through the elements of an array, and you do not need to use the element subscripts.
- If you need the element subscript for some purpose, use the regular for loop.





7.5

Processing Array Contents



Processing Array Contents

- Orange Array elements can be treated as ordinary variables of the same type as the array
- Orange When using `++`, `--` operators, don't confuse the element with the subscript:

```
tests[i]++; // add 1 to tests[i]
tests[i++]; // increment i, no
            // effect on tests
```



Array Assignment

To copy one array to another,

- Don't try to assign one array to the other:

```
newTests = tests; // Won't work
```

- Instead, assign element-by-element:

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



Printing the Contents of an Array

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

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

But, this ONLY works with character arrays!



Printing the Contents of an Array

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

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



Printing the Contents of an Array

- In C++ 11 you can use the range-based `for` loop to display an array's contents, as shown here:

```
for (int val : numbers)  
    cout << val << endl;
```



Summing and Averaging Array Elements

- Use a simple loop to add together array elements:

```
int tnum;  
double average, sum = 0;  
for(tnum = 0; tnum < SIZE; tnum++)  
    sum += tests[tnum];
```

- Once summed, can compute average:

```
average = sum / SIZE;
```



Summing and Averaging Array Elements

- In C++ 11 you can use the range-based for loop, as shown here:

```
double total = 0;    // Initialize accumulator
double average;      // Will hold the average
for (int val : scores)
    total += val;
average = total / NUM_SCORES;
```



Finding the Highest Value in an Array

```
int count;
int highest;
highest = numbers[0];
for (count = 1; count < SIZE; count++)
{
    if (numbers[count] > highest)
        highest = numbers[count];
}
```

When this code is finished, the `highest` variable will contain the highest value in the `numbers` array.



Finding the Lowest Value in an Array

```
int count;
int lowest;
lowest = numbers[0];
for (count = 1; count < SIZE; count++)
{
    if (numbers[count] < lowest)
        lowest = numbers[count];
}
```

When this code is finished, the `lowest` variable will contain the lowest value in the `numbers` array.



Partially-Filled Arrays

- If it is unknown how much data an array will be holding:
 - Make the array large enough to hold the largest expected number of elements.
 - Use a counter variable to keep track of the number of items stored in the array.

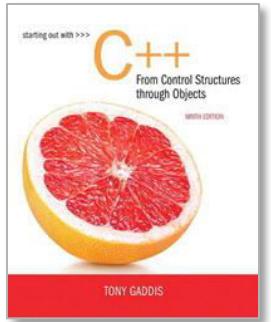


Comparing Arrays

- To compare two arrays, you must compare element-by-element:

```
const int SIZE = 5;
int firstArray[SIZE] = { 5, 10, 15, 20, 25 };
int secondArray[SIZE] = { 5, 10, 15, 20, 25 };
bool arraysEqual = true; // Flag variable
int count = 0;           // Loop counter variable
// Compare the two arrays.
while (arraysEqual && count < SIZE)
{
    if (firstArray[count] != secondArray[count])
        arraysEqual = false;
    count++;
}
if (arraysEqual)
    cout << "The arrays are equal.\n";
else
    cout << "The arrays are not equal.\n";
```





7.6

Using Parallel Arrays



Using Parallel Arrays

- Parallel arrays: two or more arrays that contain related data
- A subscript is used to relate arrays: elements at same subscript are related
- Arrays may be of different types



Parallel Array Example

```
const int SIZE = 5;      // Array size
int id[SIZE];           // student ID
double average[SIZE];   // course average
char grade[SIZE];       // course grade

...
for(int i = 0; i < SIZE; i++)
{
    cout << "Student ID: " << id[i]
        << " average: " << average[i]
        << " grade: " << grade[i]
        << endl;
}
```



Parallel Arrays in Program 7-15

Program 7-15

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

(Program Continues)



Parallel Arrays in Program 7-15

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

Program Output with Example Input Shown in Bold

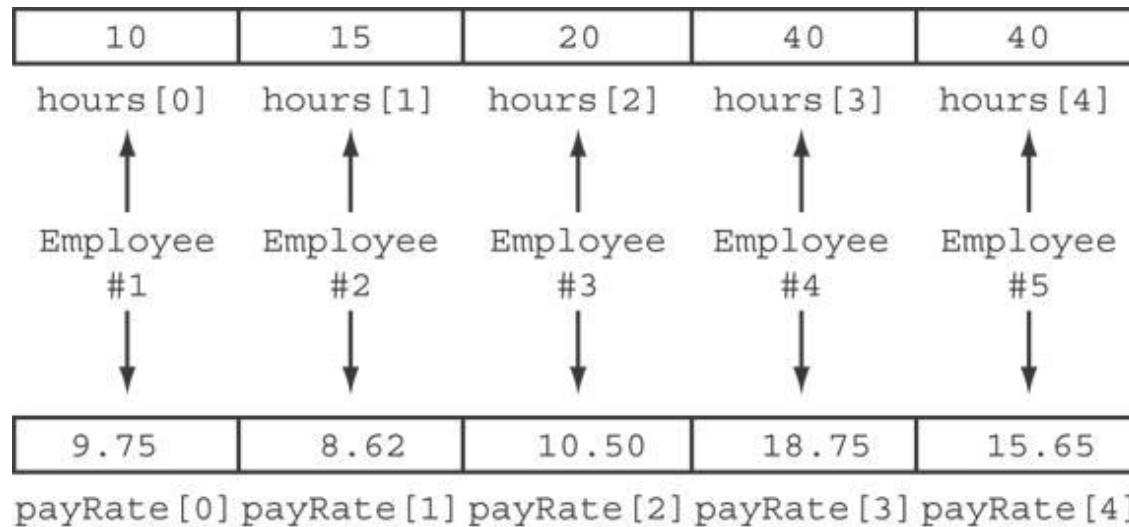
Enter the hours worked by 5 employees and their hourly pay rates.

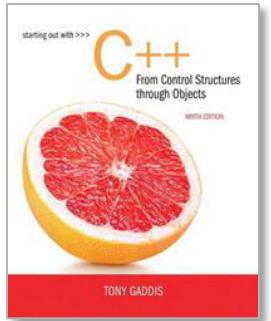
```
Hours worked by employee #1: 10 [Enter]  
Hourly pay rate for employee #1: 9.75 [Enter]  
Hours worked by employee #2: 15 [Enter]  
Hourly pay rate for employee #2: 8.62 [Enter]  
Hours worked by employee #3: 20 [Enter]  
Hourly pay rate for employee #3: 10.50 [Enter]  
Hours worked by employee #4: 40 [Enter]  
Hourly pay rate for employee #4: 18.75 [Enter]  
Hours worked by employee #5: 40 [Enter]  
Hourly pay rate for employee #5: 15.65 [Enter]  
Here is the gross pay for each employee:  
Employee #1: $97.50  
Employee #2: $129.30  
Employee #3: $210.00  
Employee #4: $750.00  
Employee #5: $626.00
```



Parallel Arrays in Program 7-15

The hours and payRate arrays are related through their subscripts:





7.7

Arrays as Function Arguments



Arrays as Function Arguments

- To pass an array to a function, just use the array name:

```
showScores(tests);
```

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

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



Arrays as Function Arguments

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

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

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

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

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



Passing an Array to a Function in Program 7-17

Program 7-17

```
1 // This program demonstrates an array being passed to a function.
2 #include <iostream>
3 using namespace std;
4
5 void showValues(int [], int); // Function prototype
6
7 int main()
8 {
9     const int ARRAY_SIZE = 8;
10    int numbers[ARRAY_SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
11
12    showValues(numbers, ARRAY_SIZE);
13    return 0;
14 }
15
16 //*****
17 // Definition of function showValue. *
18 // This function accepts an array of integers and * 
19 // the array's size as its arguments. The contents * 
20 // of the array are displayed. *
21 //*****
22
23 void showValues(int nums[], int size)
24 {
25     for (int index = 0; index < size; index++)
26         cout << nums[index] << " ";
27     cout << endl;
28 }
```

Program Output

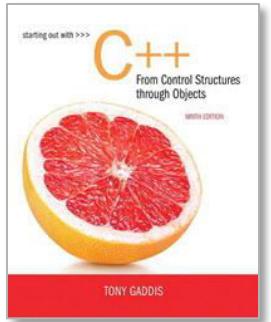
5 10 15 20 25 30 35 40



Modifying Arrays in Functions

- Array names in functions are like reference variables – changes made to array in a function are reflected in actual array in calling function
- Need to exercise caution that array is not inadvertently changed by a function





7.8

Two-Dimensional Arrays



Two-Dimensional Arrays

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

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

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



Two-Dimensional Array Representation

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

columns			
rows	exams [0] [0]	exams [0] [1]	exams [0] [2]
	exams [1] [0]	exams [1] [1]	exams [1] [2]
	exams [2] [0]	exams [2] [1]	exams [2] [2]
	exams [3] [0]	exams [3] [1]	exams [3] [2]

- Use two subscripts to access element:

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



A Two-dimensional Array in Program 7-21

Program 7-21

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

(program continues)



A Two-dimensional Array in Program 7-21

Program 7-21 *(continued)*

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



A Two-dimensional Array in Program 7-21

Program Output with Example Input Shown in Bold

This program will calculate the total sales of all the company's divisions.

Enter the following sales data:

Division 1, Quarter 1: **\$31569.45 [Enter]**

Division 1, Quarter 2: **\$29654.23 [Enter]**

Division 1, Quarter 3: **\$32982.54 [Enter]**

Division 1, Quarter 4: **\$39651.21 [Enter]**

Division 2, Quarter 1: **\$56321.02 [Enter]**

Division 2, Quarter 2: **\$54128.63 [Enter]**

Division 2, Quarter 3: **\$41235.85 [Enter]**

Division 2, Quarter 4: **\$54652.33 [Enter]**

Division 3, Quarter 1: **\$29654.35 [Enter]**

Division 3, Quarter 2: **\$28963.32 [Enter]**

Division 3, Quarter 3: **\$25353.55 [Enter]**

Division 3, Quarter 4: **\$32615.88 [Enter]**

The total sales for the company are: \$456782.34



2D Array Initialization

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

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

84	78
92	97

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



Two-Dimensional Array as Parameter, Argument

- Use array name as argument in function call:

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

- Use empty [] for row, size declarator for column in prototype, header:

```
const int COLS = 2;
```

```
// Prototype
```

```
void getExams(int [] [COLS], int);
```

```
// Header
```

```
void getExams(int exams[] [COLS], int rows)
```



Example – The showArray Function from Program 7-22

```
30 //*****  
31 // Function Definition for showArray *  
32 // The first argument is a two-dimensional int array with COLS *  
33 // columns. The second argument, rows, specifies the number of *  
34 // rows in the array. The function displays the array's contents. *  
35 //*****  
36  
37 void showArray(int array[][COLS], int rows)  
38 {  
39     for (int x = 0; x < rows; x++)  
40     {  
41         for (int y = 0; y < COLS; y++)  
42         {  
43             cout << setw(4) << array[x][y] << " ";  
44         }  
45         cout << endl;  
46     }  
47 }
```



How showArray is Called

```
15     int table1[TBL1_ROWS][COLS] = {{1, 2, 3, 4},  
16                               {5, 6, 7, 8},  
17                               {9, 10, 11, 12}};  
18     int table2[TBL2_ROWS][COLS] = {{10, 20, 30, 40},  
19                               {50, 60, 70, 80},  
20                               {90, 100, 110, 120},  
21                               {130, 140, 150, 160}};  
22  
23     cout << "The contents of table1 are:\n";  
24     showArray(table1, TBL1_ROWS);  
25     cout << "The contents of table2 are:\n";  
26     showArray(table2, TBL2_ROWS);
```



Summing All the Elements in a Two-Dimensional Array

- Given the following definitions:

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



Summing All the Elements in a Two-Dimensional Array

```
// Sum the array elements.  
for (int row = 0; row < NUM_ROWS; row++)  
{  
    for (int col = 0; col < NUM_COLS; col++)  
        total += numbers[row][col];  
  
    // Display the sum.  
    cout << "The total is " << total << endl;
```



Summing the Rows of a Two-Dimensional Array

- Given the following definitions:

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



Summing the Rows of a Two-Dimensional Array

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



Summing the Columns of a Two-Dimensional Array

- Given the following definitions:

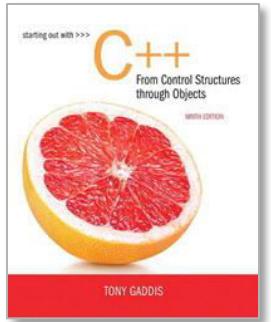
```
const int NUM_STUDENTS = 3;
const int NUM_SCORES = 5;
double total;    // Accumulator
double average; // To hold average scores
double scores[NUM_STUDENTS][NUM_SCORES] =
    {{88, 97, 79, 86, 94},
     {86, 91, 78, 79, 84},
     {82, 73, 77, 82, 89}};
```



Summing the Columns of a Two-Dimensional Array

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





7.9

Arrays with Three or More Dimensions



Arrays with Three or More Dimensions

- Orange icon: Can define arrays with any number of dimensions:

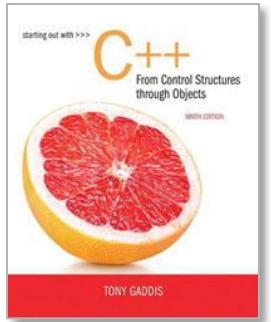
```
short rectSolid[2][3][5];
```

```
double timeGrid[3][4][3][4];
```

- Orange icon: When used as parameter, specify all but 1st dimension in prototype, heading:

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





7.11

Introduction to the STL vector



Introduction to the STL vector

- A data type defined in the Standard Template Library (covered more in Chapter 17)
- Can hold values of any type:

```
vector<int> scores;
```
- Automatically adds space as more is needed – no need to determine size at definition
- Can use [] to access elements



Declaring Vectors

- You must #include<vector>

- Declare a vector to hold int element:

```
vector<int> scores;
```

- Declare a vector with initial size 30:

```
vector<int> scores(30);
```

- Declare a vector and initialize all elements to 0:

```
vector<int> scores(30, 0);
```

- Declare a vector initialized to size and contents of another vector:

```
vector<int> finals(scores);
```



Adding Elements to a Vector

- If you are using C++ 11, you can initialize a vector with a list of values:

```
vector<int> numbers { 10, 20, 30, 40 };
```

- Use `push_back` member function to add element to a full array or to an array that had no defined size:

```
scores.push_back(75);
```

- Use `size` member function to determine size of a vector:

```
howbig = scores.size();
```



Removing Vector Elements

- Orange Use `pop_back` member function to remove last element from vector:

```
scores.pop_back();
```

- Orange To remove all contents of vector, use `clear` member function:

```
scores.clear();
```

- Orange To determine if vector is empty, use `empty` member function:

```
while (!scores.empty()) ...
```



Using the Range-Based for Loop with a vector

Program 7-25

```
1 // This program demonstrates the range-based for loop with a vector.  
2 include <iostream>  
3 #include <vector>  
4 using namespace std;  
5  
6 int main()  
7 {  
8     // Define and initialize a vector.  
9     vector<int> numbers { 10, 20, 30, 40, 50 };  
10  
11    // Display the vector elements.  
12    for (int val : numbers)  
13        cout << val << endl;  
14  
15    return 0;  
16 }
```

Program Output

```
10  
20  
30  
40  
50
```



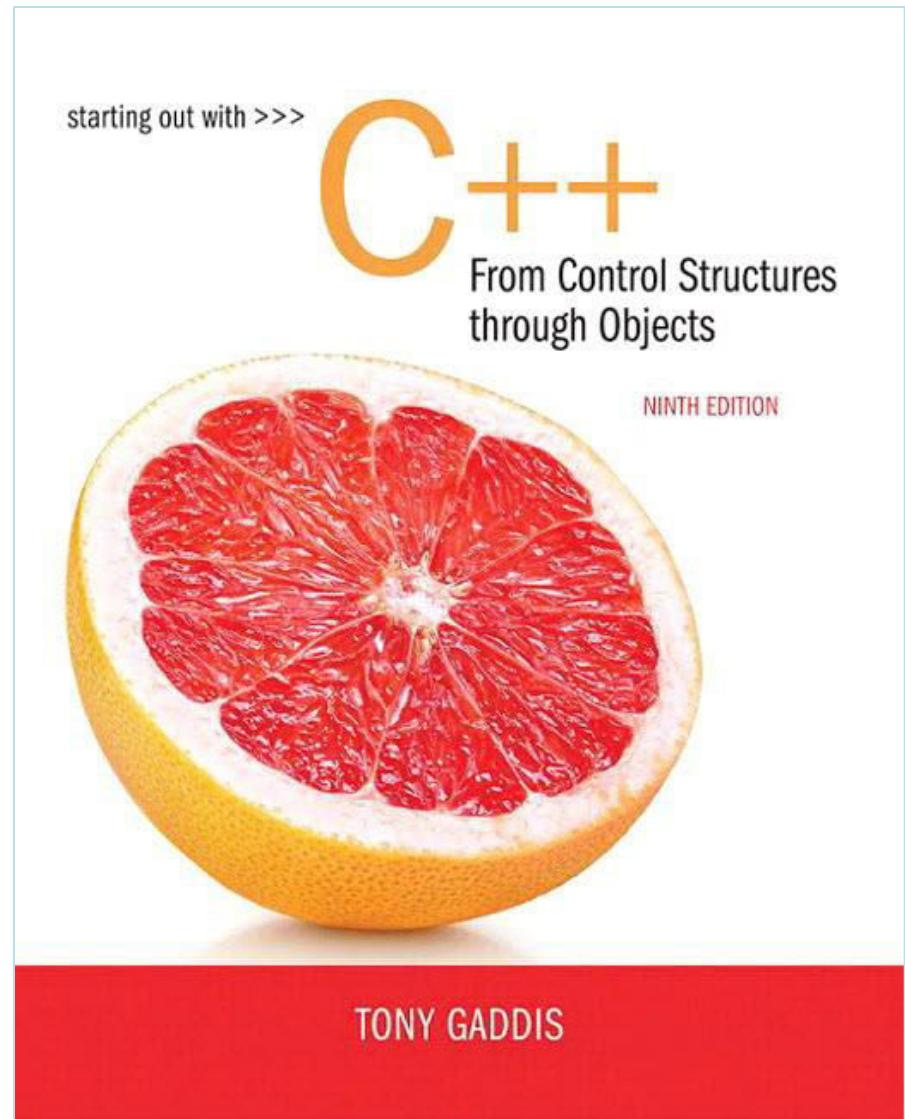
Other Useful Member Functions

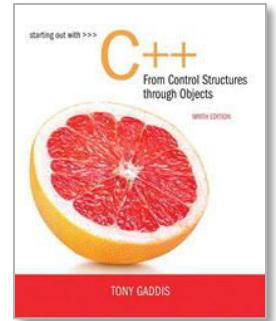
Member Function	Description	Example
<code>at (i)</code>	Returns the value of the element at position <code>i</code> in the vector	<code>cout << vec1.at(i);</code>
<code>capacity()</code>	Returns the maximum number of elements a vector can store without allocating more memory	<code>maxElements = vec1.capacity();</code>
<code>reverse()</code>	Reverse the order of the elements in a vector	<code>vec1.reverse();</code>
<code>resize (n, val)</code>	Resizes the vector so it contains <code>n</code> elements. If new elements are added, they are initialized to <code>val</code> .	<code>vec1.resize(5, 0);</code>
<code>swap (vec2)</code>	Exchange the contents of two vectors	<code>vec1.swap(vec2);</code>



Chapter 8:

Searching and Sorting Arrays





8.1

Introduction to Search Algorithms



Introduction to Search Algorithms

- Search: locate an item in a list of information
- Two algorithms we will examine:
 - Linear search
 - Binary search



Linear Search

- Orange Also called the sequential search
- Orange Starting at the first element, this algorithm sequentially steps through an array examining each element until it locates the value it is searching for.



Linear Search - Example

- Array numlist contains:

17	23	5	11	2	29	3
----	----	---	----	---	----	---

- Searching for the value 11, linear search examines 17, 23, 5, and 11
- Searching for the value 7, linear search examines 17, 23, 5, 11, 2, 29, and 3



Linear Search

Algorithm:

```
set found to false; set position to -1; set index to 0
while index < number of elts. and found is false
    if list[index] is equal to search value
        found = true
        position = index
    end if
    add 1 to index
end while
return position
```



A Linear Search Function

```
int linearSearch(int arr[], int size, int value)
{
    int index = 0;          // Used as a subscript to search the array
    int position = -1;     // To record the position of search value
    bool found = false;    // Flag to indicate if value was found

    while (index < size && !found)
    {
        if (arr[index] == value) // If the value is found
        {
            found = true; // Set the flag
            position = index; // Record the value's subscript
        }
        index++; // Go to the next element
    }
    return position; // Return the position, or -1
}
```



Linear Search - Tradeoffs

● Benefits:

- Easy algorithm to understand
- Array can be in any order

● Disadvantages:

- Inefficient (slow): for array of N elements, examines $N/2$ elements on average for value in array, N elements for value not in array



Binary Search

Requires array elements to be in order

1. Divides the array into three sections:
 - ➊ middle element
 - ➋ elements on one side of the middle element
 - ➌ elements on the other side of the middle element
2. If the middle element is the correct value, done.
Otherwise, go to step 1. using only the half of the array that may contain the correct value.
3. Continue steps 1. and 2. until either the value is found or there are no more elements to examine



Binary Search - Example

- Array numlist2 contains:

2	3	5	11	17	23	29
---	---	---	----	----	----	----

- Searching for the value 11, binary search examines 11 and stops
- Searching for the value 7, linear search examines 11, 3, 5, and stops



Binary Search

Set first to 0

Set last to the last subscript in the array

Set found to false

Set position to -1

While found is not true and first is less than or equal to last

Set middle to the subscript half-way between array[first] and array[last].

If array[middle] equals the desired value

Set found to true

Set position to middle

Else If array[middle] is greater than the desired value

Set last to middle - 1

Else

Set first to middle + 1

End If.

End While.

Return position.



A Binary Search Function

```
int binarySearch(int array[], int size, int value)
{
    int first = 0,                      // First array element
        last = size - 1,                // Last array element
        middle,                         // Mid point of search
        position = -1;                  // Position of search value
    bool found = false;                 // Flag

    while (!found && first <= last)
    {
        middle = (first + last) / 2;    // Calculate mid point
        if (array[middle] == value)      // If value is found at mid
        {
            found = true;
            position = middle;
        }
        else if (array[middle] > value)  // If value is in lower half
            last = middle - 1;
        else
            first = middle + 1;         // If value is in upper half
    }
    return position;
}
```



Binary Search - Tradeoffs

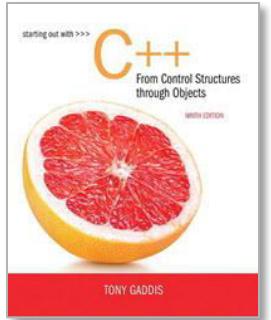
Benefits:

- Much more efficient than linear search. For array of N elements, performs at most $\log_2 N$ comparisons

Disadvantages:

- Requires that array elements be sorted





8.3

Introduction to Sorting Algorithms



Introduction to Sorting Algorithms

- Sort: arrange values into an order:
 - Alphabetical
 - Ascending numeric
 - Descending numeric

- Two algorithms considered here:
 - Bubble sort
 - Selection sort



Bubble Sort

Concept:

- Compare 1st two elements
 - If out of order, exchange them to put in order
- Move down one element, compare 2nd and 3rd elements, exchange if necessary. Continue until end of array.
- Pass through array again, exchanging as necessary
- Repeat until pass made with no exchanges



Example – First Pass

Array numlist3 contains:

17	23	5	11
----	----	---	----

compare values
17 and 23 – in correct
order, so no exchange

compare values 23 and
5 – not in correct order,
so exchange them

compare values 23 and
11 – not in correct order,
so exchange them



Example – Second Pass

After first pass, array numlist3 contains:

17	5	11	23
----	---	----	----

compare values 17 and 5 – not in correct order,
so exchange them

compare values 17 and
11 – not in correct order,
so exchange them

compare values 17 and
23 – in correct order,
so no exchange



Example – Third Pass

After second pass, array numlist3 contains:

5	11	17	23
---	----	----	----

compare values 5 and
11 – in correct order,
so no exchange

compare values 11 and
17 – in correct order,
so no exchange

compare values 17 and
23 – in correct order,
so no exchange

No exchanges, so
array is in order



A Bubble Sort Function – From Program 8-4

```
37 void bubbleSort(int array[], int size)
38 {
39     int maxElement;
40     int index;
41
42     for (maxElement = size - 1; maxElement > 0; maxElement--)
43     {
44         for (index = 0; index < maxElement; index++)
45         {
46             if (array[index] > array[index + 1])
47             {
48                 swap(array[index], array[index + 1]);
49             }
50         }
51     }
52 }
53
54 //*****
55 // The swap function swaps a and b in memory.      *
56 //*****
57 void swap(int &a, int &b)
58 {
59     int temp = a;
60     a = b;
61     b = temp;
62 }
```



Bubble Sort - Tradeoffs

- Benefit:

- Easy to understand and implement

- Disadvantage:

- Inefficient: slow for large arrays



Selection Sort

- Concept for sort in ascending order:
 - Locate smallest element in array. Exchange it with element in position 0
 - Locate next smallest element in array. Exchange it with element in position 1.
 - Continue until all elements are arranged in order



Selection Sort - Example

Array `numlist` contains:

11	2	29	3
----	---	----	---

1. Smallest element is 2. Exchange 2 with element in 1st position in array:

2	11	29	3
---	----	----	---



Example (Continued)

2. Next smallest element is 3. Exchange 3 with element in 2nd position in array:

2	3	29	11
---	---	----	----

3. Next smallest element is 11. Exchange 11 with element in 3rd position in array:

2	3	11	29
---	---	----	----



A Selection Sort Function – From Program 8-5

```
37 void selectionSort(int array[], int size)
38 {
39     int minIndex, minValue;
40
41     for (int start = 0; start < (size - 1); start++)
42     {
43         minIndex = start;
44         minValue = array[start];
45         for (int index = start + 1; index < size; index++)
46         {
47             if (array[index] < minValue)
48             {
49                 minValue = array[index];
50                 minIndex = index;
51             }
52         }
53         swap(array[minIndex], array[start]);
54     }
55 }
```



Selection Sort - Tradeoffs

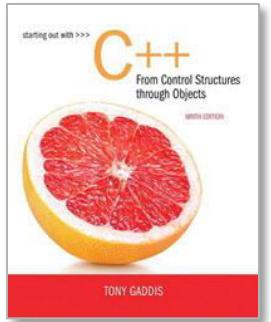
○ Benefit:

- More efficient than Bubble Sort, since fewer exchanges

○ Disadvantage:

- May not be as easy as Bubble Sort to understand





8.5

Sorting and Searching Vectors



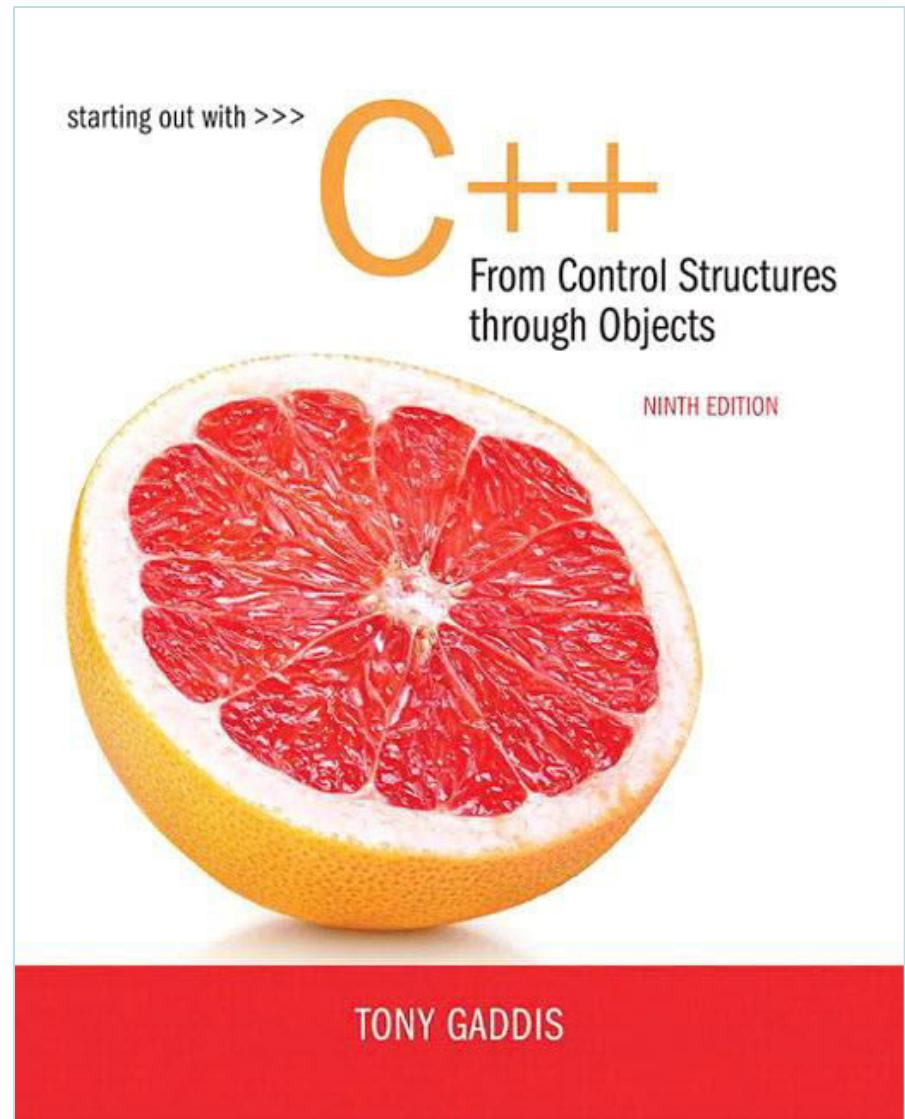
Sorting and Searching Vectors

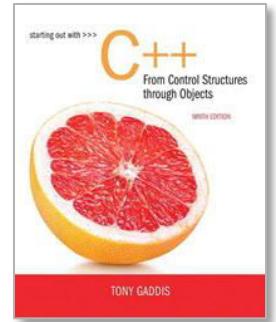
- Sorting and searching algorithms can be applied to vectors as well as arrays
- Need slight modifications to functions to use vector arguments:
 - `vector <type> &` used in prototype
 - No need to indicate vector size – functions can use `size` member function to calculate



Chapter 9:

Pointers





9.1

Getting the Address of a Variable

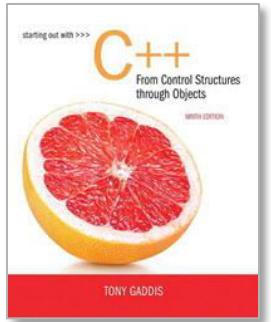


Getting the Address of a Variable

- Each variable in program is stored at a unique address
- Use address operator & to get address of a variable:

```
int num = -99;  
cout << &num; // prints address  
                  // in hexadecimal
```





9.2

Pointer Variables



Pointer Variables

- Pointer variable : Often just called a pointer, it's a variable that holds an address
- Because a pointer variable holds the address of another piece of data, it "points" to the data



Something Like Pointers: Arrays

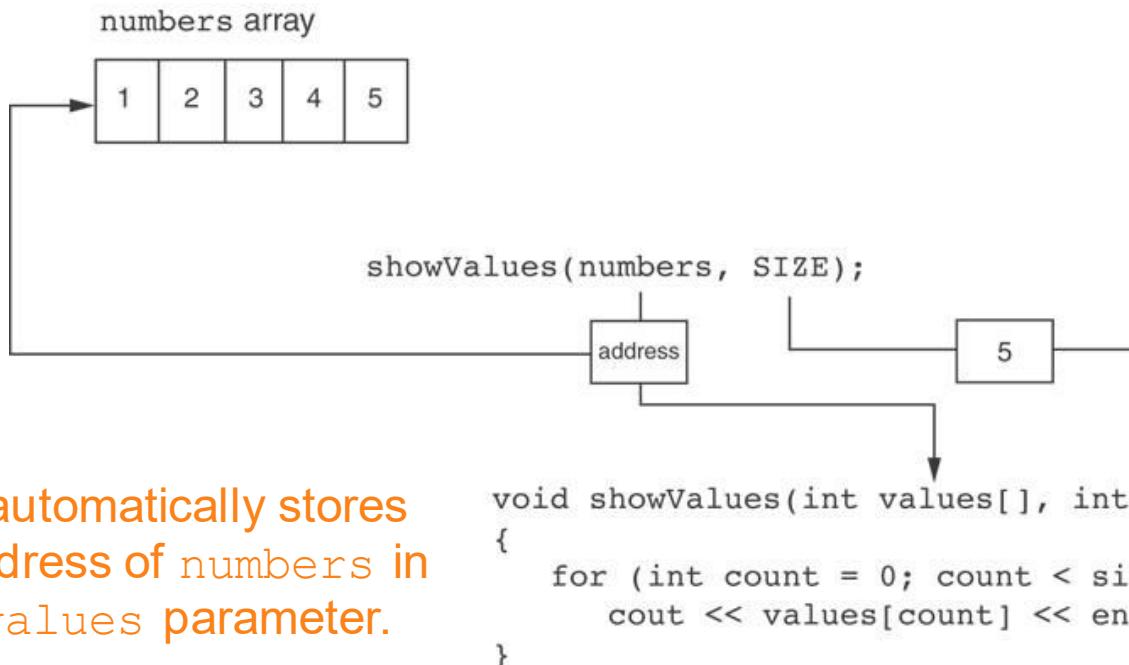
- We have already worked with something similar to pointers, when we learned to pass arrays as arguments to functions.
- For example, suppose we use this statement to pass the array numbers to the showValues function:

```
showValues (numbers, SIZE) ;
```



Something Like Pointers : Arrays

The `values` parameter, in the `showValues` function, points to the `numbers` array.



C++ automatically stores the address of `numbers` in the `values` parameter.



Something Like Pointers: Reference Variables

- We have also worked with something like pointers when we learned to use reference variables.
Suppose we have this function:

```
void getOrder(int &donuts)
{
    cout << "How many doughnuts do you want? ";
    cin >> donuts;
}
```

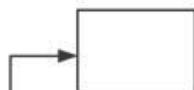
- And we call it with this code:

```
int jellyDonuts;
getOrder(jellyDonuts);
```



Something Like Pointers: Reference Variables

jellyDonuts variable



getOrder(jellyDonuts);

address

```
void getOrder(int &donuts)
{
    cout << "How many doughnuts do you want? ";
    cin >> donuts;
}
```

C++ automatically stores
the address of
jellyDonuts in the
donuts parameter.

The donuts parameter, in the getOrder function,
points to the jellyDonuts variable.



Pointer Variables

- Pointer variables are yet another way using a memory address to work with a piece of data.
- Pointers are more "low-level" than arrays and reference variables.
- This means you are responsible for finding the address you want to store in the pointer and correctly using it.



Pointer Variables

Orange Definition:

```
int *intptr;
```

Orange Read as:

“intptr can hold the address of an int”

Orange Spacing in definition does not matter:

```
int * intptr; // same as above  
int* intptr; // same as above
```

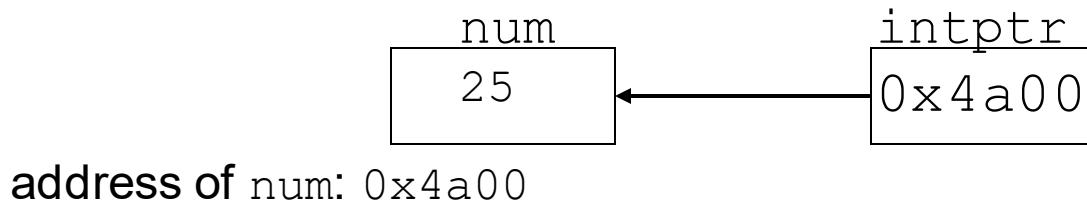


Pointer Variables

- Assigning an address to a pointer variable:

```
int *intptr;  
intptr = &num;
```

- Memory layout:



Pointer Variables

- Orange Initialize pointer variables with the special value `nullptr`.
- Orange In C++ 11, the `nullptr` key word was introduced to represent the address 0.
- Orange Here is an example of how you define a pointer variable and initialize it with the value `nullptr`:

```
int *ptr = nullptr;
```



A Pointer Variable in Program 9-2

Program 9-2

```
1 // This program stores the address of a variable in a pointer.  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main()  
6 {  
7     int x = 25;          // int variable  
8     int *ptr = nullptr;  // Pointer variable, can point to an int  
9  
10    ptr = &x;           // Store the address of x in ptr  
11    cout << "The value in x is " << x << endl;  
12    cout << "The address of x is " << ptr << endl;  
13    return 0;  
14 }
```

Program Output

```
The value in x is 25  
The address of x is 0x7e00
```



The Indirection Operator

- The indirection operator (*) dereferences a pointer.
- It allows you to access the item that the pointer points to.

```
int x = 25;  
int *intptr = &x;  
cout << *intptr << endl;
```



This prints 25.



The Indirection Operator in Program 9-3

Program 9-3

```
1 // This program demonstrates the use of the indirection operator.  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main()  
6 {  
7     int x = 25;           // int variable  
8     int *ptr = nullptr;   // Pointer variable, can point to an int  
9  
10    ptr = &x;            // Store the address of x in ptr  
11  
12    // Use both x and ptr to display the value in x.  
13    cout << "Here is the value in x, printed twice:\n";  
14    cout << x << endl;    // Displays the contents of x  
15    cout << *ptr << endl; // Displays the contents of x  
16  
17    // Assign 100 to the location pointed to by ptr. This  
18    // will actually assign 100 to x.  
19    *ptr = 100;
```

(program continues)



The Indirection Operator in Program 9-3

Program 9-3

(continued)

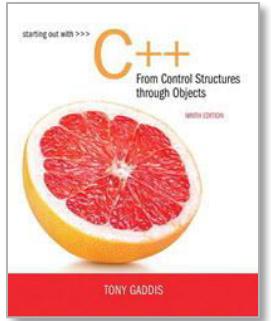
```
20
21      // Use both x and ptr to display the value in x.
22      cout << "Once again, here is the value in x:\n";
23      cout << x << endl;      // Displays the contents of x
24      cout << *ptr << endl; // Displays the contents of x
25      return 0;
26 }
```

Program Output

Here is the value in x, printed twice:

```
25
25
Once again, here is the value in x:
100
100
```





9.3

The Relationship Between Arrays and Pointers



The Relationship Between Arrays and Pointers

- Array name is starting address of array

```
int vals[] = {4, 7, 11};
```



starting address of `vals`: 0x4a00

```
cout << vals;           // displays  
                      // 0x4a00  
cout << vals[0];      // displays 4
```

The Relationship Between Arrays and Pointers

- Array name can be used as a pointer constant:

```
int vals[] = {4, 7, 11};  
cout << *vals; // displays 4
```

- Pointer can be used as an array name:

```
int *valptr = vals;  
cout << valptr[1]; // displays 7
```



The Array Name Being Dereferenced in Program 9-5

Program 9-5

```
1 // This program shows an array name being dereferenced with the *
2 // operator.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     short numbers[] = {10, 20, 30, 40, 50};
9
10    cout << "The first element of the array is ";
11    cout << *numbers << endl;
12    return 0;
13 }
```

Program Output

The first element of the array is 10



Pointers in Expressions

Given:

```
int vals[] = {4, 7, 11}, *valptr;  
valptr = vals;
```

What is valptr + 1? It means (address in
valptr) + (1 * size of an int)

```
cout << * (valptr+1); //displays 7  
cout << * (valptr+2); //displays 11
```

Must use () as shown in the expressions



Array Access

- Array elements can be accessed in many ways:

Array access method	Example
array name and []	<code>vals[2] = 17;</code>
pointer to array and []	<code>valptr[2] = 17;</code>
array name and subscript arithmetic	<code>* (vals + 2) = 17;</code>
pointer to array and subscript arithmetic	<code>* (valptr + 2) = 17;</code>



Array Access

- Conversion: `vals[i]` is equivalent to
`* (vals + i)`
- No bounds checking performed on array access, whether using array name or a pointer



From Program 9-7

```
9     const int NUM_COINS = 5;
10    double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
11    double *doublePtr; // Pointer to a double
12    int count;         // Array index
13
14    // Assign the address of the coins array to doublePtr.
15    doublePtr = coins;
16
17    // Display the contents of the coins array. Use subscripts
18    // with the pointer!
19    cout << "Here are the values in the coins array:\n";
20    for (count = 0; count < NUM_COINS; count++)
21        cout << doublePtr[count] << " ";
22
23    // Display the contents of the array again, but this time
24    // use pointer notation with the array name!
25    cout << "\nAnd here they are again:\n";
26    for (count = 0; count < NUM_COINS; count++)
27        cout << *(coins + count) << " ";
28    cout << endl;
```

Program Output

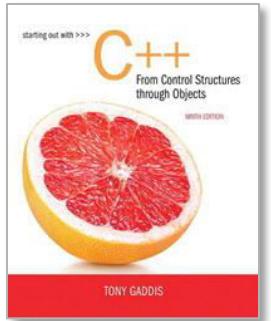
Here are the values in the coins array:

0.05 0.1 0.25 0.5 1

And here they are again:

0.05 0.1 0.25 0.5 1





9.4

Pointer Arithmetic



Pointer Arithmetic

Operations on pointer variables:

Operation	Example
	<pre>int vals[] = {4, 7, 11}; int *valptr = vals;</pre>
<code>++, --</code>	<code>valptr++; // points at 7 valptr--; // now points at 4</code>
<code>+, - (pointer and int)</code>	<code>cout << *(valptr + 2); // 11</code>
<code>+=, -= (pointer and int)</code>	<code>valptr = vals; // points at 4 valptr += 2; // points at 11</code>
<code>- (pointer from pointer)</code>	<code>cout << valptr - val; // difference // (number of ints) between valptr // and val</code>



From Program 9-9

```
7     const int SIZE = 8;
8     int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
9     int *numPtr = nullptr; // Pointer
10    int count;           // Counter variable for loops
11
12    // Make numPtr point to the set array.
13    numPtr = set;
14
15    // Use the pointer to display the array contents.
16    cout << "The numbers in set are:\n";
17    for (count = 0; count < SIZE; count++)
18    {
19        cout << *numPtr << " ";
20        numPtr++;
21    }
22
23    // Display the array contents in reverse order.
24    cout << "\nThe numbers in set backward are:\n";
25    for (count = 0; count < SIZE; count++)
26    {
27        numPtr--;
28        cout << *numPtr << " ";
29    }
30    return 0;
31 }
```

Program Output

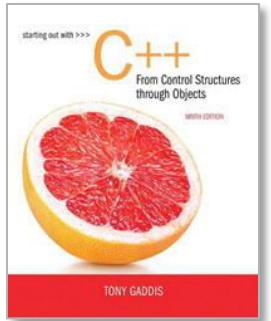
The numbers in set are:

5 10 15 20 25 30 35 40

The numbers in set backward are:

40 35 30 25 20 15 10 5





9.5

Initializing Pointers



Initializing Pointers

- Can initialize at definition time:

```
int num, *numptr = &num;  
int val[3], *valptr = val;
```

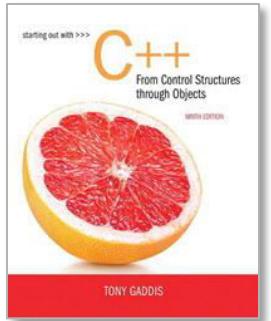
- Cannot mix data types:

```
double cost;  
int *ptr = &cost; // won't work
```

- Can test for an invalid address for ptr with:

```
if (!ptr) ...
```





9.6

Comparing Pointers

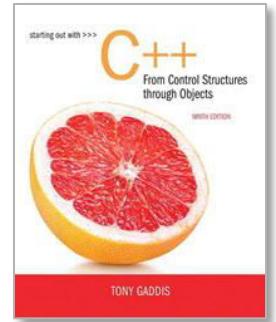


Comparing Pointers

- Relational operators (<, >=, etc.) can be used to compare addresses in pointers
- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)      // compares  
                      // addresses  
if (*ptr1 == *ptr2)  // compares  
                      // contents
```





9.7

Pointers as Function Parameters



Pointers as Function Parameters

- A pointer can be a parameter
- Works like reference variable to allow change to argument from within function
- Requires:

- 1) asterisk * on parameter in prototype and heading

```
void getNum(int *ptr); // ptr is pointer to an int
```

- 2) asterisk * in body to dereference the pointer

```
cin >> *ptr;
```

- 3) address as argument to the function

```
getNum(&num); // pass address of num to getNum
```



Example

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

```
int num1 = 2, num2 = -3;
swap(&num1, &num2);
```



Pointers as Function Parameters in Program 9-11

Program 9-11

```
1 // This program uses two functions that accept addresses of
2 // variables as arguments.
3 #include <iostream>
4 using namespace std;
5
6 // Function prototypes
7 void getNumber(int *);
8 void doubleValue(int *);
9
10 int main()
11 {
12     int number;
13
14     // Call getNumber and pass the address of number.
15     getNumber(&number);
16
17     // Call doubleValue and pass the address of number.
18     doubleValue(&number);
19
20     // Display the value in number.
21     cout << "That value doubled is " << number << endl;
22     return 0;
23 }
24
```

(Program Continues)



Pointers as Function Parameters in Program 9-11

Program 9-11 *(continued)*

```
25 //*****
26 // Definition of getNumber. The parameter, input, is a pointer. *
27 // This function asks the user for a number. The value entered   *
28 // is stored in the variable pointed to by input.                 *
29 //*****
30
31 void getNumber(int *input)
32 {
33     cout << "Enter an integer number: ";
34     cin >> *input;
35 }
36
37 //*****
38 // Definition of doubleValue. The parameter, val, is a pointer. *
39 // This function multiplies the variable pointed to by val by   *
40 // two.                                                       *
41 //*****
42
43 void doubleValue(int *val)
44 {
45     *val *= 2;
46 }
```

Program Output with Example Input Shown in Bold

Enter an integer number: **10** [Enter]

That value doubled is 20



Pointers to Constants

- If we want to store the address of a constant in a pointer, then we need to store it in a pointer-to-const.



Pointers to Constants

- Example: Suppose we have the following definitions:

```
const int SIZE = 6;  
const double payRates[SIZE] =  
    { 18.55, 17.45, 12.85,  
      14.97, 10.35, 18.89 };
```

- In this code, `payRates` is an array of constant doubles.



Pointers to Constants

- Suppose we wish to pass the `payRates` array to a function? Here's an example of how we can do it.

```
void displayPayRates(const double *rates, int size)
{
    for (int count = 0; count < size; count++)
    {
        cout << "Pay rate for employee " << (count + 1)
            << " is $" << *(rates + count) << endl;
    }
}
```

The parameter, `rates`, is a pointer to `const double`.



Declaration of a Pointer to Constant

The asterisk indicates that
rates is a pointer.

```
const double *rates
```

This is what rates points to.



Constant Pointers

● A constant pointer is a pointer that is initialized with an address, and cannot point to anything else.

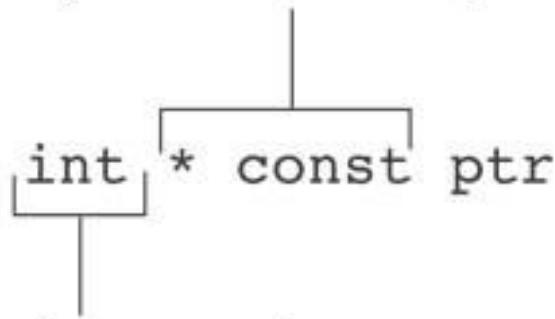
● Example

```
int value = 22;  
int * const ptr = &value;
```



Constant Pointers

* const indicates that
ptr is a constant pointer.



This is what ptr points to.



Constant Pointers to Constants

- A constant pointer to a constant is:
 - a pointer that points to a constant
 - a pointer that cannot point to anything except what it is pointing to

- Example:

```
int value = 22;  
const int * const ptr = &value;
```



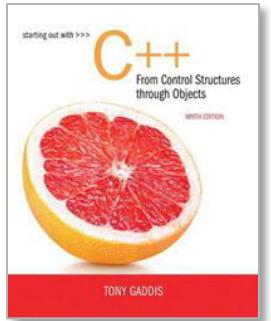
Constant Pointers to Constants

* const indicates that
ptr is a constant pointer.

const int * const ptr

This is what ptr points to.





9.8

Dynamic Memory Allocation



Dynamic Memory Allocation

- Orange icon: Can allocate storage for a variable while program is running
- Orange icon: Computer returns address of newly allocated variable
- Orange icon: Uses `new` operator to allocate memory:

```
double *dptr = nullptr;  
dptr = new double;
```

- Orange icon: `new` returns address of memory location



Dynamic Memory Allocation

- Can also use `new` to allocate array:

```
const int SIZE = 25;  
arrayPtr = new double[SIZE];
```

- Can then use `[]` or pointer arithmetic to access array:

```
for(i = 0; i < SIZE; i++)  
    *arrayptr[i] = i * i;
```

or

```
for(i = 0; i < SIZE; i++)  
    *(arrayptr + i) = i * i;
```

- Program will terminate if not enough memory available to allocate



Releasing Dynamic Memory

- Use `delete` to free dynamic memory:

```
delete fptr;
```

- Use `[]` to free dynamic array:

```
delete [] arrayptr;
```

- Only use `delete` with dynamic memory!



Dynamic Memory Allocation in Program 9-14

Program 9-14

```
1 // This program totals and averages the sales figures for any
2 // number of days. The figures are stored in a dynamically
3 // allocated array.
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;
7
8 int main()
9 {
10     double *sales = nullptr, // To dynamically allocate an array
11         total = 0.0,        // Accumulator
12         average;          // To hold average sales
13     int numDays,           // To hold the number of days of sales
14         count;            // Counter variable
15
16     // Get the number of days of sales.
17     cout << "How many days of sales figures do you wish ";
18     cout << "to process? ";
19     cin >> numDays;
```



Dynamic Memory Allocation in Program 9-14

```
20
21     // Dynamically allocate an array large enough to hold
22     // that many days of sales amounts.
23     sales = new double[numDays];
24
25     // Get the sales figures for each day.
26     cout << "Enter the sales figures below.\n";
27     for (count = 0; count < numDays; count++)
28     {
29         cout << "Day " << (count + 1) << ": ";
30         cin >> sales[count];
31     }
32
33     // Calculate the total sales
34     for (count = 0; count < numDays; count++)
35     {
36         total += sales[count];
37     }
38
39     // Calculate the average sales per day
40     average = total / numDays;
41
42     // Display the results
43     cout << fixed << showpoint << setprecision(2);
44     cout << "\n\nTotal Sales: $" << total << endl;
45     cout << "Average Sales: $" << average << endl;
```

Program 9-14 (Continued)



Dynamic Memory Allocation in Program 9-14

Program 9-14 (Continued)

```
46
47     // Free dynamically allocated memory
48     delete [] sales;
49     sales = nullptr;      // Make sales a null pointer.
50
51     return 0;
52 }
```

Program Output with Example Input Shown in Bold

How many days of sales figures do you wish to process? **5 [Enter]**
Enter the sales figures below.

Day 1: **898.63 [Enter]**

Day 2: **652.32 [Enter]**

Day 3: **741.85 [Enter]**

Day 4: **852.96 [Enter]**

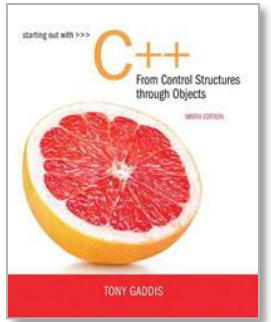
Day 5: **921.37 [Enter]**

Total Sales: \$4067.13

Average Sales: \$813.43

Notice that in line 49 `nullptr` is assigned to the `sales` pointer. The `delete` operator is designed to have no effect when used on a null pointer.





9.9

Returning Pointers from Functions



Returning Pointers from Functions

- Orange Pointer can be the return type of a function:

```
int* newNum();
```

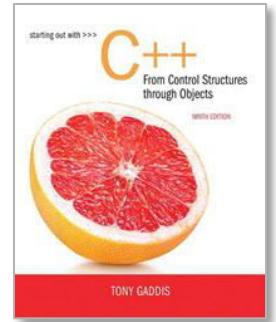
- Orange The function must not return a pointer to a local variable in the function.
- Orange A function should only return a pointer:
 - Orange to data that was passed to the function as an argument, or
 - Orange to dynamically allocated memory



From Program 9-15

```
34 int *getRandomNumbers(int num)
35 {
36     int *arr = nullptr; // Array to hold the numbers
37
38     // Return a null pointer if num is zero or negative.
39     if (num <= 0)
40         return nullptr;
41
42     // Dynamically allocate the array.
43     arr = new int[num];
44
45     // Seed the random number generator by passing
46     // the return value of time(0) to srand.
47     srand( time(0) );
48
49     // Populate the array with random numbers.
50     for (int count = 0; count < num; count++)
51         arr[count] = rand();
52
53     // Return a pointer to the array.
54     return arr;
55 }
```





9.10

Using Smart Pointers to Avoid Memory Leaks



Using Smart Pointers to Avoid Memory Leaks

- In C++ 11, you can use *smart pointers* to dynamically allocate memory and not worry about deleting the memory when you are finished using it.
- Three types of smart pointer:

unique_ptr
shared_ptr
weak_ptr

- Must #include the memory header file:

```
#include <memory>
```

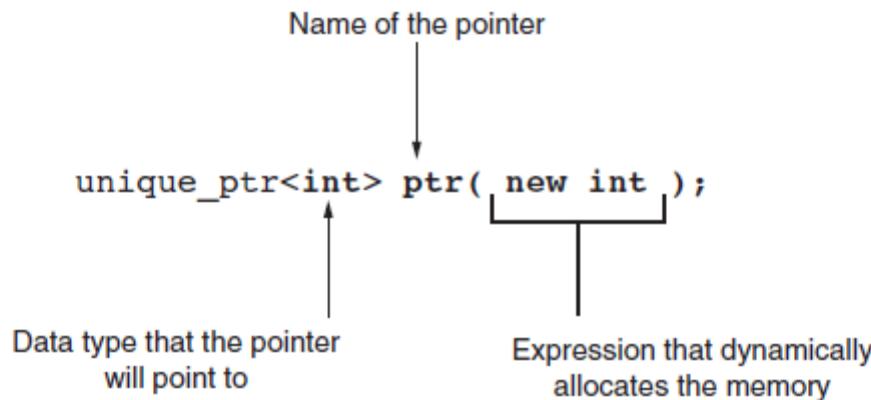
- In this book, we introduce unique_ptr:

```
unique_ptr<int> ptr( new int );
```



Using Smart Pointers to Avoid Memory Leaks

Figure 9-12



- The notation `<int>` indicates that the pointer can point to an `int`.
- The name of the pointer is `ptr`.
- The expression `new int` allocates a chunk of memory to hold an `int`.
- The address of the chunk of memory will be assigned to `ptr`.



Using Smart Pointers in Program 9-17

Program 9-17

```
1 // This program demonstrates a unique_ptr.  
2 #include <iostream>  
3 #include <memory>  
4 using namespace std;  
5  
6 int main()  
7 {  
8     // Define a unique_ptr smart pointer, pointing  
9     // to a dynamically allocated int.  
10    unique_ptr<int> ptr( new int );  
11  
12    // Assign 99 to the dynamically allocated int.  
13    *ptr = 99;  
14  
15    // Display the value of the dynamically allocated int.  
16    cout << *ptr << endl;  
17    return 0;  
18 }
```

Program Output

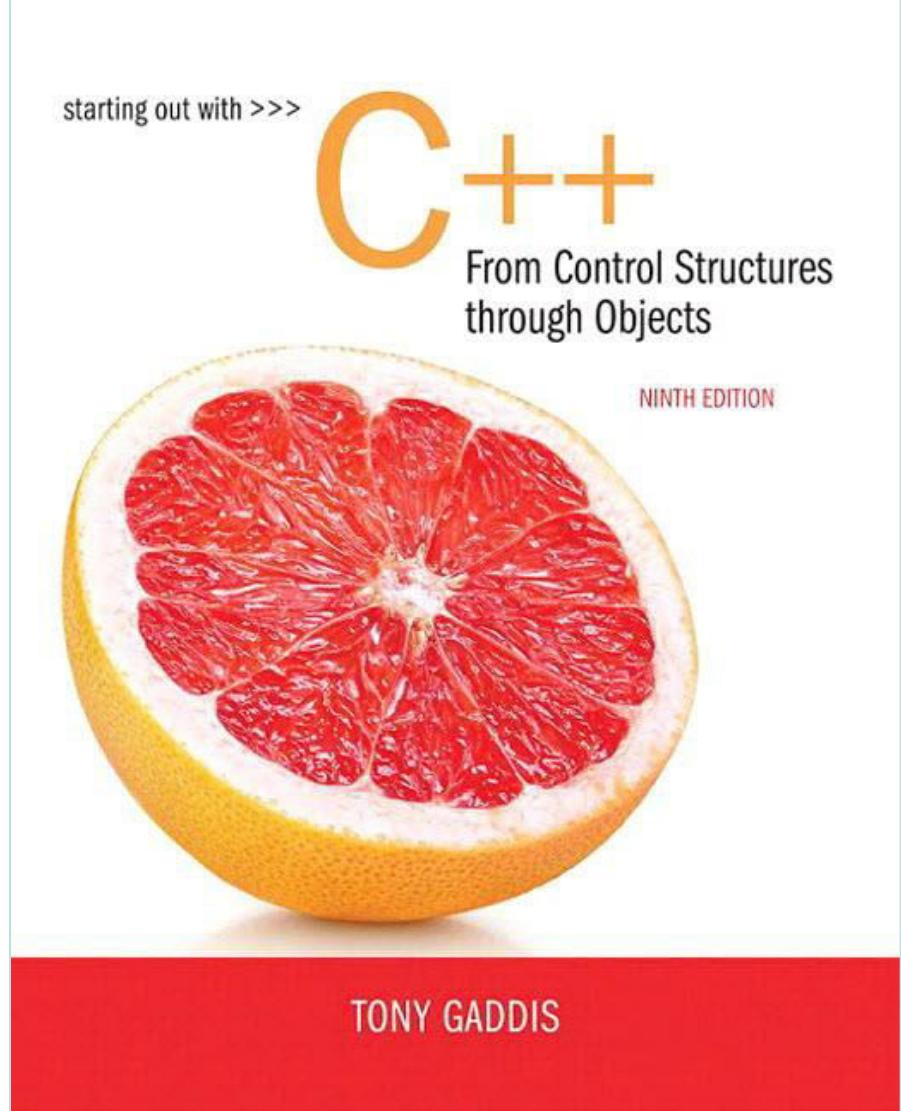
```
99
```

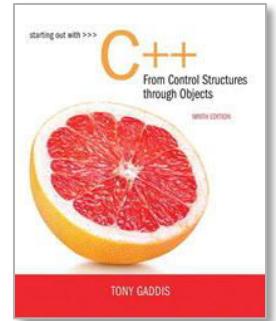


Chapter 10:

Characters, C- Strings, and

More About the string Class





10.1

Character Testing



Character Testing

- Requires `cctype` header file

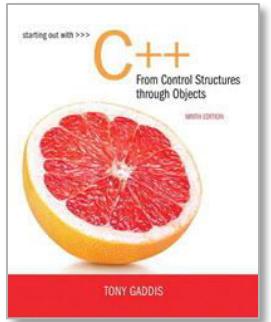
FUNCTION	MEANING
<code>isalpha</code>	true if arg. is a letter, false otherwise
<code>isalnum</code>	true if arg. is a letter or digit, false otherwise
<code>isdigit</code>	true if arg. is a digit 0-9, false otherwise
<code>islower</code>	true if arg. is lowercase letter, false otherwise
<code>isprint</code>	true if arg. is a printable character, false otherwise
<code>ispunct</code>	true if arg. is a punctuation character, false otherwise
<code>isupper</code>	true if arg. is an uppercase letter, false otherwise
<code>isspace</code>	true if arg. is a whitespace character, false otherwise



From Program 10-1

```
10    cout << "Enter any character: ";
11    cin.get(input);
12    cout << "The character you entered is: " << input << endl;
13    if (isalpha(input))
14        cout << "That's an alphabetic character.\n";
15    if (isdigit(input))
16        cout << "That's a numeric digit.\n";
17    if (islower(input))
18        cout << "The letter you entered is lowercase.\n";
19    if (isupper(input))
20        cout << "The letter you entered is uppercase.\n";
21    if (isspace(input))
22        cout << "That's a whitespace character.\n";
```





10.2

Character Case Conversion



Character Case Conversion

- Require `cctype` header file
- Functions:

`toupper`: if `char` argument is lowercase letter, return uppercase equivalent; otherwise, return input unchanged

```
char ch1 = 'H';
char ch2 = 'e';
char ch3 = '!';

cout << toupper(ch1); // displays 'H'
cout << toupper(ch2); // displays 'E'
cout << toupper(ch3); // displays '!'
```



Character Case Conversion

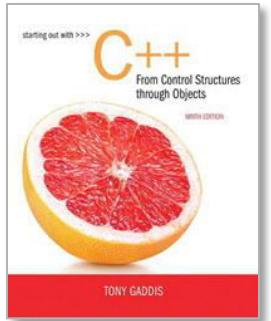
● Functions:

`tolower`: if `char` argument is uppercase letter, return lowercase equivalent; otherwise, return input unchanged

```
char ch1 = 'H';
char ch2 = 'e';
char ch3 = '!';

cout << tolower(ch1); // displays 'h'
cout << tolower(ch2); // displays 'e'
cout << tolower(ch3); // displays '!'
```





10.3

C-Strings



C-Strings

- C-string: sequence of characters stored in adjacent memory locations and terminated by `\0` character
- String literal (string constant): sequence of characters enclosed in double quotes " " :
"Hi there!"

H	i		t	h	e	r	e	!	\0
---	---	--	---	---	---	---	---	---	----



C-Strings

- Array of `char`s can be used to define storage for string:

```
const int SIZE = 20;  
char city[SIZE];
```

- Leave room for `NULL` at end
- Can enter a value using `cin` or `>>`
 - Input is whitespace-terminated
 - No check to see if enough space
- For input containing whitespace, and to control amount of input, use `cin.getline()`



Using C-Strings in Program 10-5

Program 10-5

```
1 // This program displays a string stored in a char array.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     const int SIZE = 80;    // Array size
8     char line[SIZE];       // To hold a line of input
9     int count = 0;          // Loop counter variable
10
11    // Get a line of input.
12    cout << "Enter a sentence of no more than "
13        << (SIZE - 1) << " characters:\n";
14    cin.getline(line, SIZE);
15
16    // Display the input one character at a time.
17    cout << "The sentence you entered is:\n";
18    while (line[count] != '\0')
19    {
20        cout << line[count];
21        count++;
22    }
23    return 0;
24 }
```

Program Output with Example Input Shown in Bold

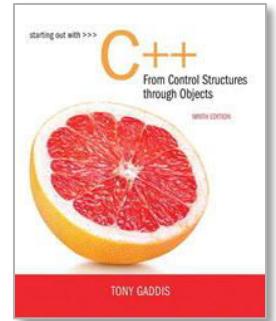
Enter a sentence of no more than 79 characters:

C++ is challenging but fun! [Enter]

The sentence you entered is:

C++ is challenging but fun!





10.4

Library Functions for Working with C-Strings



Library Functions for Working with C-Strings

- Require the `cstring` header file
- Functions take one or more C-strings as arguments. Can use:
 - C-string name
 - pointer to C-string
 - literal string



Library Functions for Working with C-Strings

Functions:

- `strlen(str)`: returns length of C-string str

```
char city[SIZE] = "Missoula";  
cout << strlen(city); // prints 8
```

- `strcat(str1, str2)`: appends str2 to the end of str1

```
char location[SIZE] = "Missoula, ";  
char state[3] = "MT";  
strcat(location, state);  
// location now has "Missoula, MT"
```



Library Functions for Working with C-Strings

Functions:

- orange icon **strncpy(str1, str2)**: copies str2 to str1

```
const int SIZE = 20;  
char fname[SIZE] = "Maureen", name[SIZE];  
strncpy(name, fname);
```

Note: `strcat` and `strncpy` perform no bounds checking to determine if there is enough space in receiving character array to hold the string it is being assigned.



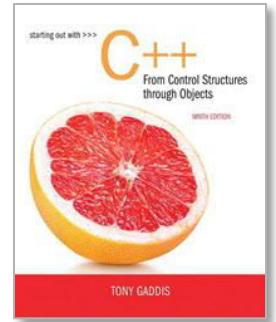
C-string Inside a C-string

Function:

- orange icon **strstr(str1, str2)**: finds the first occurrence of str2 in str1. Returns a pointer to match, or NULL if no match.

```
char river[] = "Wabash";  
char word[] = "aba";  
cout << strstr(state, word);  
// displays "abash"
```





10.5

C-String/Numeric Conversion Functions



C-String/Numeric Conversion Functions

- Requires `<cstdlib>` header file

FUNCTION	PARAMETER	ACTION
atoi	C-string	converts C-string to an <code>int</code> value, returns the value
atol	C-string	converts C-string to a <code>long</code> value, returns the value
atof	C-string	converts C-string to a <code>double</code> value, returns the value
itoa	<code>int</code> , C-string, <code>int</code>	converts 1 st <code>int</code> parameter to a C-string, stores it in 2 nd parameter. 3 rd parameter is base of converted value



C-String/Numeric Conversion Functions

```
int iNum;  
long lNum;  
double dNum;  
char intChar[10];  
  
iNum = atoi("1234"); // puts 1234 in iNum  
lNum = atol("5678"); // puts 5678 in lNum  
dNum = atof("35.7"); // puts 35.7 in dNum  
  
itoa(iNum, intChar, 8); // puts the string  
// "2322" (base 8 for 123410) in intChar
```



C-String/Numeric Conversion Functions - Notes

- if C-string contains non-digits, results are undefined
 - function may return result up to non-digit
 - function may return 0
- itoa does no bounds checking – make sure there is enough space to store the result



string to Number Conversion

Table 10-5 string to Number Functions

Function	Description
<code>stoi(string str)</code>	Accepts a <code>string</code> argument and returns that argument's value converted to an <code>int</code> .
<code>stol(string str)</code>	Accepts a <code>string</code> argument and returns that argument's value converted to a <code>long</code> .
<code>stoul(string str)</code>	Accepts a <code>string</code> argument and returns that argument's value converted to an <code>unsigned long</code> .
<code>stoll(string str)</code>	Accepts a <code>string</code> argument and returns that argument's value converted to a <code>long long</code> .
<code>stoull(string str)</code>	Accepts a <code>string</code> argument and returns that argument's value converted to an <code>unsigned long long</code> .
<code>stof(string str)</code>	Accepts a <code>string</code> argument and returns that argument's value converted to a <code>float</code> .
<code>stod(string str)</code>	Accepts a <code>string</code> argument and returns that argument's value converted to a <code>double</code> .
<code>stold(string str)</code>	Accepts a <code>string</code> argument and returns that argument's value converted to a <code>long double</code> .

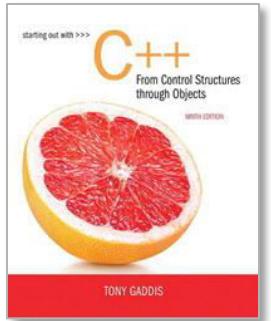


The `to_string` Function

Table 10-6 Overloaded Versions of the `to_string` Function

Function	Description
<code>to_string(int value);</code>	Accepts an <code>int</code> argument and returns that argument converted to a <code>string</code> object.
<code>to_string(long value);</code>	Accepts a <code>long</code> argument and returns that argument converted to a <code>string</code> object.
<code>to_string(long long value);</code>	Accepts a <code>long long</code> argument and returns that argument converted to a <code>string</code> object.
<code>to_string(unsigned value);</code>	Accepts an <code>unsigned</code> argument and returns that argument converted to a <code>string</code> object.
<code>to_string(unsigned long value);</code>	Accepts an <code>unsigned long</code> argument and returns that argument converted to a <code>string</code> object.
<code>to_string(unsigned long long value);</code>	Accepts an <code>unsigned long long</code> argument and returns that argument converted to a <code>string</code> object.
<code>to_string(float value);</code>	Accepts a <code>float</code> argument and returns that argument converted to a <code>string</code> object.
<code>to_string(double value);</code>	Accepts a <code>double</code> argument and returns that argument converted to a <code>string</code> object.
<code>to_string(long double value);</code>	Accepts a <code>long double</code> argument and returns that argument converted to a <code>string</code> object.





10.6

Writing Your Own C-String Handling Functions



Writing Your Own C-String Handling Functions

- Orange Designing C-String Handling Functions
 - Orange can pass arrays or pointers to char arrays
 - Orange Can perform bounds checking to ensure enough space for results
 - Orange Can anticipate unexpected user input



From Program 10-12

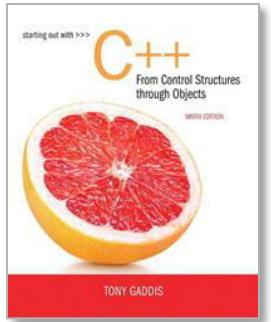
```
31 void stringCopy(char string1[], char string2[])
32 {
33     int index = 0; // Loop counter
34
35     // Step through string1, copying each element to
36     // string2. Stop when the null character is encountered.
37     while (string1[index] != '\0')
38     {
39         string2[index] = string1[index];
40         index++;
41     }
42
43     // Place a null character in string2.
44     string2[index] = '\0';
45 }
```



From Program 10-13

```
29 void nameSlice(char userName[])
30 {
31     int count = 0; // Loop counter
32
33     // Locate the first space, or the null terminator if there
34     // are no spaces.
35     while (userName[count] != ' ' && userName[count] != '\0')
36         count++;
37
38     // If a space was found, replace it with a null terminator.
39     if (userName[count] == ' ')
40         userName[count] = '\0';
41 }
```





10.7

More About the C++ string Class



The C++ string Class

- Orange Special data type supports working with strings

- Orange #include <string>

- Orange Can define string variables in programs:

```
string firstName, lastName;
```

- Orange Can receive values with assignment operator:

```
firstName = "George";
```

```
lastName = "Washington";
```

- Orange Can be displayed via cout

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



Using the **string** class in Program 10-15

Program 10-15

```
1 // This program demonstrates the string class.  
2 #include <iostream>  
3 #include <string>      // Required for the string class.  
4 using namespace std;  
5  
6 int main()  
7 {  
8     string movieTitle;  
9  
10    movieTitle = "Wheels of Fury";  
11    cout << "My favorite movie is " << movieTitle << endl;  
12    return 0;  
13 }
```

Program Output

My favorite movie is Wheels of Fury



Input into a string Object

- Use `cin >>` to read an item into a string:

```
string firstName;  
cout << "Enter your first name: ";  
cin >> firstName;
```



Using `cin` and `string` objects in program 10-16

Program 10-16

```
1 // This program demonstrates how cin can read a string into
2 // a string class object.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string name;
10
11    cout << "What is your name? ";
12    cin >> name;
13    cout << "Good morning " << name << endl;
14    return 0;
15 }
```

Program Output with Example Input Shown in Bold

What is your name? **Peggy** [Enter]

Good morning Peggy



Input into a string Object

- Use `getline` function to put a line of input, possibly including spaces, into a string:

```
string address;  
cout << "Enter your address: ";  
getline(cin, address);
```



string Comparison

- Can use relational operators directly to compare string objects:

```
string str1 = "George",
        str2 = "Georgia";
if (str1 < str2)
    cout << str1 << " is less than "
        << str2;
```

- Comparison is performed similar to strcmp function.
Result is true or false



Program 10-18

```
1 // This program uses relational operators to alphabetically
2 // sort two strings entered by the user.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main ()
8 {
9     string name1, name2;
10
11    // Get a name.
12    cout << "Enter a name (last name first): ";
13    getline(cin, name1);
14
15    // Get another name.
16    cout << "Enter another name: ";
17    getline(cin, name2);
18
19    // Display them in alphabetical order.
20    cout << "Here are the names sorted alphabetically:\n";
21    if (name1 < name2)
22        cout << name1 << endl << name2 << endl;
23    else if (name1 > name2)
24        cout << name2 << endl << name1 << endl;
25    else
26        cout << "You entered the same name twice!\n";
27    return 0;
28 }
```

Program Output with Example Input Shown in Bold

```
Enter a name (last name first): Smith, Richard 
Enter another name: Jones, John 
Here are the names sorted alphabetically:
Jones, John
Smith, Richard
```



Other Definitions of C++ strings

Definition	Meaning
string name;	defines an empty string object
string myname ("Chris");	defines a string and initializes it
string yourname (myname);	defines a string and initializes it
string fname (myname, 3);	defines a string and initializes it with first 3 characters of myname
string verb (myname, 3, 2);	defines a string and initializes it with 2 characters from myname starting at position 3
string noname ('A', 5);	defines string and initializes it to 5 'A's



string Operators

OPERATOR	MEANING
>>	extracts characters from stream up to whitespace, insert into string
<<	inserts string into stream
=	assigns string on right to string object on left
+=	appends string on right to end of contents on left
+	concatenates two strings
[]	references character in string using array notation
>, >=, <, <=, ==, !=	relational operators for string comparison. Return true or false



string Operators

```
string word1, phrase;
string word2 = " Dog";
cin >> word1; // user enters "Hot Tamale"
                // word1 has "Hot"
phrase = word1 + word2; // phrase has
                        // "Hot Dog"
phrase += " on a bun";
for (int i = 0; i < 16; i++)
    cout << phrase[i]; // displays
                        // "Hot Dog on a bun"
```



Program 10-20

```
1 // This program demonstrates the C++ string class.  
2 #include <iostream>  
3 #include <string>  
4 using namespace std;  
5  
6 int main ()  
7 {  
8     // Define three string objects.  
9     string str1, str2, str3;  
10  
11    // Assign values to all three.  
12    str1 = "ABC";  
13    str2 = "DEF";  
14    str3 = str1 + str2;  
15  
16    // Display all three.  
17    cout << str1 << endl;  
18    cout << str2 << endl;  
19    cout << str3 << endl;  
20  
21    // Concatenate a string onto str3 and display it.  
22    str3 += "GHI";  
23    cout << str3 << endl;  
24    return 0;  
25 }
```

Program Output

ABC
DEF
ABCDEF
ABCDEFGH



string Member Functions

- Are behind many overloaded operators
- Categories:
 - **assignment:** assign, copy, data
 - **modification:** append, clear, erase, insert, replace, swap
 - **space management:** capacity, empty, length, resize, size
 - **substrings:** find, front, back, at, substr
 - **comparison:** compare
- See Table 10-8 for a list of functions.



string Member Functions

```
string word1, word2, phrase;  
cin >> word1;           // word1 is "Hot"  
word2.assign(" Dog");  
phrase.append(word1);  
phrase.append(word2);  // phrase has "Hot Dog"  
phrase.append(" with mustard relish", 13);  
                      // phrase has "Hot Dog with mustard"  
phrase.insert(8, "on a bun ");  
cout << phrase << endl; // displays  
                      // "Hot Dog on a bun with mustard"
```



string Member Functions in Program 10-21

Program 10-21

```
1 // This program demonstrates a string
2 // object's length member function.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main ()
8 {
9     string town;
10
11    cout << "Where do you live? ";
12    cin >> town;
13    cout << "Your town's name has " << town.length() ;
14    cout << " characters\n";
15    return 0;
16 }
```

Program Output with Example Input Shown in Bold

Where do you live? **Jacksonville** [Enter]

Your town's name has 12 characters



Chapter 11:

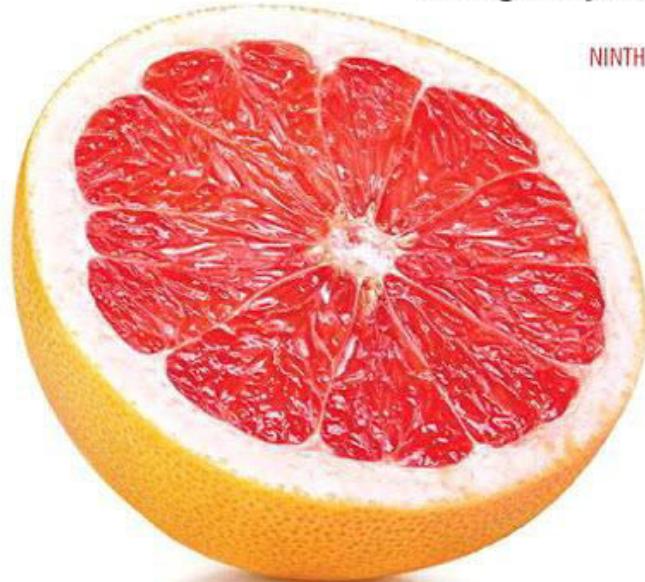
Structured Data

starting out with >>>

C++

From Control Structures
through Objects

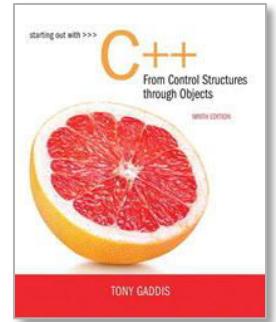
NINTH EDITION



TONY GADDIS



Pearson Copyright © 2018, 2015, 2012, 2009 Pearson Education, Inc. All rights reserved.



11.1

Abstract Data Types



Abstract Data Types

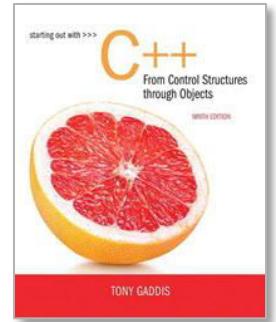
- A data type that specifies
 - values that can be stored
 - operations that can be done on the values
- User of an abstract data type does not need to know the implementation of the data type, e.g., how the data is stored
- ADTs are created by programmers



Abstraction and Data Types

- **Abstraction:** a definition that captures general characteristics without details
 - Ex: An abstract triangle is a 3-sided polygon. A specific triangle may be scalene, isosceles, or equilateral
- **Data Type** defines the values that can be stored in a variable and the operations that can be performed on it





11.2

Combining Data into Structures



Combining Data into Structures

- **Structure:** C++ construct that allows multiple variables to be grouped together
- General Format:

```
struct <structName>
{
    type1 field1;
    type2 field2;
    ...
};
```



Example struct Declaration

```
struct Student {  
    int studentID;  
    string name;  
    short yearInSchool;  
    double gpa;  
};
```

The diagram illustrates the structure of a C-style struct declaration. It shows the 'structure tag' (the identifier 'Student') and the 'structure members' (the data fields 'studentID', 'name', 'yearInSchool', and 'gpa'). The code is annotated with orange arrows and brackets to highlight these components.



struct Declaration Notes

- Must have ; after closing }
- struct names commonly begin with uppercase letter
- Multiple fields of same type can be in comma-separated list:

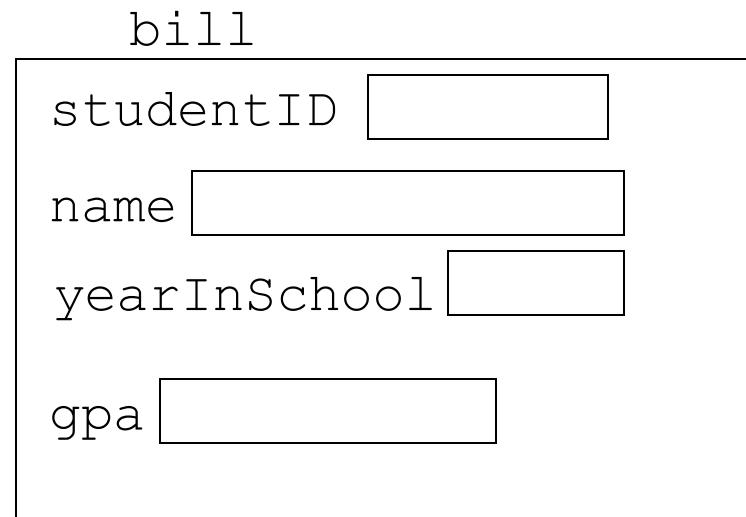
```
string name,  
       address;
```

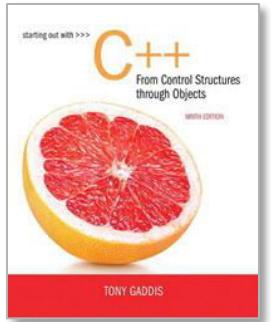


Defining Variables

- struct declaration does not allocate memory or create variables
- To define variables, use structure tag as type name:

```
Student bill;
```





11.3

Accessing Structure Members



Accessing Structure Members

- Orange Use the dot (.) operator to refer to members of struct variables:

```
cin >> stu1.studentID;  
getline(cin, stu1.name);  
stu1.gpa = 3.75;
```

- Orange Member variables can be used in any manner appropriate for their data type



Program 11-1

```
1 // This program demonstrates the use of structures.  
2 #include <iostream>  
3 #include <string>  
4 #include <iomanip>  
5 using namespace std;  
6  
7 struct PayRoll  
8 {  
9     int empNumber;      // Employee number  
10    string name;        // Employee's name  
11    double hours;       // Hours worked  
12    double payRate;     // Hourly payRate  
13    double grossPay;    // Gross pay  
14};  
15  
16 int main()  
17 {  
18     PayRoll employee; // employee is a PayRoll structure.  
19  
20     // Get the employee's number.  
21     cout << "Enter the employee's number: ";  
22     cin >> employee.empNumber;  
23  
24     // Get the employee's name.  
25     cout << "Enter the employee's name: ";
```



```
26     cin.ignore(); // To skip the remaining '\n' character
27     getline(cin, employee.name);
28
29     // Get the hours worked by the employee.
30     cout << "How many hours did the employee work? ";
31     cin >> employee.hours;
32
33     // Get the employee's hourly pay rate.
34     cout << "What is the employee's hourly payRate? ";
35     cin >> employee.payRate;
36
37     // Calculate the employee's gross pay.
38     employee.grossPay = employee.hours * employee.payRate;
39
40     // Display the employee data.
41     cout << "Here is the employee's payroll data:\n";
42     cout << "Name: " << employee.name << endl;
43     cout << "Number: " << employee.empNumber << endl;
44     cout << "Hours worked: " << employee.hours << endl;
45     cout << "Hourly payRate: " << employee.payRate << endl;
46     cout << fixed << showpoint << setprecision(2);
47     cout << "Gross Pay: $" << employee.grossPay << endl;
48
49 }
```



Program Output with Example Input Shown in Bold

Enter the employee's number: **489 [Enter]**

Enter the employee's name: **Jill Smith [Enter]**

How many hours did the employee work? **40 [Enter]**

What is the employee's hourly pay rate? **20 [Enter]**

Here is the employee's payroll data:

Name: Jill Smith

Number: 489

Hours worked: 40

Hourly pay rate: 20

Gross pay: \$800.00



Displaying a struct Variable

- To display the contents of a struct variable, must display each field separately, using the dot operator:

```
cout << bill; // won't work  
cout << bill.studentID << endl;  
cout << bill.name << endl;  
cout << bill.yearInSchool;  
cout << " " << bill.gpa;
```



Comparing struct Variables

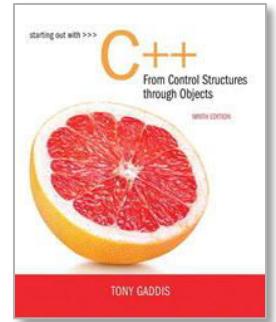
- Orange icon: Cannot compare struct variables directly:

```
if (bill == william) // won't work
```

- Orange icon: Instead, must compare on a field basis:

```
if (bill.studentID ==  
    william.studentID) ...
```





11.4

Initializing a Structure



Initializing a Structure

- struct variable can be initialized when defined:

```
Student s = {11465, "Joan", 2, 3.75};
```

- Can also be initialized member-by-member after definition:

```
s.name = "Joan";
```

```
s.gpa = 3.75;
```



More on Initializing a Structure

- May initialize only some members:

```
Student bill = {14579};
```

- Cannot skip over members:

```
Student s = {1234, "John", ,  
             2.83}; // illegal
```

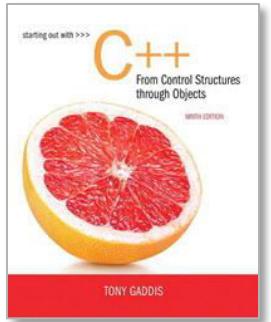
- Cannot initialize in the structure declaration, since this does not allocate memory



Excerpts From Program 11-3

```
8 struct EmployeePay
9 {
10     string name;           // Employee name
11     int empNum;           // Employee number
12     double payRate;        // Hourly pay rate
13     double hours;          // Hours worked
14     double grossPay;       // Gross pay
15 };
19 EmployeePay employee1 = {"Betty Ross", 141, 18.75};
20 EmployeePay employee2 = {"Jill Sandburg", 142, 17.50};
```





11.5

Arrays of Structures



Arrays of Structures

- Structures can be defined in arrays
- Can be used in place of parallel arrays

```
const int NUM_STUDENTS = 20;  
Student stuList[NUM_STUDENTS];
```

- Individual structures accessible using subscript notation
- Fields within structures accessible using dot notation:

```
cout << stuList[5].studentID;
```



Program 11-4

```
1 // This program uses an array of structures.  
2 #include <iostream>  
3 #include <iomanip>  
4 using namespace std;  
5  
6 struct PayInfo  
7 {  
8     int hours;          // Hours worked  
9     double payRate;    // Hourly pay rate  
10};  
11  
12 int main()  
13 {  
14     const int NUM_WORKERS = 3;      // Number of workers  
15     PayInfo workers[NUM_WORKERS]; // Array of structures  
16     int index;                  // Loop counter  
17}
```



```
18 // Get employee pay data.  
19 cout << "Enter the hours worked by " << NUM_WORKERS  
20     << " employees and their hourly rates.\n";  
21  
22 for (index = 0; index < NUM_WORKERS; index++)  
23 {  
24     // Get the hours worked by an employee.  
25     cout << "Hours worked by employee #" << (index + 1);  
26     cout << ":";  
27     cin >> workers[index].hours;  
28  
29     // Get the employee's hourly pay rate.  
30     cout << "Hourly pay rate for employee #";  
31     cout << (index + 1) << ":";  
32     cin >> workers[index].payRate;  
33     cout << endl;  
34 }  
35  
36 // Display each employee's gross pay.  
37 cout << "Here is the gross pay for each employee:\n";  
38 cout << fixed << showpoint << setprecision(2);  
39 for (index = 0; index < NUM_WORKERS; index++)  
40 {  
41     double gross;  
42     gross = workers[index].hours * workers[index].payRate;  
43     cout << "Employee #" << (index + 1);  
44     cout << ":" $" << gross << endl;  
45 }  
46 return 0;  
47 }
```



Program Output with Example Input Shown in Bold

Enter the hours worked by 3 employees and their hourly rates.

Hours worked by employee #1: **10 [Enter]**

Hourly pay rate for employee #1: **9.75 [Enter]**

Hours worked by employee #2: **20 [Enter]**

Hourly pay rate for employee #2: **10.00 [Enter]**

Hours worked by employee #3: **40 [Enter]**

Hourly pay rate for employee #3: **20.00 [Enter]**

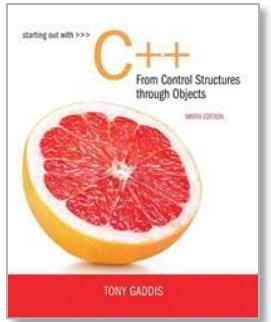
Here is the gross pay for each employee:

Employee #1: \$97.50

Employee #2: \$200.00

Employee #3: \$800.00





11.6

Nested Structures



Nested Structures

A structure can contain another structure as a member:

```
struct PersonInfo
{
    string name,
          address,
          city;
};

struct Student
{
    int studentID;
    PersonInfo pData;
    short yearInSchool;
    double gpa;
};
```

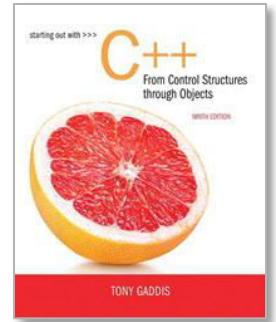


Members of Nested Structures

- Use the dot operator multiple times to refer to fields of nested structures:

```
Student s;  
s.pData.name = "Joanne";  
s.pData.city = "Tulsa";
```





11.7

Structures as Function Arguments



Structures as Function Arguments

- May pass members of struct variables to functions:

```
computeGPA(stu.gpa);
```

- May pass entire struct variables to functions:

```
showData(stu);
```

- Can use reference parameter if function needs to modify contents of structure variable



Excerpts from Program 11-6

```
8  struct InventoryItem
9  {
10     int partNum;                      // Part number
11     string description;              // Item description
12     int onHand;                      // Units on hand
13     double price;                   // Unit price
14 };
15
16 void showItem(InventoryItem p)
17 {
18     cout << fixed << showpoint << setprecision(2);
19     cout << "Part Number: " << p.partNum << endl;
20     cout << "Description: " << p.description << endl;
21     cout << "Units On Hand: " << p.onHand << endl;
22     cout << "Price: $" << p.price << endl;
23 }
```



Structures as Function Arguments - Notes

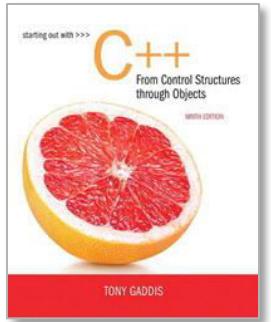
- Using value parameter for structure can slow down a program, waste space
- Using a reference parameter will speed up program, but function may change data in structure
- Using a `const` reference parameter allows read-only access to reference parameter, does not waste space, speed



Revised showItem Function

```
void showItem(const InventoryItem &p)
{
    cout << fixed << showpoint << setprecision(2);
    cout << "Part Number: " << p.partNum << endl;
    cout << "Description: " << p.description << endl;
    cout << "Units On Hand: " << p.onHand << endl;
    cout << "Price: $" << p.price << endl;
}
```





11.8

Returning a Structure from a Function



Returning a Structure from a Function

- Function can return a struct:

```
Student getStudentData(); // prototype  
stu1 = getStudentData(); // call
```

- Function must define a local structure
 - for internal use
 - for use with return statement



Returning a Structure from a Function - Example

```
Student getStudentData()
{
    Student tempStu;
    cin >> tempStu.studentID;
    getline(cin, tempStu.pData.name);
    getline(cin, tempStu.pData.address);
    getline(cin, tempStu.pData.city);
    cin >> tempStu.yearInSchool;
    cin >> tempStu.gpa;
    return tempStu;
}
```



Program 11-7

```
1 // This program uses a function to return a structure. This
2 // is a modification of Program 11-2.
3 #include <iostream>
4 #include <iomanip>
5 #include <cmath> // For the pow function
6 using namespace std;
7
8 // Constant for pi.
9 const double PI = 3.14159;
10
11 // Structure declaration
12 struct Circle
13 {
14     double radius;          // A circle's radius
15     double diameter;        // A circle's diameter
16     double area;            // A circle's area
17 };
18
19 // Function prototype
20 Circle getInfo();
21
22 int main()
23 {
24     Circle c;              // Define a structure variable
```



```
25
26     // Get data about the circle.
27     c = getInfo();
28
29     // Calculate the circle's area.
30     c.area = PI * pow(c.radius, 2.0);
31
32     // Display the circle data.
33     cout << "The radius and area of the circle are:\n";
34     cout << fixed << setprecision(2);
35     cout << "Radius: " << c.radius << endl;
36     cout << "Area: " << c.area << endl;
37     return 0;
38 }
39
```



```
40 //*****
41 // Definition of function getInfo. This function uses a local      *
42 // variable, tempCircle, which is a circle structure. The user      *
43 // enters the diameter of the circle, which is stored in          *
44 // tempCircle.diameter. The function then calculates the radius   *
45 // which is stored in tempCircle.radius. tempCircle is then       *
46 // returned from the function.                                     *
47 //*****
48
49 Circle getInfo()
50 {
51     Circle tempCircle; // Temporary structure variable
52
53     // Store circle data in the temporary variable.
54     cout << "Enter the diameter of a circle: ";
55     cin >> tempCircle.diameter;
56     tempCircle.radius = tempCircle.diameter / 2.0;
57
58     // Return the temporary variable.
59     return tempCircle;
60 }
```

Program Output with Example Input Shown in Bold

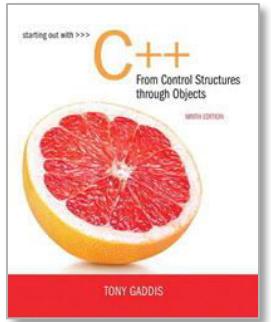
Enter the diameter of a circle: **10 [Enter]**

The radius and area of the circle are:

Radius: 5.00

Area: 78.54





11.9

Pointers to Structures



Pointers to Structures

- A structure variable has an address
- Pointers to structures are variables that can hold the address of a structure:

```
Student *stuPtr;
```

- Can use & operator to assign address:

```
stuPtr = & stu1;
```
- Structure pointer can be a function parameter



Accessing Structure Members via Pointer Variables

- Must use () to dereference pointer variable, not field within structure:

```
cout << (*stuPtr).studentID;
```

- Can use structure pointer operator to eliminate () and use clearer notation:

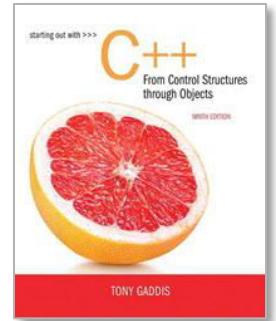
```
cout << stuPtr->studentID;
```



From Program 11-8

```
42 void getData(Student *s)
43 {
44     // Get the student name.
45     cout << "Student name: ";
46     getline(cin, s->name);
47
48     // Get the student ID number.
49     cout << "Student ID Number: ";
50     cin >> s->idNum;
51
52     // Get the credit hours enrolled.
53     cout << "Credit Hours Enrolled: ";
54     cin >> s->creditHours;
55
56     // Get the GPA.
57     cout << "Current GPA: ";
58     cin >> s->gpa;
59 }
```





11.11

Enumerated Data Types



Enumerated Data Types

- An enumerated data type is a programmer-defined data type. It consists of values known as *enumerators*, which represent integer constants.



Enumerated Data Types

Example:

```
enum Day { MONDAY, TUESDAY,  
          WEDNESDAY, THURSDAY,  
          FRIDAY } ;
```

The identifiers MONDAY, TUESDAY, WEDNESDAY, THURSDAY, and FRIDAY, which are listed inside the braces, are *enumerators*. They represent the values that belong to the Day data type.



Enumerated Data Types

```
enum Day { MONDAY, TUESDAY,  
          WEDNESDAY, THURSDAY,  
          FRIDAY } ;
```

Note that the enumerators are not strings,
so they aren't enclosed in quotes.
They are identifiers.



Enumerated Data Types

- Once you have created an enumerated data type in your program, you can define variables of that type. Example:

```
Day workDay;
```

- This statement defines `workDay` as a variable of the `Day` type.



Enumerated Data Types

- We may assign any of the enumerators MONDAY, TUESDAY, WEDNESDAY, THURSDAY, or FRIDAY to a variable of the Day type. Example:

```
workDay = WEDNESDAY;
```



Enumerated Data Types

- So, what is an *enumerator*?
- Think of it as an integer named constant
- Internally, the compiler assigns integer values to the enumerators, beginning at 0.



Enumerated Data Types

```
enum Day { MONDAY, TUESDAY,  
          WEDNESDAY, THURSDAY,  
          FRIDAY } ;
```

In memory...

MONDAY = 0

TUESDAY = 1

WEDNESDAY = 2

THURSDAY = 3

FRIDAY = 4



Enumerated Data Types

- Using the Day declaration, the following code...

```
cout << MONDAY << " "
    << WEDNESDAY << " "
    << FRIDAY << endl;
```

...will produce this output:

0 2 4



Assigning an integer to an enum Variable

- You cannot directly assign an integer value to an enum variable. This will not work:

```
workDay = 3; // Error!
```

- Instead, you must cast the integer:

```
workDay = static_cast<Day>(3);
```



Assigning an Enumerator to an int Variable

- You CAN assign an enumerator to an int variable. For example:

```
int x;  
x = THURSDAY;
```

- This code assigns 3 to x.



Comparing Enumerator Values

- Enumerator values can be compared using the relational operators. For example, using the Day data type the following code will display the message "Friday is greater than Monday."

```
if (FRIDAY > MONDAY)
{
    cout << "Friday is greater "
        << "than Monday.\n";
}
```



Program 11-9

```
1 // This program demonstrates an enumerated data type.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
7
8 int main()
9 {
10     const int NUM_DAYS = 5;           // The number of days
11     double sales[NUM_DAYS];         // To hold sales for each day
12     double total = 0.0;             // Accumulator
13     int index;                     // Loop counter
14
15     // Get the sales for each day.
16     for (index = MONDAY; index <= FRIDAY; index++)
17     {
18         cout << "Enter the sales for day "
19             << index << ": ";
20         cin >> sales[index];
21     }
22 }
```



Program 11-9 (Continued)

```
23     // Calculate the total sales.  
24     for (index = MONDAY; index <= FRIDAY; index++)  
25         total += sales[index];  
26  
27     // Display the total.  
28     cout << "The total sales are $" << setprecision(2)  
29         << fixed << total << endl;  
30  
31     return 0;  
32 }
```

Program Output with Example Input Shown in Bold

Enter the sales for day 0: **1525.00**

Enter the sales for day 1: **1896.50**

Enter the sales for day 2: **1975.63**

Enter the sales for day 3: **1678.33**

Enter the sales for day 4: **1498.52**

The total sales are \$8573.98



Enumerated Data Types

- Program 11-9 shows enumerators used to control a loop:

```
// Get the sales for each day.  
for (index = MONDAY; index <= FRIDAY; index++)  
{  
    cout << "Enter the sales for day "  
        << index << ":";  
    cin >> sales[index];  
}
```



Anonymous Enumerated Types

- An *anonymous enumerated type* is simply one that does not have a name. For example, in Program 11-10 we could have declared the enumerated type as:

```
enum { MONDAY, TUESDAY,  
       WEDNESDAY, THURSDAY,  
       FRIDAY } ;
```



Using Math Operators with enum Variables

- You can run into problems when trying to perform math operations with `enum` variables. For example:

```
Day day1, day2; // Define two Day variables.  
day1 = TUESDAY; // Assign TUESDAY to day1.  
day2 = day1 + 1; // ERROR! Will not work!
```

- The third statement will not work because the expression `day1 + 1` results in the integer value 2, and you cannot store an `int` in an `enum` variable.



Using Math Operators with enum Variables

- Orange You can fix this by using a cast to explicitly convert the result to Day, as shown here:

```
// This will work.  
day2 = static_cast<Day>(day1 + 1);
```



Using an enum Variable to Step through an Array's Elements

- Because enumerators are stored in memory as integers, you can use them as array subscripts. For example:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY,  
          THURSDAY, FRIDAY } ;  
const int NUM_DAYS = 5;  
double sales[NUM_DAYS];  
sales[MONDAY] = 1525.0;  
sales[TUESDAY] = 1896.5;  
sales[WEDNESDAY] = 1975.63;  
sales[THURSDAY] = 1678.33;  
sales[FRIDAY] = 1498.52;
```



Using an enum Variable to Step through an Array's Elements

- Remember, though, you cannot use the `++` operator on an enum variable. So, the following loop will NOT work.

```
Day workDay; // Define a Day variable
// ERROR!!! This code will NOT work.
for (workDay = MONDAY; workDay <= FRIDAY; workDay++)
{
    cout << "Enter the sales for day "
        << workDay << ":" ;
    cin >> sales[workDay];
}
```



Using an enum Variable to Step through an Array's Elements

- You must rewrite the loop's update expression using a cast instead of ++:

```
for (workDay = MONDAY; workDay <= FRIDAY;  
     workDay = static_cast<Day>(workDay + 1))  
{  
    cout << "Enter the sales for day "  
        << workDay << ":";  
    cin >> sales[workDay];  
}
```



Program 11-10

```
1 // This program demonstrates an enumerated data type.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
7
8 int main()
9 {
10     const int NUM_DAYS = 5;          // The number of days
11     double sales[NUM_DAYS];        // To hold sales for each day
12     double total = 0.0;            // Accumulator
13     Day workDay;                 // Loop counter
14
15     // Get the sales for each day.
16     for (workDay = MONDAY; workDay <= FRIDAY;
17                      workDay = static_cast<Day>(workDay + 1))
18     {
19         cout << "Enter the sales for day "
20             << workDay << ": ";
21         cin >> sales[workDay];
22     }
```



```
23
24     // Calculate the total sales.
25     for (workDay = MONDAY; workDay <= FRIDAY;
26             workDay = static_cast<Day>(workDay + 1))
27         total += sales[workDay];
28
29     // Display the total.
30     cout << "The total sales are $" << setprecision(2)
31             << fixed << total << endl;
32
33     return 0;
34 }
```

Program Output with Example Input Shown in Bold

Enter the sales for day 0: **1525.00**

Enter the sales for day 1: **1896.50**

Enter the sales for day 2: **1975.63**

Enter the sales for day 3: **1678.33**

Enter the sales for day 4: **1498.52**

The total sales are \$8573.98



Enumerators Must Be Unique Within the same Scope

- Enumerators must be unique within the same scope. (Unless strongly typed)
- For example, an error will result if both of the following enumerated types are declared within the same scope:

```
enum Presidents { MCKINLEY, ROOSEVELT, TAFT };
```

```
enum VicePresidents { ROOSEVELT, FAIRBANKS,  
                      SHERMAN };
```

ROOSEVELT is declared twice.



Using Strongly Typed enums in C++ 11

- In C++ 11, you can use a new type of `enum`, known as a *strongly typed enum*
- Allows you to have multiple enumerators in the same scope with the same name

```
enum class Presidents { MCKINLEY, ROOSEVELT, TAFT };  
enum class VicePresidents { ROOSEVELT, FAIRBANKS, SHERMAN };
```

- Prefix the enumerator with the name of the `enum`, followed by the `::` operator:

```
Presidents prez = Presidents::ROOSEVELT;  
VicePresidents vp = VicePresidents::ROOSEVELT;
```

- Use a cast operator to retrieve integer value:

```
int x = static_cast<int>(Presidents::ROOSEVELT);
```



Declaring the Type and Defining the Variables in One Statement

- You can declare an enumerated data type and define one or more variables of the type in the same statement. For example:

```
enum Car { PORSCHE, FERRARI, JAGUAR } sportsCar;
```

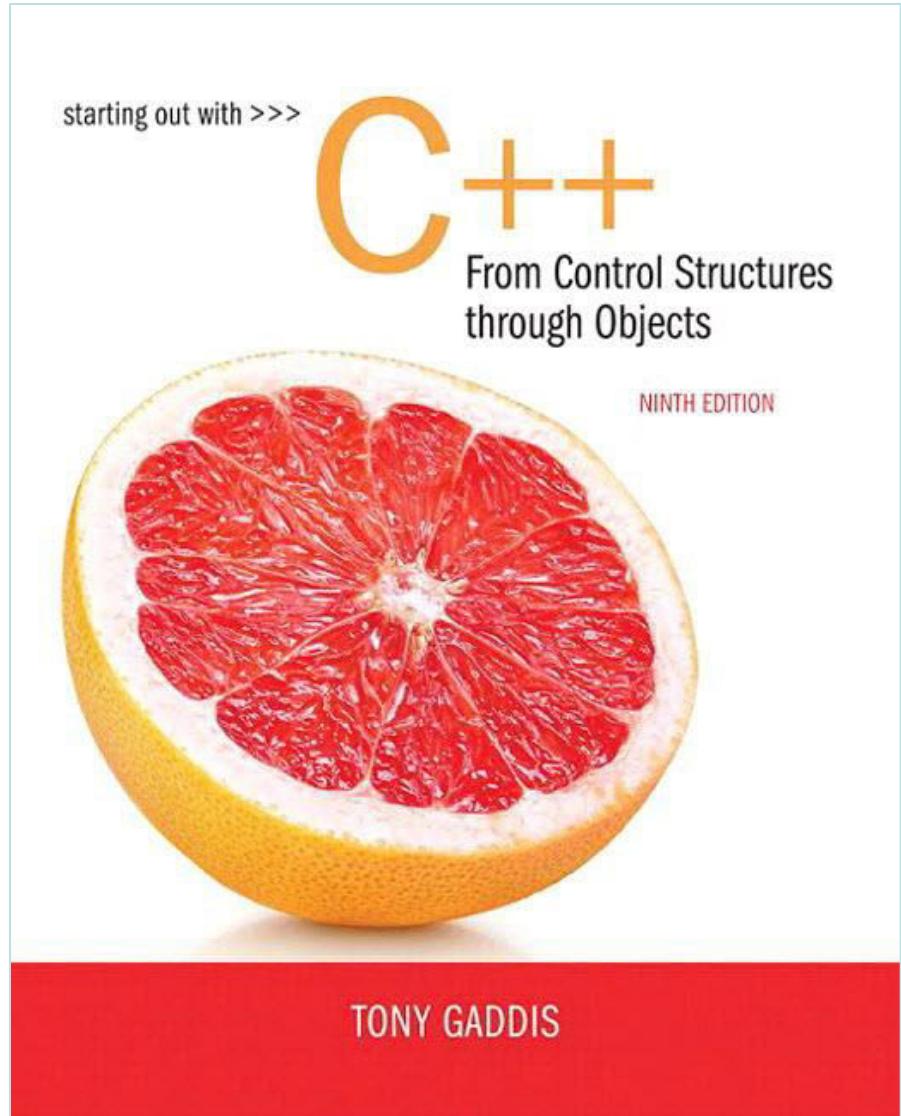
This code declares the `Car` data type and defines a variable named `sportsCar`.

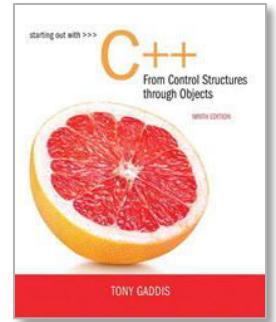


Chapter 12:

Advanced

File Operations





12.1

File Operations



File Operations

- File: a set of data stored on a computer, often on a disk drive
- Programs can read from, write to files
- Used in many applications:
 - Word processing
 - Databases
 - Spreadsheets
 - Compilers



Using Files

1. Requires `fstream` header file
 - use `ifstream` data type for input files
 - use `ofstream` data type for output files
 - use `fstream` data type for both input, output files
2. Can use `>>`, `<<` to read from, write to a file
3. Can use `eof` member function to test for end of input file



`fstream` Object

- `fstream` object can be used for either input or output
- Must specify mode on the `open` statement
- Sample modes:
 - `ios::in` – input
 - `ios::out` – output
- Can be combined on `open` call:

```
dFile.open("class.txt", ios::in | ios::out);
```



File Access Flags

Table 12-2

File Access Flag	Meaning
<code>ios::app</code>	Append mode. If the file already exists, its contents are preserved and all output is written to the end of the file. By default, this flag causes the file to be created if it does not exist.
<code>ios::ate</code>	If the file already exists, the program goes directly to the end of it. Output may be written anywhere in the file.
<code>ios::binary</code>	Binary mode. When a file is opened in binary mode, data is written to or read from it in pure binary format. (The default mode is text.)
<code>ios::in</code>	Input mode. Data will be read from the file. If the file does not exist, it will not be created and the open function will fail.
<code>ios::out</code>	Output mode. Data will be written to the file. By default, the file's contents will be deleted if it already exists.
<code>ios::trunc</code>	If the file already exists, its contents will be deleted (truncated). This is the default mode used by <code>ios::out</code> .



Using Files - Example

```
// copy 10 numbers between files
// open the files
fstream infile("input.txt", ios::in);
fstream outfile("output.txt", ios::out);
int num;
for (int i = 1; i <= 10; i++)
{
    infile >> num;      // use the files
    outfile << num;
}
infile.close();          // close the files
outfile.close();
```



Default File Open Modes

- ifstream:
 - open for input only
 - file cannot be written to
 - open fails if file does not exist

- ofstream:
 - open for output only
 - file cannot be read from
 - file created if no file exists
 - file contents erased if file exists



More File Open Details

- Can use filename, flags in definition:

```
ifstream gradeList ("grades.txt");
```

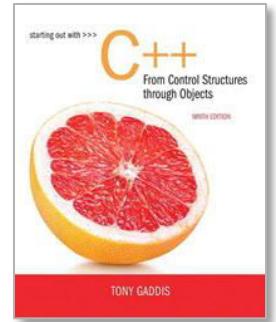
- File stream object set to 0 (false) if open failed:

```
if (!gradeList) ...
```

- Can also check fail member function to detect file open error:

```
if (gradeList.fail ()) ...
```





12.2

File Output Formatting



File Output Formatting

- Use the same techniques with file stream objects as with cout: showpoint, setw(x), showprecision(x), etc.
- Requires iomanip to use manipulators



Program 12-3

```
1 // This program uses the setprecision and fixed
2 // manipulators to format file output.
3 #include <iostream>
4 #include <iomanip>
5 #include <fstream>
6 using namespace std;
7
8 int main()
9 {
10     fstream dataFile;
11     double num = 17.816392;
12
13     dataFile.open("numfile.txt", ios::out);    // Open in output mode
14
15     dataFile << fixed;                      // Format for fixed-point notation
16     dataFile << num << endl;                // Write the number
17
18     dataFile << setprecision(4);            // Format for 4 decimal places
19     dataFile << num << endl;                // Write the number
20
21     dataFile << setprecision(3);            // Format for 3 decimal places
22     dataFile << num << endl;                // Write the number
23
```



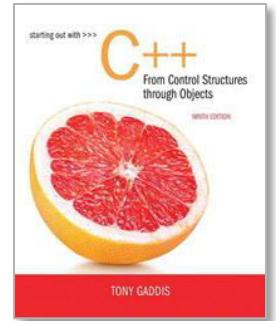
Program 12-3 (Continued)

```
24     dataFile << setprecision(2); // Format for 2 decimal places
25     dataFile << num << endl;    // Write the number
26
27     dataFile << setprecision(1); // Format for 1 decimal place
28     dataFile << num << endl;    // Write the number
29
30     cout << "Done.\n";
31     dataFile.close();          // Close the file
32     return 0;
33 }
```

Contents of File numfile.txt

```
17.816392
17.8164
17.816
17.82
17.8
```





12.3

Passing File Stream Objects to Functions



Passing File Stream Objects to Functions

- It is very useful to pass file stream objects to functions
- Be sure to always pass file stream objects by reference



Program 12-5

```
1 // This program demonstrates how file stream objects may
2 // be passed by reference to functions.
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 using namespace std;
7
8 // Function prototypes
9 bool openFileIn(fstream &, string);
10 void showContents(fstream &);
11
12 int main()
13 {
14     fstream dataFile;
15
16     if (openFileIn(dataFile, "demofile.txt"))
17     {
18         cout << "File opened successfully.\n";
19         cout << "Now reading data from the file.\n\n";
20         showContents(dataFile);
21         dataFile.close();
22         cout << "\nDone.\n";
23     }
}
```



```
24     else
25         cout << "File open error!" << endl;
26
27     return 0;
28 }
29
30 //*****
31 // Definition of function openFileIn. Accepts a reference *
32 // to an fstream object as an argument. The file is opened *
33 // for input. The function returns true upon success, false *
34 // upon failure.                                         *
35 //*****
36
37 bool openFileIn(fstream &file, string name)
38 {
39     file.open(name, ios::in);
40     if (file.fail())
41         return false;
42     else
43         return true;
44 }
45
46 //*****
47 // Definition of function showContents. Accepts an fstream *
48 // reference as its argument. Uses a loop to read each name *
49 // from the file and displays it on the screen.             *
50 //*****
```



```
51
52 void showContents(fstream &file)
53 {
54     string line;
55
56     while (file >> line)
57     {
58         cout << line << endl;
59     }
60 }
```

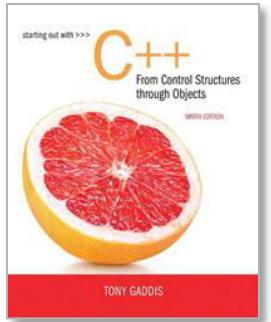
Program Output

File opened successfully.
Now reading data from the file.

Jones
Smith
Willis
Davis

Done.





12.4

More Detailed Error Testing



More Detailed Error Testing

- Can examine error state bits to determine stream status
- Bits tested/cleared by stream member functions

<code>ios::eofbit</code>	set when end of file detected
<code>ios::failbit</code>	set when operation failed
<code>ios::hardfail</code>	set when error occurred and no recovery
<code>ios::badbit</code>	set when invalid operation attempted
<code>ios::goodbit</code>	set when no other bits are set



Member Functions / Flags

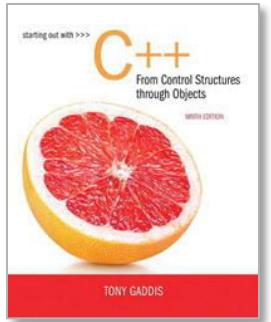
<code>eof()</code>	true if <code>eofbit</code> set , false otherwise
<code>fail()</code>	true if <code>failbit</code> or <code>hardfail</code> set , false otherwise
<code>bad()</code>	true if <code>badbit</code> set , false otherwise
<code>good()</code>	true if <code>goodbit</code> set , false otherwise
<code>clear()</code>	clear all flags (no arguments) , or clear a specific flag



From Program 12-6

```
68 void showState(fstream &file)
69 {
70     cout << "File Status:\n";
71     cout << "  eof bit: " << file.eof() << endl;
72     cout << "  fail bit: " << file.fail() << endl;
73     cout << "  bad bit: " << file.bad() << endl;
74     cout << "  good bit: " << file.good() << endl;
75     file.clear(); // Clear any bad bits
76 }
```





12.5

Member Functions for Reading and Writing Files



Member Functions for Reading and Writing Files

- Functions that may be used for input with whitespace, to perform single character I/O, or to return to the beginning of an input file
- Member functions:

getline: reads input including whitespace

get: reads a single character

put: writes a single character



The getline Function

- Three arguments:
 - Name of a file stream object
 - Name of a string object
 - Delimiter character of your choice
 - Examples, using the file stream object `myFile`, and the string objects `name` and `address`:
`getline(myFile, name);`
`getline(myFile, address, '\t');`
- If left out, '`\n`' is default for third argument



Program 12-8

```
1 // This program uses the getline function to read a line of
2 // data from the file.
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 using namespace std;
7
8 int main()
9 {
10     string input;      // To hold file input
11     fstream nameFile; // File stream object
12
13     // Open the file in input mode.
14     nameFile.open("murphy.txt", ios::in);
15
16     // If the file was successfully opened, continue.
17     if (nameFile)
18     {
19         // Read an item from the file.
20         getline(nameFile, input);
21     }
```



```
22      // While the last read operation
23      // was successful, continue.
24      while (nameFile)
25      {
26          // Display the last item read.
27          cout << input << endl;
28
29          // Read the next item.
30          getline(nameFile, input);
31      }
32
33      // Close the file.
34      nameFile.close();
35  }
36 else
37 {
38     cout << "ERROR: Cannot open file.\n";
39 }
40 return 0;
41 }
```

Program Output

Jayne Murphy
47 Jones Circle
Almond, NC 28702



Single Character I/O

- get: read a single character from a file

```
char letterGrade;
```

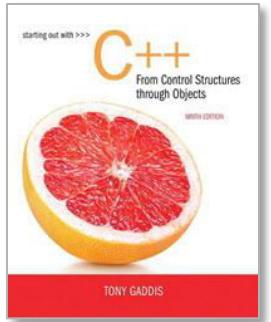
```
gradeFile.get(letterGrade);
```

Will read any character, including whitespace

- put: write a single character to a file

```
reportFile.put(letterGrade);
```





12.6

Working with Multiple Files



Working with Multiple Files

- Can have more than one file open at a time in a program
- Files may be open for input or output
- Need to define file stream object for each file



Program 12-12

```
1 // This program demonstrates reading from one file and writing
2 // to a second file.
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 #include <cctype> // Needed for the toupper function.
7 using namespace std;
8
9 int main()
10 {
11     string fileName;      // To hold the file name
12     char ch;              // To hold a character
13     ifstream inFile;      // Input file
14
15     // Open a file for output.
16     ofstream outFile("out.txt");
17
18     // Get the input file name.
19     cout << "Enter a file name: ";
20     cin >> fileName;
21
22     // Open the file for input.
23     file.open(name, ios::in);
24
25     // If the input file opened successfully, continue.
```



```
26     if (inFile)
27     {
28         // Read a char from file 1.
29         inFile.get(ch);
30
31         // While the last read operation was
32         // successful, continue.
33         while (inFile)
34         {
35             // Write uppercase char to file 2.
36             outFile.put(toupper(ch));
37
38             // Read another char from file 1.
39             inFile.get(ch);
40         }
41
42         // Close the two files.
43         inFile.close();
44         outFile.close();
45         cout << "File conversion done.\n";
46     }
47 else
48     cout << "Cannot open " << fileName << endl;
49 return 0;
50 }
```



Program Screen Output with Example Input Shown in Bold

Enter a file name: **hownow.txt [Enter]**

File conversion done.

Contents of hownow.txt

how now brown cow.

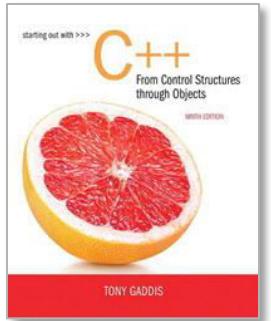
How Now?

Resulting Contents of out.txt

HOW NOW BROWN COW.

HOW NOW?





12.7

Binary Files



Binary Files

- Binary file contains unformatted, non-ASCII data
- Indicate by using binary flag on open:

```
inFile.open("nums.dat", ios::in |  
ios::binary);
```



Binary Files

- Use `read` and `write` instead of `<<`, `>>`

```
char ch;  
// read in a letter from file  
inFile.read(&ch, sizeof(ch));
```

address of where to put
the data being read in.
The `read` function expects
to read `chars`

how many bytes to
read from the file

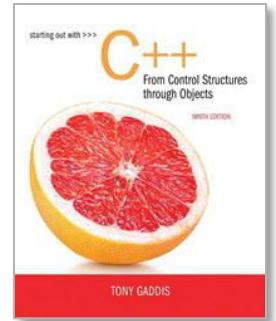
```
// send a character to a file  
outFile.write(&ch, sizeof(ch));
```

Binary Files

- To read, write non-character data, must use a typecast operator to treat the address of the data as a character address

```
int num;  
// read in a binary number from a file  
inFile.read(reinterpret_cast<char *>&num,  
            // treat the address of num as  
            // the address of a char           sizeof(num));  
  
// send a binary value to a file  
outf.write(reinterpret_cast<char *>&num,  
            sizeof(num));
```





12.8

Creating Records with Structures



Creating Records with Structures

- Orange icon: Can write structures to, read structures from files
- Orange icon: To work with structures and files,
 - Orange icon: use `ios::binary` file flag upon open
 - Orange icon: use read, write member functions

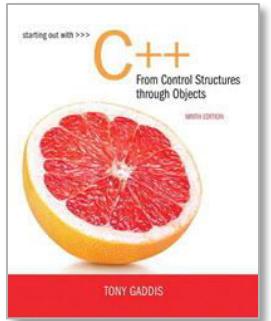


Creating Records with Structures

```
struct TestScore
{
    int studentId;
    double score;
    char grade;
};

TestScore oneTest;
...
// write out oneTest to a file
gradeFile.write(reinterpret_cast<char *>
(&oneTest), sizeof(oneTest));
```





12.9

Random-Access Files



Random-Access Files

- Sequential access: start at beginning of file and go through data in file, in order, to end
 - to access 100th entry in file, go through 99 preceding entries first
- Random access: access data in a file in any order
 - can access 100th entry directly



Random Access Member Functions

- `seekg (seek get)`: used with files open for input
- `seekp (seek put)`: used with files open for output
- Used to go to a specific position in a file



Random Access Member Functions

- seekg, seekp arguments:
 - offset: number of bytes, as a long
 - mode flag: starting point to compute offset

● Examples:

```
inData.seekg(25L, ios::beg);  
// set read position at 26th byte  
// from beginning of file  
outData.seekp(-10L, ios::cur);  
// set write position 10 bytes  
// before current position
```



Important Note on Random Access

- If `eof` is true, it must be cleared before `seekg` or `seekp`:

```
gradeFile.clear();  
gradeFile.seekg(0L, ios::beg);  
// go to the beginning of the file
```



Random Access Information

- orange tellg member function: return current byte position in input file

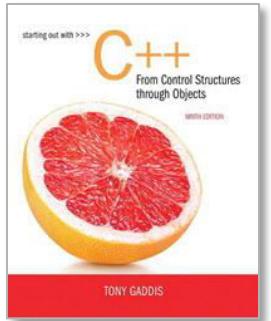
```
long int whereAmI;
```

```
whereAmI = inData.tellg();
```

- orange tellp member function: return current byte position in output file

```
whereAmI = outData.tellp();
```





12.10

Opening a File for Both Input and Output



Opening a File for Both Input and Output

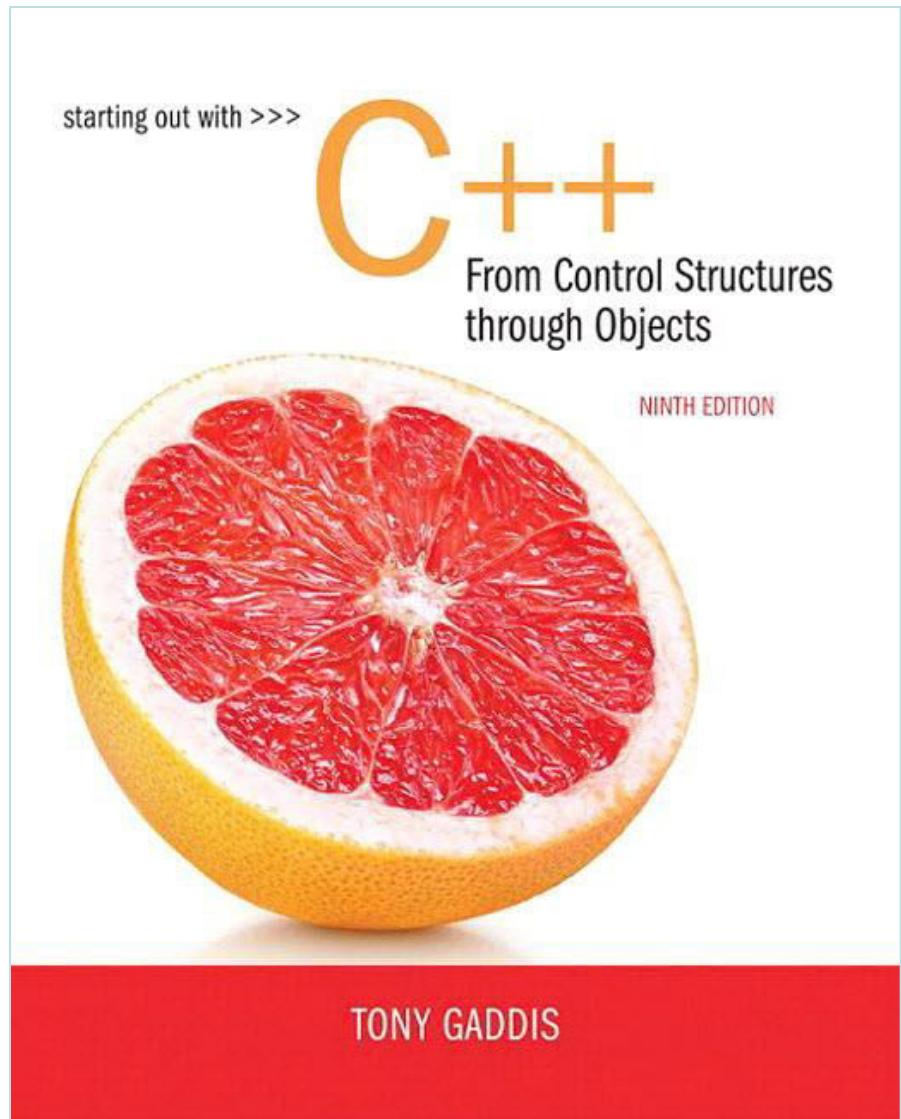
- File can be open for input and output simultaneously
- Supports updating a file:
 - read data from file into memory
 - update data
 - write data back to file
- Use `fstream` for file object definition:

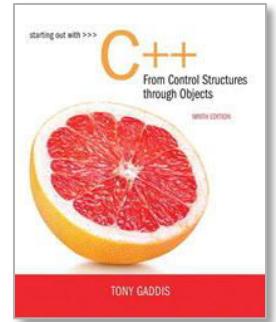
```
fstream gradeList ("grades.dat",
                    ios::in | ios::out);
```
- Can also use `ios::binary` flag for binary data



Chapter 13:

Introduction to Classes





13.1

Procedural and Object-Oriented Programming



Procedural and Object-Oriented Programming

- Procedural programming focuses on the process/actions that occur in a program
- Object-Oriented programming is based on the data and the functions that operate on it. Objects are instances of ADTs that represent the data and its functions



Limitations of Procedural Programming

- If the data structures change, many functions must also be changed
- Programs that are based on complex function hierarchies are:
 - difficult to understand and maintain
 - difficult to modify and extend
 - easy to break



Object-Oriented Programming Terminology

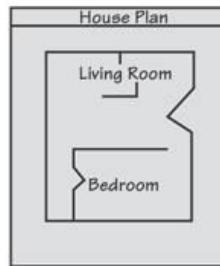
- class: like a struct (allows bundling of related variables), but variables and functions in the class can have different properties than in a struct
- object: an instance of a class, in the same way that a variable can be an instance of a struct



Classes and Objects

- A Class is like a blueprint and objects are like houses built from the blueprint

Blueprint that describes a house.



Instances of the house described by the blueprint.



Object-Oriented Programming Terminology

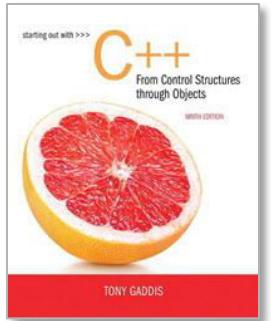
- attributes: members of a class
- methods or behaviors: member functions of a class



More on Objects

- data hiding: restricting access to certain members of an object
- public interface: members of an object that are available outside of the object. This allows the object to provide access to some data and functions without sharing its internal details and design, and provides some protection from data corruption





13.2

Introduction to Classes



Introduction to Classes

- Objects are created from a class
- Format:

```
class ClassName
{
    declaration;
    declaration;
}
```



Class Example

```
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
};
```



Access Specifiers

- Used to control access to members of the class
-  public: can be accessed by functions outside of the class
-  private: can only be called by or accessed by functions that are members of the class



Class Example

```
class Rectangle
{
    private:
        double width;
        double length;
public:
    void setWidth(double);
    void setLength(double);
    double getWidth() const;
    double getLength() const;
    double getArea() const;
};
```

The code defines a class `Rectangle` with private data members `width` and `length`, and public member functions `setWidth`, `setLength`, `getWidth`, `getLength`, and `getArea`. The private members are highlighted by an orange box and labeled "Private Members". The public members are highlighted by an orange box and labeled "Public Members".



More on Access Specifiers

- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is private



Using `const` With Member Functions

- `const` appearing after the parentheses in a member function declaration specifies that the function will not change any data in the calling object.

```
double getWidth() const;  
double getLength() const;  
double getArea() const;
```



Defining a Member Function

- When defining a member function:
 - Put prototype in class declaration
 - Define function using class name and scope resolution operator (::)

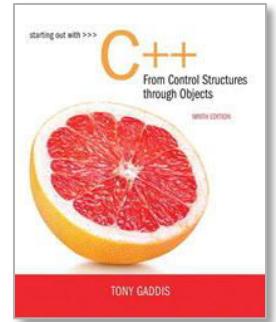
```
int Rectangle::setWidth(double w)
{
    width = w;
}
```



Accessors and Mutators

- Mutator: a member function that stores a value in a private member variable, or changes its value in some way
- Accessor: function that retrieves a value from a private member variable.
Accessors do not change an object's data, so they should be marked `const`.





13.3

Defining an Instance of a Class



Defining an Instance of a Class

- An object is an instance of a class
- Defined like structure variables:

```
Rectangle r;
```
- Access members using dot operator:

```
r.setWidth(5.2);  
cout << r.getWidth();
```
- Compiler error if attempt to access private member using dot operator



Program 13-1

```
1 // This program demonstrates a simple class.  
2 #include <iostream>  
3 using namespace std;  
4  
5 // Rectangle class declaration.  
6 class Rectangle  
7 {  
8     private:  
9         double width;  
10        double length;  
11    public:  
12        void setWidth(double);  
13        void setLength(double);  
14        double getWidth() const;  
15        double getLength() const;  
16        double getArea() const;  
17    };  
18  
19 //*****  
20 // setWidth assigns a value to the width member.  *  
21 //*****  
22  
23 void Rectangle::setWidth(double w)  
24 {  
25     width = w;  
26 }  
27  
28 //*****  
29 // setLength assigns a value to the length member. *  
30 //*****  
31
```



Program 13-1 (Continued)

```
32 void Rectangle::setLength(double len)
33 {
34     length = len;
35 }
36
37 //*****
38 // getWidth returns the value in the width member. *
39 //*****
40
41 double Rectangle::getWidth() const
42 {
43     return width;
44 }
45
46 //*****
47 // getLength returns the value in the length member. *
48 //*****
49
50 double Rectangle::getLength() const
51 {
52     return length;
53 }
54
```



Program 13-1 (Continued)

```
55 //*****
56 // getArea returns the product of width times length. *
57 //*****
58
59 double Rectangle::getArea() const
60 {
61     return width * length;
62 }
63
64 //*****
65 // Function main
66 //*****
67
68 int main()
69 {
70     Rectangle box;      // Define an instance of the Rectangle class
71     double rectWidth;   // Local variable for width
72     double rectLength;  // Local variable for length
73
74     // Get the rectangle's width and length from the user.
75     cout << "This program will calculate the area of a\n";
76     cout << "rectangle. What is the width? ";
77     cin >> rectWidth;
78     cout << "What is the length? ";
79     cin >> rectLength;
80
81     // Store the width and length of the rectangle
82     // in the box object.
83     box.setWidth(rectWidth);
84     box.setLength(rectLength);
```



Program 13-1 (Continued)

```
85
86     // Display the rectangle's data.
87     cout << "Here is the rectangle's data:\n";
88     cout << "Width: " << box.getWidth() << endl;
89     cout << "Length: " << box.getLength() << endl;
90     cout << "Area: " << box.getArea() << endl;
91     return 0;
92 }
```

Program Output

This program will calculate the area of a rectangle. What is the width? **10 [Enter]**

What is the length? **5 [Enter]**

Here is the rectangle's data:

Width: 10

Length: 5

Area: 50



Avoiding Stale Data

- Some data is the result of a calculation.
- In the Rectangle class the area of a rectangle is calculated.
 - $\text{length} \times \text{width}$
- If we were to use an area variable here in the Rectangle class, its value would be dependent on the length and the width.
- If we change length or width without updating area, then area would become *stale*.
- To avoid stale data, it is best to calculate the value of that data within a member function rather than store it in a variable.



Pointer to an Object

- Orange icon: Can define a pointer to an object:

```
Rectangle *rPtr = nullptr;
```

- Orange icon: Can access public members via pointer:

```
rPtr = &otherRectangle;  
rPtr->setLength(12.5);  
cout << rPtr->getLength() << endl;
```

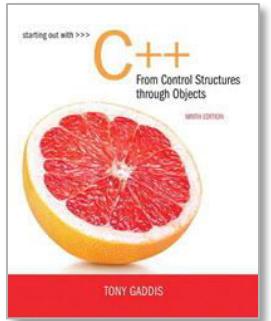


Dynamically Allocating an Object

- We can also use a pointer to dynamically allocate an object.

```
1 // Define a Rectangle pointer.  
2 Rectangle *rectPtr = nullptr;  
3  
4 // Dynamically allocate a Rectangle object.  
5 rectPtr = new Rectangle;  
6  
7 // Store values in the object's width and length.  
8 rectPtr->setWidth(10.0);  
9 rectPtr->setLength(15.0);  
10  
11 // Delete the object from memory.  
12 delete rectPtr;  
13 rectPtr = nullptr;
```





13.4

Why Have Private Members?

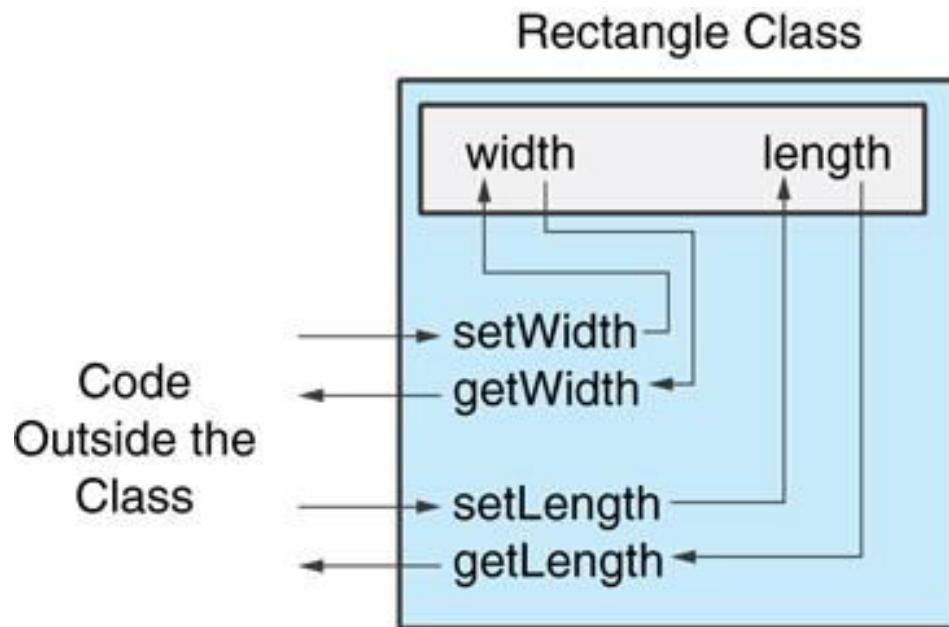


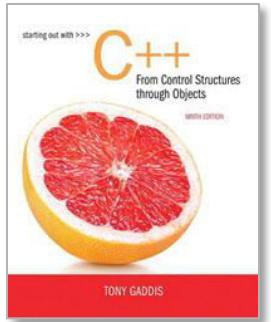
Why Have Private Members?

- ➊ Making data members private provides data protection
- ➋ Data can be accessed only through public functions
- ➌ Public functions define the class's public interface



Code outside the class must use the class's public member functions to interact with the object.





13.5

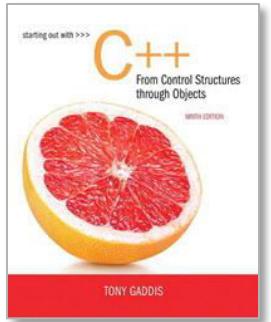
Separating Specification from Implementation



Separating Specification from Implementation

- Place class declaration in a header file that serves as the class specification file. Name the file *ClassName.h*, for example, *Rectangle.h*
- Place member function definitions in *ClassName.cpp*, for example, *Rectangle.cpp* File should `#include` the class specification file
- Programs that use the class must `#include` the class specification file, and be compiled and linked with the member function definitions





13.6

Inline Member Functions



Inline Member Functions

- Member functions can be defined
 - inline: in class declaration
 - after the class declaration
- Inline appropriate for short function bodies:

```
int getWidth() const  
{ return width; }
```



Rectangle Class with Inline Member Functions

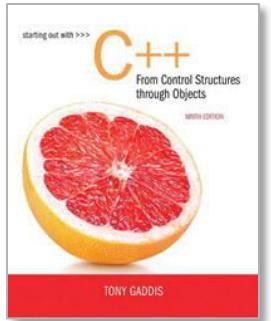
```
1 // Specification file for the Rectangle class
2 // This version uses some inline member functions.
3 #ifndef RECTANGLE_H
4 #define RECTANGLE_H
5
6 class Rectangle
7 {
8     private:
9         double width;
10        double length;
11    public:
12        void setWidth(double);
13        void setLength(double);
14
15        double getWidth() const
16            { return width; }
17
18        double getLength() const
19            { return length; }
20
21        double getArea() const
22            { return width * length; }
23    };
24#endif
```



Tradeoffs – Inline vs. Regular Member Functions

- Orange Regular functions – when called, compiler stores return address of call, allocates memory for local variables, etc.
- Orange Code for an inline function is copied into program in place of call – larger executable program, but no function call overhead, hence faster execution





13.7

Constructors



Constructors

- Member function that is automatically called when an object is created
- Purpose is to construct an object
- Constructor function name is class name
- Has no return type



Contents of Rectangle.h (Version 3)

```
1 // Specification file for the Rectangle class
2 // This version has a constructor.
3 #ifndef RECTANGLE_H
4 #define RECTANGLE_H
5
6 class Rectangle
7 {
8     private:
9         double width;
10        double length;
11    public:
12        Rectangle();           // Constructor
13        void setWidth(double);
14        void setLength(double);
15
16        double getWidth() const
17            { return width; }
18
19        double getLength() const
20            { return length; }
21
22        double getArea() const
23            { return width * length; }
24    };
25 #endif
```



Contents of Rectangle.cpp (Version 3)

```
1 // Implementation file for the Rectangle class.  
2 // This version has a constructor.  
3 #include "Rectangle.h"    // Needed for the Rectangle class  
4 #include <iostream>        // Needed for cout  
5 #include <cstdlib>         // Needed for the exit function  
6 using namespace std;  
7  
8 //*****  
9 // The constructor initializes width and length to 0.0.      *  
10 //*****  
11  
12 Rectangle::Rectangle()  
13 {  
14     width = 0.0;  
15     length = 0.0;  
16 }
```

Continues...



Contents of Rectangle.cpp Version3

```
17 //*****
18 // setWidth sets the value of the member variable width. *
19 //*****
20
21 void Rectangle::setWidth(double w)
22 {
23     if (w >= 0)
24         width = w;
25     else
26     {
27         cout << "Invalid width\n";
28         exit(EXIT_FAILURE);
29     }
30 }
31
32 //*****
33 // setLength sets the value of the member variable length. *
34 //*****
35
36 void Rectangle::setLength(double len)
37 {
38     if (len >= 0)
39         length = len;
40     else
41     {
42         cout << "Invalid length\n";
43         exit(EXIT_FAILURE);
44     }
45 }
46 }
```

(continued)



Program 13-7

```
1 // This program uses the Rectangle class's constructor.  
2 #include <iostream>  
3 #include "Rectangle.h" // Needed for Rectangle class  
4 using namespace std;  
5  
6 int main()  
7 {  
8     Rectangle box;      // Define an instance of the Rectangle class  
9  
10    // Display the rectangle's data.  
11    cout << "Here is the rectangle's data:\n";  
12    cout << "Width: " << box.getWidth() << endl;  
13    cout << "Length: " << box.getLength() << endl;  
14    cout << "Area: " << box.getArea() << endl;  
15    return 0;  
16 }
```

Program Output

Here is the rectangle's data:
Width: 0
Length: 0
Area: 0



In-Place Initialization

- If you are using C++11 or later, you can initialize a member variable in its declaration statement, just as you can with a regular variable.
- This is known as in-place initialization. Here is an example:

```
class Rectangle
{
private:
    double width = 0.0;
    double length = 0.0;
public:
    Public member functions appear here...
};
```

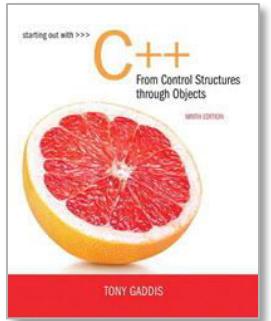


Default Constructors

- A default constructor is a constructor that takes no arguments.
- If you write a class with no constructor at all, C++ will write a default constructor for you, one that does nothing.
- A simple instantiation of a class (with no arguments) calls the default constructor:

```
Rectangle r;
```





13.8

Passing Arguments to Constructors



Passing Arguments to Constructors

- To create a constructor that takes arguments:
 - indicate parameters in prototype:

```
Rectangle(double, double);
```

- Use parameters in the definition:

```
Rectangle::Rectangle(double w, double  
len)  
{  
    width = w;  
    length = len;  
}
```



Passing Arguments to Constructors

- Orange You can pass arguments to the constructor when you create an object:

```
Rectangle r(10, 5);
```



More About Default Constructors

- If all of a constructor's parameters have default arguments, then it is a default constructor. For example:

```
Rectangle (double = 0, double = 0);
```

- Creating an object and passing no arguments will cause this constructor to execute:

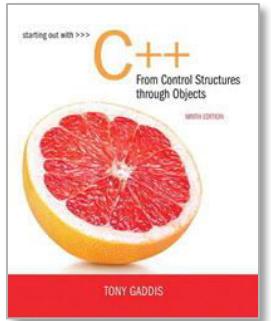
```
Rectangle r;
```



Classes with No Default Constructor

- When all of a class's constructors require arguments, then the class has NO default constructor.
- When this is the case, you must pass the required arguments to the constructor when creating an object.





13.9

Destructors



Destructors

- Member function automatically called when an object is destroyed
- Destructor name is `~classname`, e.g.,
`~Rectangle`
- Has no return type; takes no arguments
- Only one destructor per class, *i.e.*, it cannot be overloaded
- If constructor allocates dynamic memory, destructor should release it



Contents of InventoryItem.h (Version 1)

```
1 // Specification file for the InventoryItem class.  
2 #ifndef INVENTORYITEM_H  
3 #define INVENTORYITEM_H  
4 #include <cstring> // Needed for strlen and strcpy  
5  
6 // InventoryItem class declaration.  
7 class InventoryItem  
8 {  
9 private:  
10    char *description; // The item description  
11    double cost;       // The item cost  
12    int units;         // Number of units on hand
```



Contents of InventoryItem.h Version1

```
13 public:  
14     // Constructor  
15     InventoryItem(char *desc, double c, int u)  
16     { // Allocate just enough memory for the description.  
17         description = new char [strlen(desc) + 1];  
18  
19         // Copy the description to the allocated memory.  
20         strcpy(description, desc);  
21  
22         // Assign values to cost and units.  
23         cost = c;  
24         units = u;}  
25  
26     // Destructor  
27     ~InventoryItem()  
28     { delete [] description; }  
29  
30     const char *getDescription() const  
31     { return description; }  
32  
33     double getCost() const  
34     { return cost; }  
35  
36     int getUnits() const  
37     { return units; }  
38 };  
39 #endif
```

(continued)



Program 13-12

```
1 // This program demonstrates a class with a destructor.  
2 #include <iostream>  
3 #include "ContactInfo.h"  
4 using namespace std;  
5  
6 int main()  
7 {  
8     // Define a ContactInfo object with the following data:  
9     // Name: Kristen Lee Phone Number: 555-2021  
10    ContactInfo entry("Kristen Lee", "555-2021");  
11  
12    // Display the object's data.  
13    cout << "Name: " << entry.getName() << endl;  
14    cout << "Phone Number: " << entry.getPhoneNumber() << endl;  
15    return 0;  
16 }
```

Program Output

Name: Kristen Lee
Phone Number: 555-2021



Constructors, Destructors, and Dynamically Allocated Objects

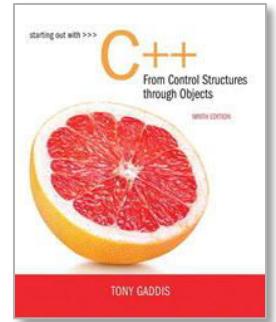
- When an object is dynamically allocated with the new operator, its constructor executes:

```
Rectangle *r = new Rectangle(10, 20);
```

- When the object is destroyed, its destructor executes:

```
delete r;
```





13.10

Overloading Constructors



Overloading Constructors

- A class can have more than one constructor
- Overloaded constructors in a class must have different parameter lists:

```
Rectangle();
```

```
Rectangle(double);
```

```
Rectangle(double, double);
```



```
1 // This class has overloaded constructors.  
2 #ifndef INVENTORYITEM_H  
3 #define INVENTORYITEM_H  
4 #include <string>  
5 using namespace std;  
6  
7 class InventoryItem  
8 {  
9 private:  
10     string description; // The item description  
11     double cost;        // The item cost  
12     int units;          // Number of units on hand  
13 public:  
14     // Constructor #1  
15     InventoryItem()  
16     { // Initialize description, cost, and units.  
17         description = "";  
18         cost = 0.0;  
19         units = 0; }  
20  
21     // Constructor #2  
22     InventoryItem(string desc)  
23     { // Assign the value to description.  
24         description = desc;  
25  
26         // Initialize cost and units.  
27         cost = 0.0;  
28         units = 0; }
```

Continues...



```
29
30     // Constructor #3
31     InventoryItem(string desc, double c, int u)
32         { // Assign values to description, cost, and units.
33             description = desc;
34             cost = c;
35             units = u; }
36
37     // Mutator functions
38     void setDescription(string d)
39         { description = d; }
40
41     void setCost(double c)
42         { cost = c; }
43
44     void setUnits(int u)
45         { units = u; }
46
47     // Accessor functions
48     string getDescription() const
49         { return description; }
50
51     double getCost() const
52         { return cost; }
53
54     int getUnits() const
55         { return units; }
56 };
57 #endif
```



Constructor Delegation

- Sometimes a class will have multiple constructors that perform a similar set of steps. For example, look at the following Contact class:

```
class Contact
{
private:
    string name;
    string email;
    string phone;
public:
    // Constructor #1 (default)
    Contact()
    { name = "";
        email = "";
        phone = "";
    }

    // Constructor #2
    Contact(string n, string e, string p)
    { name = n;
        email = e;
        phone = p;
    }

    Other member functions follow...
};
```



Constructor Delegation

- Both constructors perform a similar operation: They assign values to the name, email, and phone member variables.
- The default constructor assigns empty strings to the members, and the parameterized constructor assigns specified values to the members.

```
class Contact
{
private:
    string name;
    string email;
    string phone;
public:
    // Constructor #1 (default)
    Contact()
    { name = "";
        email = "";
        phone = "";
    }

    // Constructor #2
    Contact(string n, string e, string p)
    { name = n;
        email = e;
        phone = p;
    }

    Other member functions follow...
};
```



Constructor Delegation

- In C++ 11, it is possible for one constructor to call another constructor in the same class.
- This is known as *constructor delegation*.

```
class Contact
{
private:
    string name;
    string email;
    string phone;
public:
    // Constructor #1 (default)
    Contact() : Contact("", "", "")
    { }

    // Constructor #2
    Contact(string n, string e, string p)
    { name = n;
        email = e;
        phone = p;
    }

    Other member functions follow...
};
```



Only One Default Constructor and One Destructor

- Do not provide more than one default constructor for a class: one that takes no arguments and one that has default arguments for all parameters

```
Square();
```

```
Square(int = 0); // will not compile
```

- Since a destructor takes no arguments, there can only be one destructor for a class



Member Function Overloading

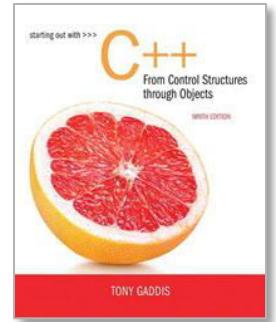
- Orange Non-constructor member functions can also be overloaded:

```
void setCost (double) ;
```

```
void setCost (char *) ;
```

- Orange Must have unique parameter lists as for constructors





3.11

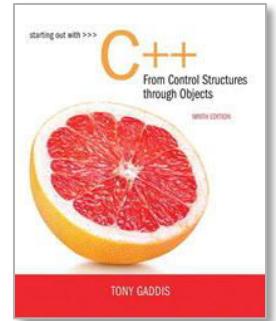
Using Private Member Functions



Using Private Member Functions

- A private member function can only be called by another member function
- It is used for internal processing by the class, not for use outside of the class
- See the `createDescription` function in **ContactInfo.h** (Version 2)





13.12

Arrays of Objects



Arrays of Objects

- Objects can be the elements of an array:

```
InventoryItem inventory[40];
```

- Default constructor for object is used when array is defined



Arrays of Objects

- Must use initializer list to invoke constructor that takes arguments:

```
InventoryItem inventory[3] =  
{ "Hammer", "Wrench", "Pliers" };
```



Arrays of Objects

- If the constructor requires more than one argument, the initializer must take the form of a function call:

```
InventoryItem inventory[ 3 ] = { InventoryItem("Hammer", 6.95, 12),  
                               InventoryItem("Wrench", 8.75, 20),  
                               InventoryItem("Pliers", 3.75, 10) };
```



Arrays of Objects

- It isn't necessary to call the same constructor for each object in an array:

```
InventoryItem inventory[3] = { "Hammer",  
                             InventoryItem("Wrench", 8.75, 20),  
                             "Pliers" };
```



Accessing Objects in an Array

- Objects in an array are referenced using subscripts
- Member functions are referenced using dot notation:

```
inventory[2].setUnits(30);  
cout << inventory[2].getUnits();
```



Program 13-14

```
1 // This program demonstrates an array of class objects.  
2 #include <iostream>  
3 #include <iomanip>  
4 #include "InventoryItem.h"  
5 using namespace std;  
6  
7 int main()  
8 {  
9     const int NUM_ITEMS = 5;  
10    InventoryItem inventory[NUM_ITEMS] = {  
11        InventoryItem("Hammer", 6.95, 12),  
12        InventoryItem("Wrench", 8.75, 20),  
13        InventoryItem("Pliers", 3.75, 10),  
14        InventoryItem("Ratchet", 7.95, 14),  
15        InventoryItem("Screwdriver", 2.50, 22) };  
16
```



Program 13-14 (Continued)

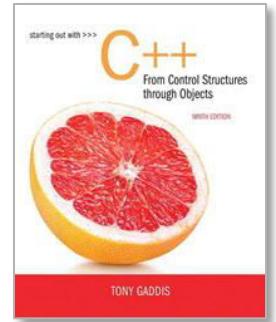
```
17     cout << setw(14) << "Inventory Item"
18             << setw(8) << "Cost" << setw(8)
19             << setw(16) << "Units on Hand\n";
20     cout << "-----\n";
21
22     for (int i = 0; i < NUM_ITEMS; i++)
23     {
24         cout << setw(14) << inventory[i].getDescription();
25         cout << setw(8) << inventory[i].getCost();
26         cout << setw(7) << inventory[i].getUnits() << endl;
27     }
28
29     return 0;
30 }
```

Program Output

Inventory Item	Cost	Units on Hand

Hammer	6.95	12
Wrench	8.75	20
Pliers	3.75	10
Ratchet	7.95	14
Screwdriver	2.5	22





13.16

The Unified Modeling Language



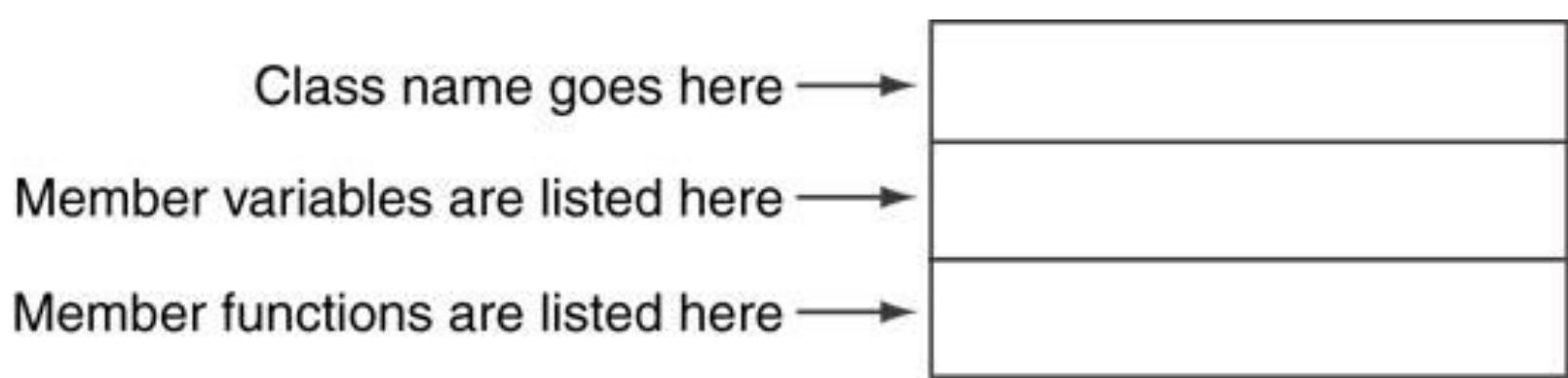
The Unified Modeling Language

- UML stands for *Unified Modeling Language*.
- The UML provides a set of standard diagrams for graphically depicting object-oriented systems



UML Class Diagram

- A UML diagram for a class has three main sections.



Example: A Rectangle Class

Rectangle
width length
setWidth() setLength() getWidth() getLength() getArea()

```
class Rectangle
{
    private:
        double width;
        double length;
    public:
        bool setWidth(double);
        bool setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
};
```

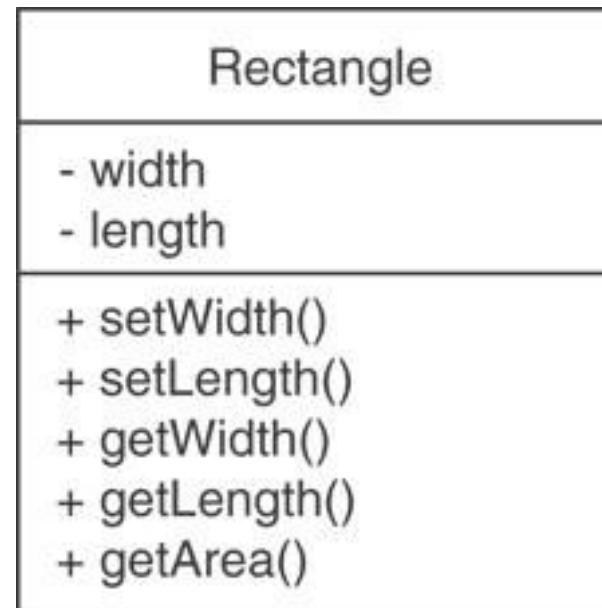


UML Access Specification Notation

- In UML you indicate a private member with a minus (-) and a public member with a plus(+).

These member variables are private.

These member functions are public.



UML Data Type Notation

- To indicate the data type of a member variable, place a colon followed by the name of the data type after the name of the variable.
 - width : double
 - length : double



UML Parameter Type Notation

- To indicate the data type of a function's parameter variable, place a colon followed by the name of the data type after the name of the variable.
 - + `setwidth(w : double)`



UML Function Return Type Notation

- To indicate the data type of a function's return value, place a colon followed by the name of the data type after the function's parameter list.

+ setwidth(w : double) : void



The Rectangle Class

Rectangle
<ul style="list-style-type: none">- width : double- length : double
<ul style="list-style-type: none">+ setWidth(w : double) : bool+ setLength(len : double) : bool+ getWidth() : double+ getLength() : double+ getArea() : double



Showing Constructors and Destructors

No return type listed for constructors or destructors

Constructors →

Destructor →

InventoryItem
<ul style="list-style-type: none">- description : char*- cost : double- units : int- createDescription(size : int, value : char*) : void
<ul style="list-style-type: none">+ InventoryItem() :+ InventoryItem(desc : char*) :+ InventoryItem(desc : char*, c : double, u : int) :+ ~InventoryItem() :+ setDescription(d : char*) : void+ setCost(c : double) : void+ setUnits(u : int) : void+ getDescription() : char*+ getCost() : double+ getUnits() : int

