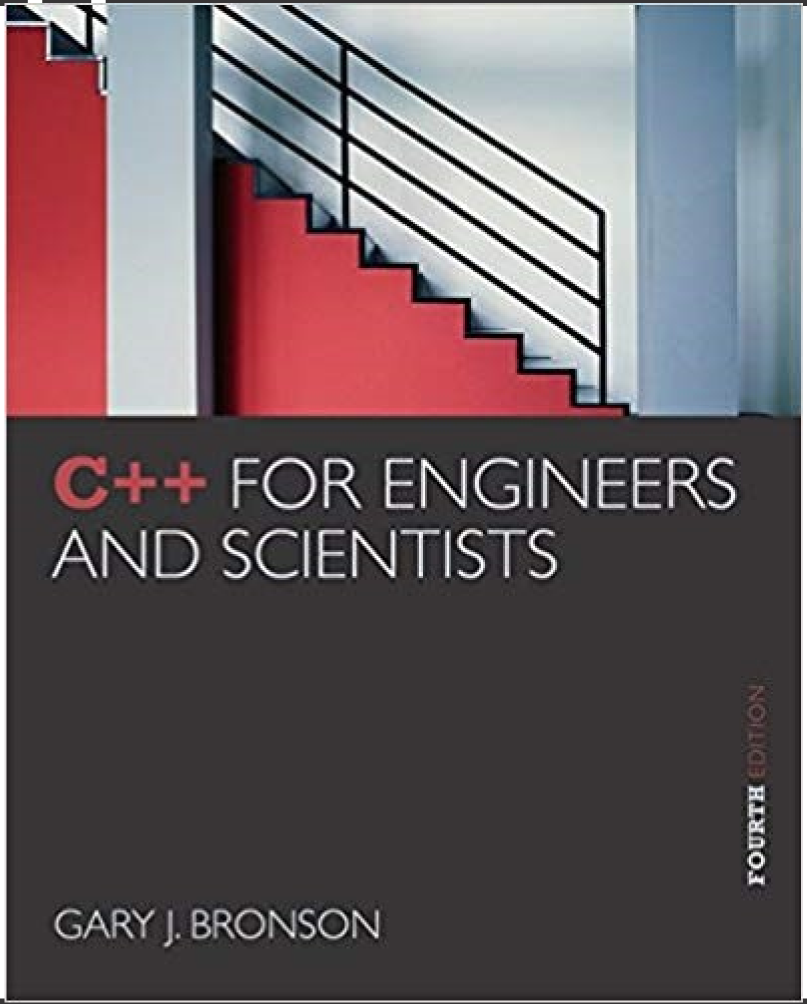
ELEG 1043
Computer Applications in Engineering



Chapter 2: Problem Solving Using C++

C++ FOR ENGINEERS AND SCIENTISTS 2

Acknowledgement

- Some of the slides or images are from various sources. The copyright of those materials belongs to their original owners.

Objectives

In this chapter, you will learn about:

- Modular programs
- Programming style
- Data types
- Arithmetic operations
- Variables and declaration statements
- Common programming errors

Introduction to C++

- **Modular program:** A program consisting of interrelated segments (or **modules)** arranged in a logical and understandable form
  - Easy to develop, correct, and modify
- Modules in C++ can be classes or functions

Introduction to C++ (continued)

- **Function:** Accepts an input, processes the input, and produces an output
  - A function’s processing is encapsulated and hidden within the function

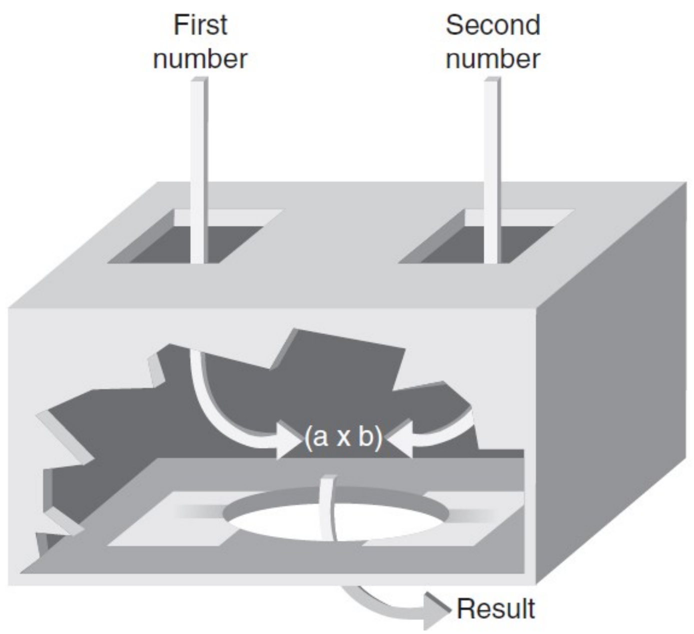


Figure 2.2 A multiplying function

Introduction to C++ (continued)

- **Class:** Contains both data and functions used to manipulate the data
- **Identifier:** A name given to an element of the language, such as a class or function
  - Rules for forming identifier names:
    - First character must be a letter or underscore
    - Only letters, digits, or underscores may follow the initial letter (no blanks allowed)
    - Keywords cannot be used as identifiers
    - Maximum length of an identifier = 1024 characters

Introduction to C++ (continued)

- **Keyword**: A reserved name that represents a built-in object or function of the language

auto	delete	goto	public	this
break	do	if	register	template
case	double	inline	return	typedef
catch	else	int	short	union
char	enum	long	signed	unsigned
class	extern	new	sizeof	virtual
const	float	overload	static	void
continue	for	private	struct	volatile
default	friend	protected	switch	while

Table 2.1: Keywords in C++

Introduction to C++ (continued)

- Examples of valid C++ identifiers:

```
degToRad  intersect   addNums
slope     bessell     multTwo
findMax   density
```

- Examples of invalid C++ identifiers:

`1AB3`	(begins with a number)
`E*6`	(contains a special character)
`while`	(this is a keyword)

Introduction to C++ (continued)

- Function names
  - Require a set of parentheses at the end
  - Can use mixed upper and lower case
  - Should be meaningful, or be a **mnemonic**
- Examples of function names:

```
addNums()   multTwoNums()
```

- Note that C++ is a case-sensitive language!
  - addNums() is different from AddNums()

The main() Function

- Overall structure of a C++ program contains one function named `main()`, called the **driver function**
- All other functions are invoked from `main()`

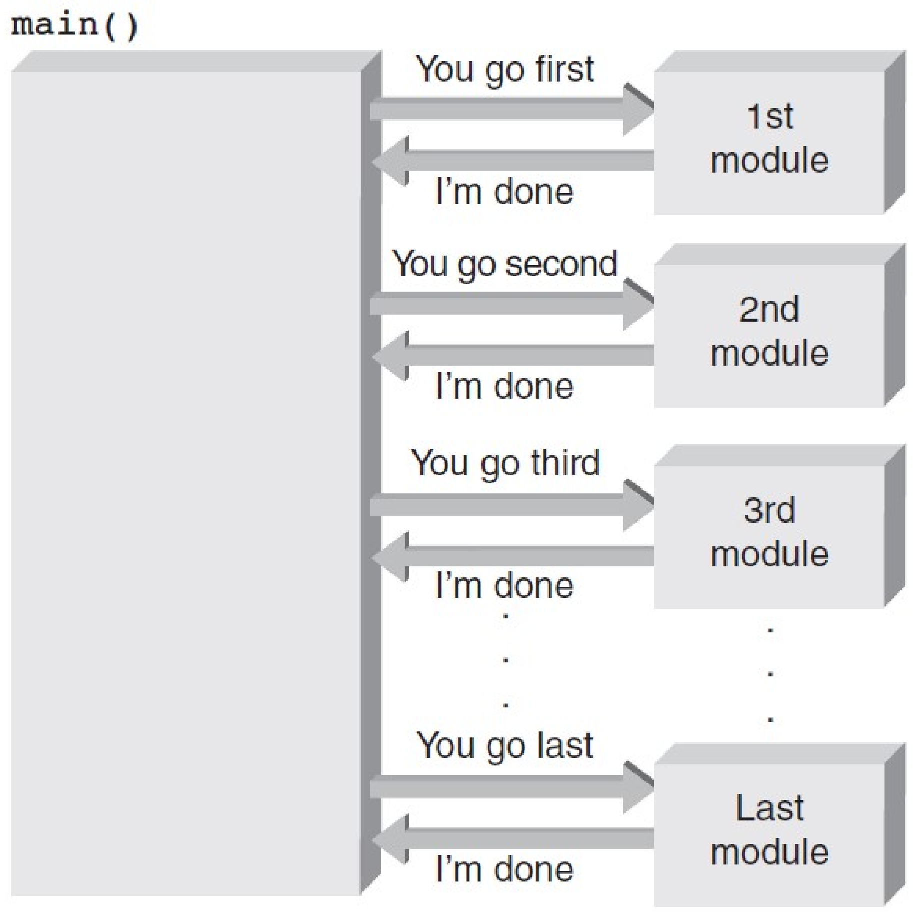


Figure 2.3 The `main()` function directs all other functions.

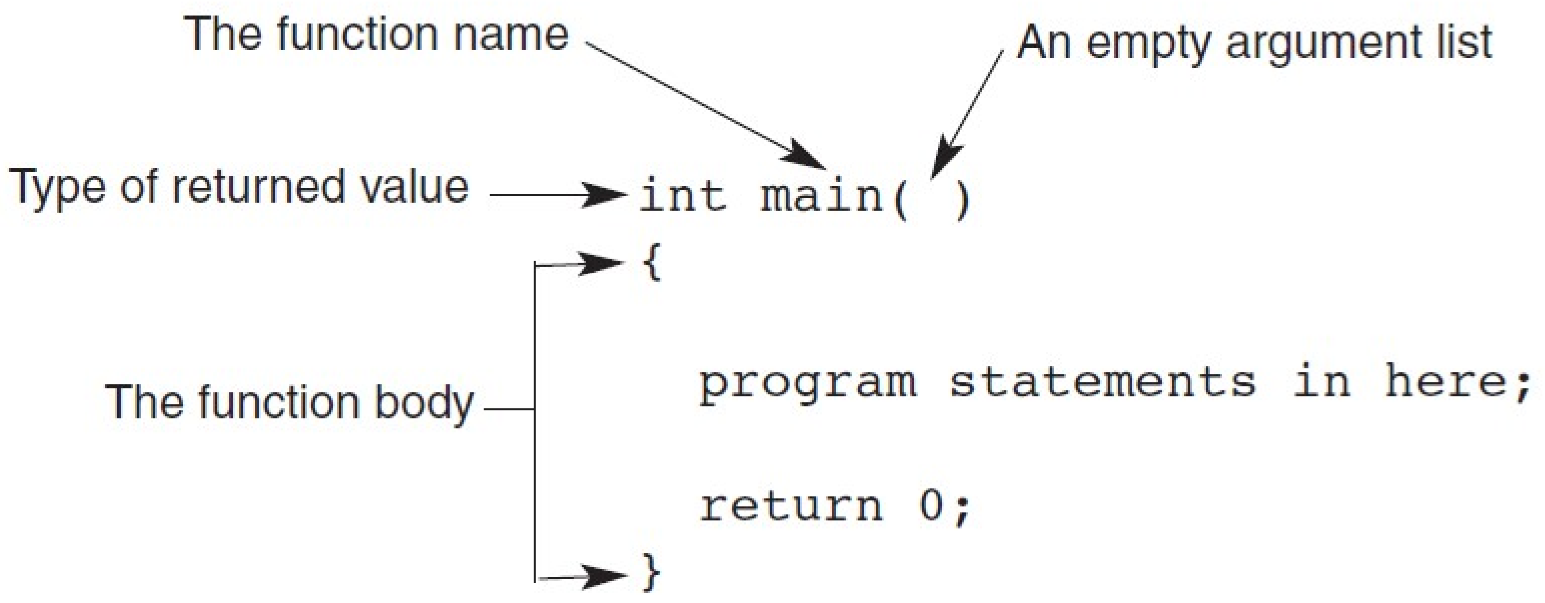
The `main()` Function (continued)

- **Function header line:** First line of a function, which contains:
  - The type of data returned by the function (if any)
  - The name of the function
  - The type of data that must be passed into the function when it is invoked (if any)
- **Arguments:** The data passed into a function
- **Function body:** The statements inside a function
  - enclosed in braces **{ }**

The `main()` Function (continued)

- Each statement inside the function must be terminated with a semicolon ";"
- **`return`:** A keyword causing the appropriate value to be returned from the function
- The statement `return 0` in the `main()` function causes the program to end

The `main()` Function (continued)



```
int main( )
{

  program statements in here;

  return 0;
}
```

Figure 2.4 The structure of a **`main()`** function

Programming Style

- Good style calls for one C++ statement per line
- Opening and closing braces **{ }** for the function body should be on separate lines

```
int add(int a, int b){
Int sum = a + b;


return sum;
}
```

Comments

- **Comments:** Explanatory remarks in the source code added by the programmer
- **Line comment:** Begins with // and continues to the end of the line
    - Example:

```
// this program displays a message
#include <iostream>
using namespace std;

 int main ()
 {
      cout << "Hello there world!"; //displays text
     return 0;
 }
```

Comments (continued)

- **Block comments:** comments that span across two or more lines
  - Begin with `/*` and end with `*/`
  - Example:

```
/* This is a block comment that
spans
across three lines */
```

Data Types

- **Data type:** A set of values and the operations that can be applied to these values
- Two fundamental C++ data groupings:
  - **Class data type** (a class): Created by the programmer
  - **Built-in data type** (primitive type): Part of the C++ compiler

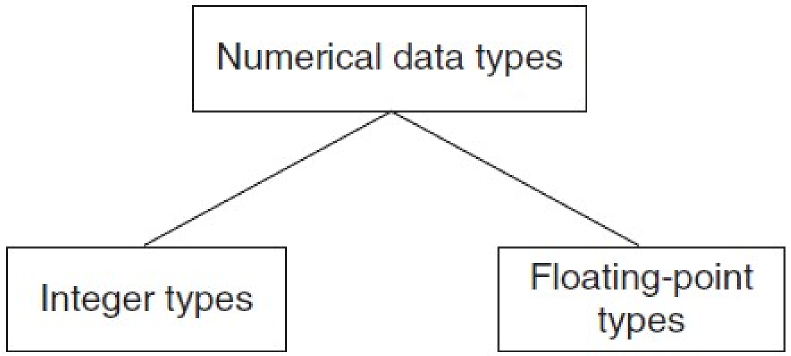
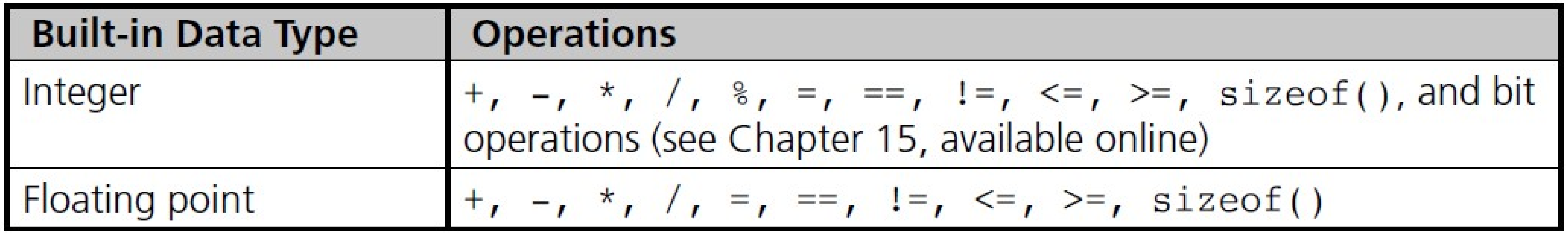


Figure 2.5 Built-in data types

Data Types (continued)



Built-in Data Type	Operations
Integer	`+, -, *, /, %, =, ==, !=, <=, >=, sizeof()`, and bit operations (see Chapter 15, available online)
Floating point	`+, -, *, /, =, ==, !=, <=, >=, sizeof()`

Table 2.2 Built-In Data Type Operations

Data Types (continued)

- **Literal (constant):** An actual value
  - Examples:

```
3.6          //numeric literal
“Hello”      //string literal
```

- **Integer:** A whole number
- C++ has nine built-in integer data types

Integer Data Types

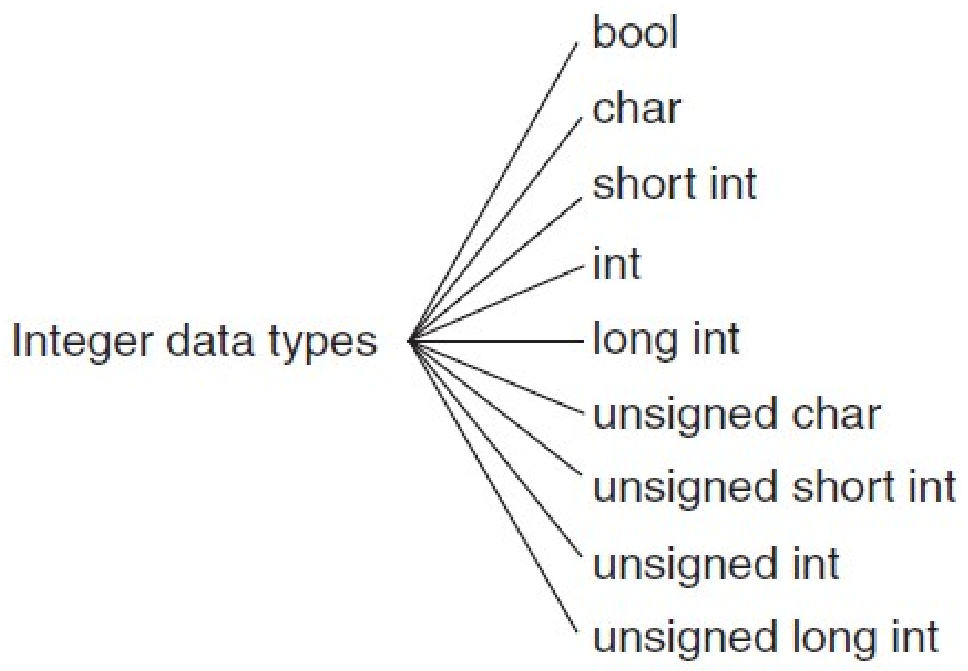


Figure 2.6 C++ integer data types

Integer Data Types (continued)

- `int` data type: Whole numbers (integers), optionally with plus (`+`) or minus (`–`) sign
  - Example: `2, -5`
- `char` data type: Individual character; any letter, digit, or special character enclosed in single quotes
  - Example: `‘A’`

Integer Data Types (continued)

- **Escape character:** The backslash, ****
  - Indicates an escape sequence
- **Escape sequence:** Tells compiler to treat the following characters as special instruction codes
  - \t, \n

```
cout<<“Hello World!\n”;
```

Integer Data Types (continued)

- `bool` data type: Represents Boolean (logical) data
  - Restricted to two values: true or false
  - Useful when a program must examine a condition and take a prescribed course of action, based on whether the condition is true or false

Determining Storage Size

- A unique feature of C++ is that you can see where and how values are stored
  - **`sizeof()`** operator provides the number of bytes used to store values of the data type named in the parenthesis

    int a = 10;

    cout<<sizeof(a);
  - Values returned by **`sizeof()`** are compiler dependent

Signed and Unsigned Data Types

- **Signed data type:** One that permits negative, positive, and zero values
- **Unsigned data type:** Permits only positive and zero values
  - An unsigned data type provides essentially double the range of its signed counterpart

Signed and Unsigned Data Types (continued)

Name of Data Type	Storage Size	Range of Values
`char`	1	256 characters
`bool`	1	true (considered as any positive value) and false (which is a 0)
`short int`	2	-32,768 to +32,767
`unsigned short int`	2	0 to 65,535
`int`	4	-2,147,483,648 to +2,147,483,647
`unsigned int`	4	0 to 4,294,967,295
`long int`	4	-2,147,483,648 to +2,147,483,647
`unsigned long int`	4	0 to 4,294,967,295

Table 2.5 Integer Data Type Storage

Floating-Point Types

- **Floating-point number** (real number): Zero or any positive or negative number containing a decimal point
  - Examples: `+10.625,  5., -6.2`
  - Three floating-point data types in C++:
    - `float` (single precision)
    - `double` (double precision)
    - `long double`

Floating-Point Types (continued)

- `float literal`: Append an `f` or `F` to the number
- `long double literal`: Append an `l` or `L` to the number
  - Examples:

```
9.234       // a double literal
9.234F      // a float literal
9.234L      // a long double literal
```

Arithmetic Operations

- C++ supports addition, subtraction, multiplication, division, and modulus division
- Different data types can be used in the same arithmetic expression
- Arithmetic operators are binary operators
  - **Binary operators:** Require two operands
  - **Unary operator:** Requires only one operand
  - **Negation operator (−):** Reverses the sign of the number

Arithmetic Operations (continued)

Operation	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus division	%

Arithmetic Operations (continued)

Program 2.6

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

int main()
{
  cout << "15.0 plus 2.0 equals "        << (15.0 + 2.0) << endl
       << "15.0 minus 2.0 equals "       << (15.0 - 2.0) << endl
       << "15.0 times 2.0 equals "       << (15.0 * 2.0) << endl
       << "15.0 divided by 2.0 equals "  << (15.0 / 2.0) << endl;

  return 0;
}
```

Integer Division

- Integer division: Yields an integer result
  - Any fractional remainders are dropped (truncated)
  - Example: `15/2` yields `7`
- Modulus (remainder) operator: Returns only the remainder
  - Example: `9 % 4` yields `1`

Operator Precedence and Associativity

- Rules for writing arithmetic expressions:
  - Never place two consecutive binary arithmetic operators side by side
    - 2 +* 3
  - Use parentheses to form groupings
    - Contents within parentheses are evaluated first

    e.g.: 3 * (4 - 1)
  - May nest parentheses within other parentheses
    - Evaluated from innermost to outermost

    e.g.: (3 + (4 - 1) - 2)
  - Use the * operator for multiplication

Operator Precedence and Associativity (continued)

- Expressions with multiple operators are evaluated by precedence of operators:
  - All negations occur first
  - Multiplication, division, and modulus are next, from left to right
  - Addition and subtraction are last, from left to right

e.g. 6 + (-5)*4/2 - 3

Operator Precedence and Associativity (continued)

- Associativity: the order in which operators of the same precedence are evaluated

Operator	Associativity
Unary –	Right to left
* / %	Left to right
+ –	Left to right

Table 2.8 Operator Precedence and Associativity

Variables and Declaration Statements

- **Variable:** All integer, float-point, and other values used in a program are stored and retrieved from the computer's memory
- Each memory location has a unique address

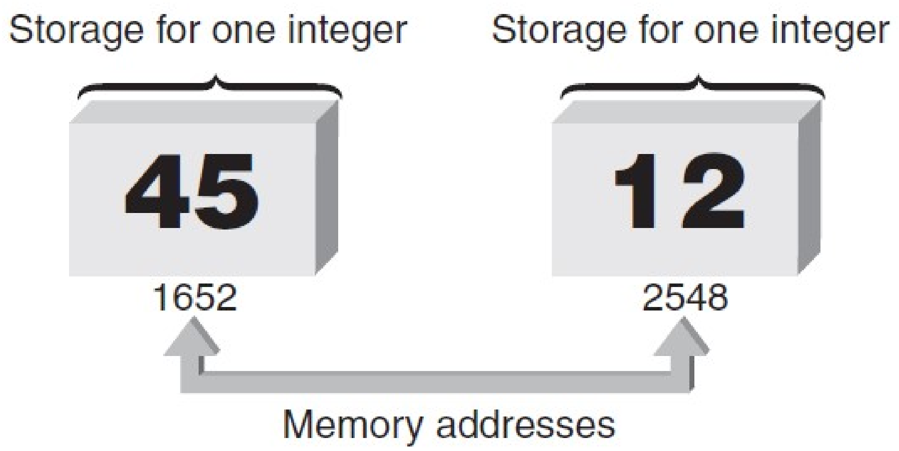


Figure 2.8 Enough storage for two integers

Variables and Declaration Statements (continued)

- **Variable:** Symbolic identifier for a memory address where data can be held
- Use identifier naming rules for variable names

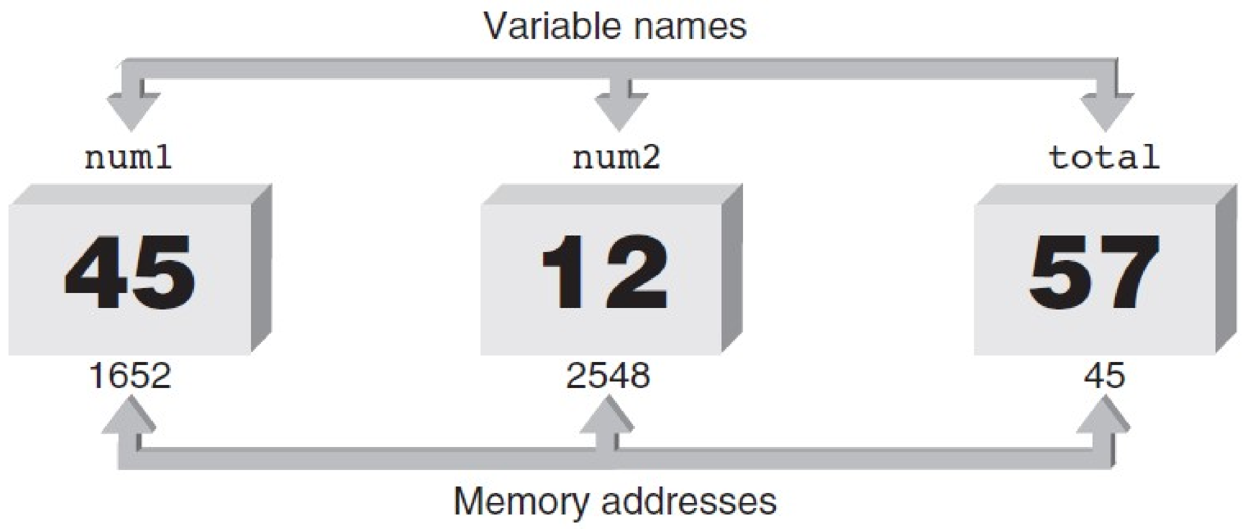


Figure 2.9 Naming storage locations

Variables and Declaration Statements (continued)

- **Assignment statement:** Used to store a value into a variable
- Value of the expression on the right is assigned to the memory location of the variable on the left side
  - Examples:

```
num1 = 45;
num2 = 12;
total = num1 + num2;
```

Variables and Declaration Statements (continued)

- **Declaration statement:** Specifies the data type and identifier of a variable;
  - Syntax: `dataType variableName;`
- Data type is any valid C++ data type
  - Example: `int sum;`
- Declarations may be used anywhere in a function
  - Usually grouped at the opening brace

Variables and Declaration Statements (continued)

- **Character variables:** Declared using the **`char`** keyword
- Multiple variables of the same data type can be declared in a single declaration statement
  - Example:

```
double grade1, grade2, total, average;
```

- Variables can be initialized in a declaration
  - Example:

```
double grade1 = 87.0;
```

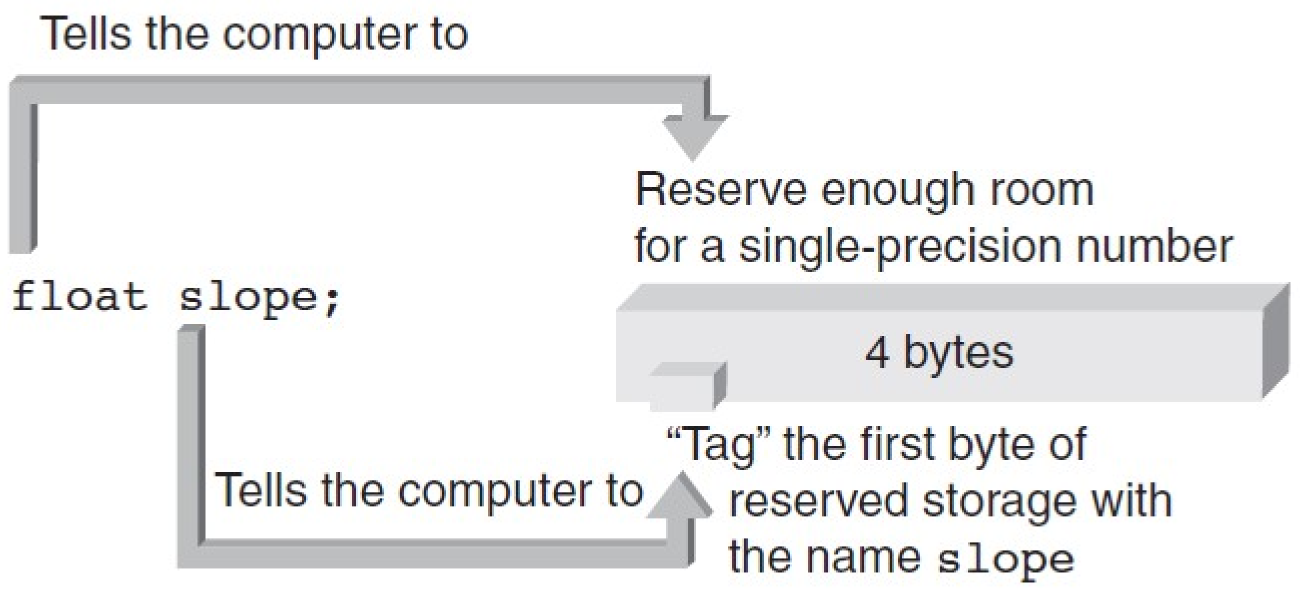
- A variable must be declared before it is used

Memory Allocation

- Three items associated with each variable:
  - Data type
  - Actual value stored in the variable (its contents)
  - Memory address of the variable
- Address operator (&) provides the variable’s address

Memory Allocation (continued)

- Declaring a variable causes memory to be allocated based on the data type



Tells the computer to
Reserve enough room for a single-precision number
float slope;
4 bytes
"Tag" the first byte of reserved storage with the name slope
Tells the computer to

Figure 2.10b Defining the floating-point variable named `slope`

Memory Allocation (continued)

Program 2.10

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

int main()
{
  int num;

  num = 22;
  cout << "The value stored in num is " << num << endl;
  cout << "The address of num = " << &num << endl;

  return 0;
}
```

A Case Study: Radar Speed Trap

- Step 1: Analyze the Problem
  - Understand the desired outputs
  - Determine the required inputs
- Step 2: Develop a Solution
  - Determine the algorithms to be used
  - Use top-down approach to design
- Step 3: Code the Solution
- Step 4: Test and Correct the Program

A Case Study: Radar Speed Trap (continued)

- Analyze the Problem
  - Output: Speed of the car
  - Inputs: Emitted frequency and received frequency
- Develop a Solution
  - Algorithm:
    - Assign values to f0 and f1
    - Calculate and display speed

A Case Study: Radar Speed Trap (continued)

- Code the Solution

Program 2.11

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

int main()
{
  double speed, fe, fr;

  fe = 2e10;
  fr = 2.0000004e10;

  speed = 6.685e8 * (fr - fe) / (fr + fe);
  cout << "The speed is " << speed << " miles/hour " << endl;

  return 0;
}
```

A Case Study: Radar Speed Trap (continued)

- Test and Correct the Program
  - Verify that the calculation and displayed value agree with the previous hand calculation
  - Use the program with different values of received frequencies

Common Programming Errors

- Omitting the parentheses after `main()`
- Omitting or incorrectly typing the opening brace, {, or the closing brace, }, that signifies the start and end of a function body
- Misspelling the name of an object or function
- Omitting a semicolon at end of statement
  - int a = 0

Common Programming Errors (continued)

- Missing `\n` to indicate new line
- Substituting letter O for zero and vice versa
- Failing to declare all variables

Summary

- A C++ program consists of one or more modules, called functions, one of which must be called **`main()`**
- All C++ statements must be terminated by a semicolon “;”
- Data types include **`int`**, **`float`**, **`bool`**

Summary (continued)

- Variables
  - Variables must be declared with their data type
  - A variable can be used only after it has been declared
  - Variables may be initialized when declared