Chapter 1: Introduction to Computers, Programs, and C++

Sections 1.1–1.3, 1.6–1.9

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

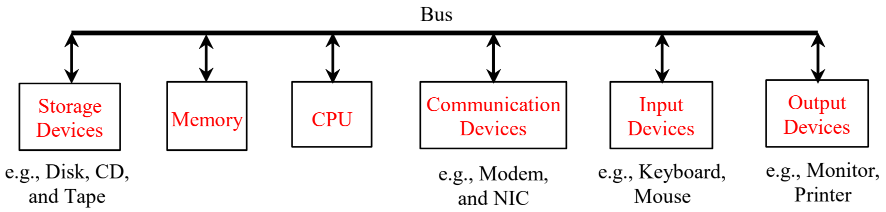
These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)

Outline

- Introduction and Computers (§§1.1–1.2)
- Programming languages (§§1.3)
- A simple C++ program for console output (§1.6)
- C++ program-development cycle (§1.7)
- Programming style and documentation (§1.8)
- Programming errors (§1.9)

What is a Computer?

A computer consists of a CPU, memory, hard disk, monitor, and communication devices.



Bus

Storage Devices

e.g., Disk, CD, and Tape

Memory

CPU

Communication Devices

e.g., Modem, and NIC

Input Devices

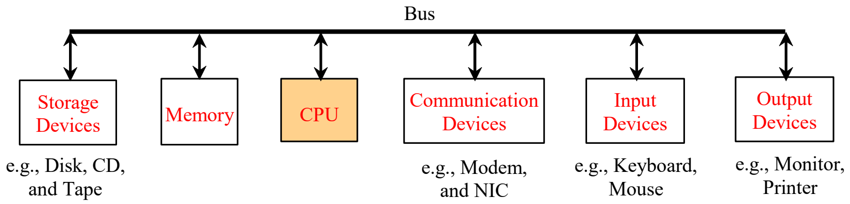
e.g., Keyboard, Mouse

Output Devices

e.g., Monitor, Printer

CPU

The central processing unit (CPU) is the brain of a computer. It retrieves instructions from memory and executes them. The CPU speed is measured in megahertz (MHz), with 1 megahertz equaling 1 million pulses per second. The speed of the CPU has been improved continuously. If you buy a PC now, you can get an Intel Core i7 Processor at 3 gigahertz (1 gigahertz is 1000 megahertz).



Bus

Storage Devices

e.g., Disk, CD, and Tape

Memory

CPU

Communication Devices

e.g., Modem, and NIC

Input Devices

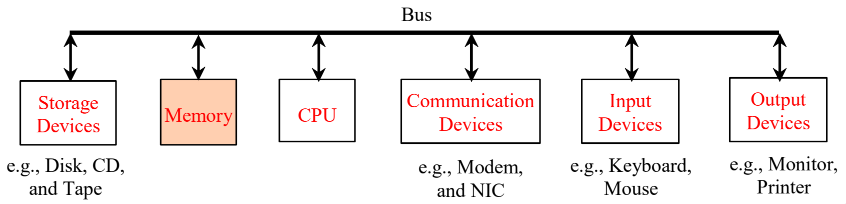
e.g., Keyboard, Mouse

Output Devices

e.g., Monitor, Printer

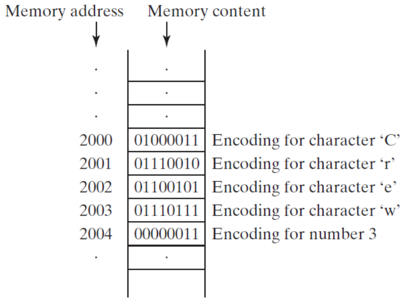
Memory

Memory is to store data and program instructions for CPU to execute. A memory unit is an ordered sequence of bytes, each holds eight bits. A program and its data must be brought to memory before they can be executed. A memory byte is never empty, but its initial content may be meaningless to your program. The current content of a memory byte is lost whenever new information is placed in it.



How Data is Stored?

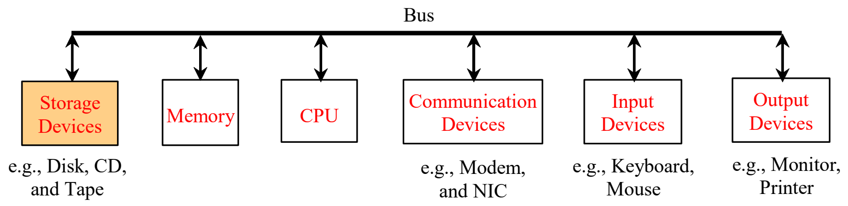
- Data of various kinds are encoded as a series of bits (*zeros* and *ones*).
- The encoding scheme varies. For example, character **'J'** is represented by 01001010 in one *byte*.
- A small number such as **3** can be stored in a single byte.
- If computer needs to store a large number that cannot fit into a single byte, it uses a number of adjacent bytes.
- A byte is the minimum storage unit.



Memory address	Memory content	
.	.	
.	.	
.	.	
2000	01000011	Encoding for character 'C'
2001	01110010	Encoding for character 'r'
2002	01100101	Encoding for character 'e'
2003	01110111	Encoding for character 'w'
2004	00000011	Encoding for number 3
.	.	

Storage Devices

Memory is volatile, because information is lost when the power is off. Programs and data are permanently stored on storage devices and are moved to memory when the computer actually uses them. There are four main types of storage devices: Disk drives (hard disks), Solid-state devices (SSD, Flash), CD drives (CD-R and CD-RW), and Tape drives.



Outline

- Introduction and Computers (§§1.1–1.2)
- Programming languages (§§1.3)
- A simple C++ program for console output (§1.6)
- C++ program-development cycle (§1.7)
- Programming style and documentation (§1.8)
- Programming errors (§1.9)

Programs

Computer *programs*, known as *software*, are instructions to the computer.

You tell a computer what to do through programs. Without programs, a computer is an empty machine. Computers do not understand human languages, so you need to use computer languages to communicate with them.

Programs are written using programming languages.

Programming Languages

Machine Language   Assembly Language   High-Level Language

Machine language is a set of primitive instructions built into every computer. The instructions are in the form of binary code, so you have to enter binary codes for various instructions. Program with native machine language is a tedious process. Moreover the programs are highly difficult to read and modify. For example, to add two numbers, you might write an instruction in binary like this:

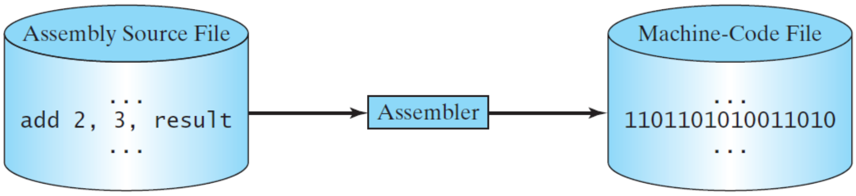
```
1101101010011010
```

Programming Languages

Machine Language Assembly Language High-Level Language

Assembly languages were developed to make programming easy. Since the computer cannot understand assembly language, however, a program called assembler is used to convert assembly language programs into machine code. For example, to add two numbers, you might write an instruction in assembly code like this:

```
add 2, 3, result
```



Programming Languages

Machine Language Assembly Language High-Level Language

The high-level languages are English-like and easy to learn and program. For example, the following is a high-level language statement that computes the area of a circle with radius 5:

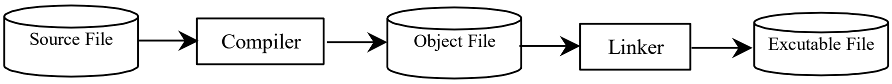
```
area = 5 * 5 * 3.1416;
```

Popular High-Level Languages

- COBOL (COmmon Business Oriented Language)
- FORTRAN (FORmula TRANslation)
- BASIC (Beginner All-purpose Symbolic Instructional Code)
- Pascal (named for Blaise Pascal)
- Ada (named for Ada Lovelace)
- C (whose developer designed B first)
- Visual Basic (Basic-like visual language developed by Microsoft)
- Delphi (Pascal-like visual language developed by Borland)
- **C++ (an object-oriented language, based on C)**
- Java (a popular object-oriented language, similar to C++)
- C# (a Java-like developed my Microsoft)

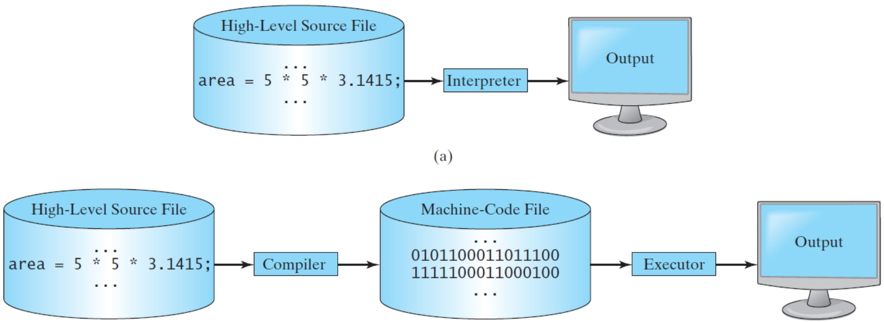
Compiling Source Code

A program written in a high-level language is called a *source program*. Since a computer cannot understand a source program. Program called a *compiler* is used to translate the source program into a machine language program called an *object program*. The object program is often then linked with other supporting library code before the object can be executed on the machine.



Compiling versus Interpretation

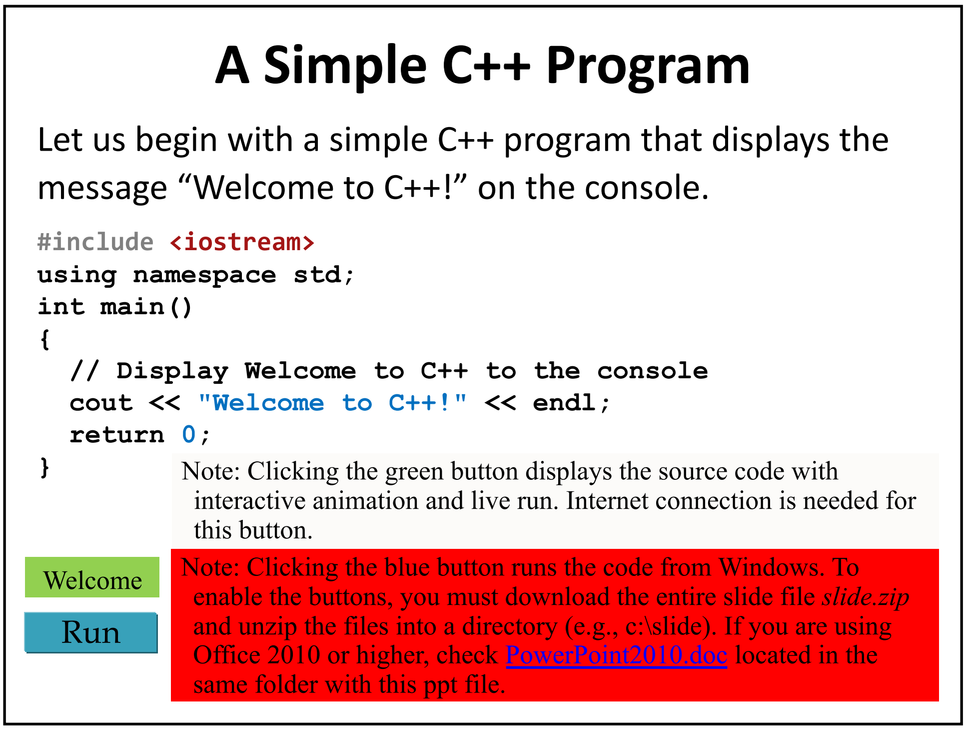
- Some programming languages like Python have interpreters that translate and execute a program one statement at a time (a).
- C++ needs a compiler that translates the entire source program into a machine-language file for execution (b).



(b)

Outline

- Introduction and Computers (§§1.1–1.2)
- Programming languages (§§1.3)
- A simple C++ program for console output (§1.6)
- C++ program-development cycle (§1.7)
- Programming style and documentation (§1.8)
- Programming errors (§1.9)



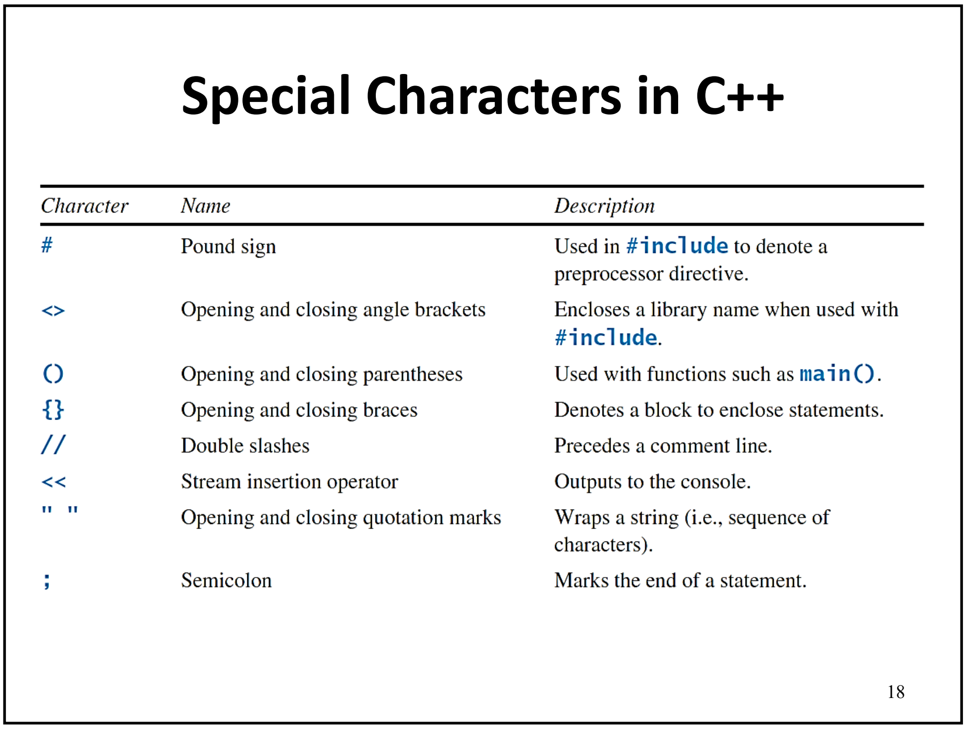
A Simple C++ Program

Let us begin with a simple C++ program that displays the message “Welcome to C++!” on the console.

```
#include <iostream>
using namespace std;
int main()
{
  // Display Welcome to C++ to the console
  cout << "Welcome to C++!" << endl;
  return 0;
}
```

Note: Clicking the green button displays the source code with interactive animation and live run. Internet connection is needed for this button.

Note: Clicking the blue button runs the code from Windows. To enable the buttons, you must download the entire slide file *slide.zip* and unzip the files into a directory (e.g., c:\slide). If you are using Office 2010 or higher, check PowerPoint2010.doc located in the same folder with this ppt file.



Special Characters in C++

Character	*Name*	*Description*
`#`	Pound sign	Used in `#include` to denote a preprocessor directive.
`<>`	Opening and closing angle brackets	Encloses a library name when used with `#include`.
`()`	Opening and closing parentheses	Used with functions such as `main()`.
`{}`	Opening and closing braces	Denotes a block to enclose statements.
`//`	Double slashes	Precedes a comment line.
`<<`	Stream insertion operator	Outputs to the console.
`" "`	Opening and closing quotation marks	Wraps a string (i.e., sequence of characters).
`;`	Semicolon	Marks the end of a statement.

Comments in C++

```
// This application program prints Welcome to C++!
/* This application program prints Welcome to C++! */
/* This application program
   prints Welcome to C++! */
```

Extending the Simple C++ Program

Once you understand the program, it is easy to extend it to display more messages. For example, you can rewrite the program to display three messages.

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Programming is fun!" << endl;
  cout << "Fundamentals First" << endl;
  cout << "Problem Driven" << endl;
  return 0;
}
```

WelcomeWithThreeMessages Run

Computing with Numbers

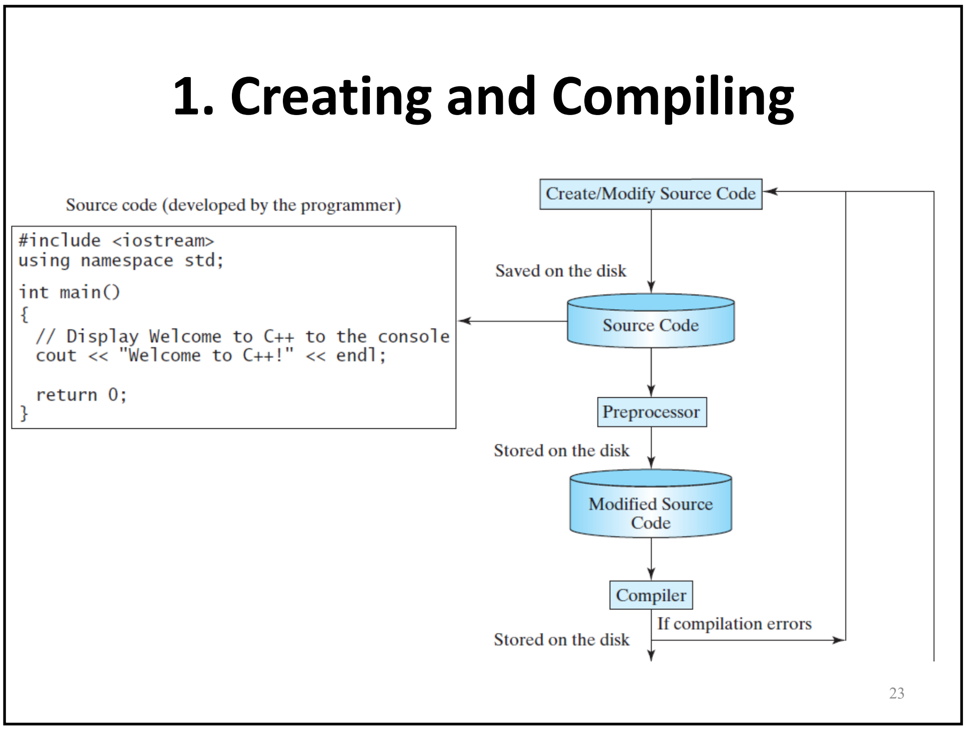
Further, you can perform mathematical computations and displays the result to the console. Listing 1.3 gives such an example.

```
#include <iostream>
using namespace std;
int main()
{
   cout << "(10.5 + 2 * 3) / (45 - 3.5) = ";
   cout << (10.5 + 2 * 3) / (45 - 3.5) << endl;

   return 0;
}
```

ComputeExpression Run

Outline

- Introduction and Computers (§§1.1–1.2)
- Programming languages (§§1.3)
- A simple C++ program for console output (§1.6)
- C++ program-development cycle (§1.7)
- Programming style and documentation (§1.8)
- Programming errors (§1.9)



1. Creating and Compiling

Source code (developed by the programmer)

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

int main()
{
  // Display Welcome to C++ to the console
  cout << "Welcome to C++!" << endl;

  return 0;
}
```

Create/Modify Source Code

Saved on the disk

Source Code

Preprocessor

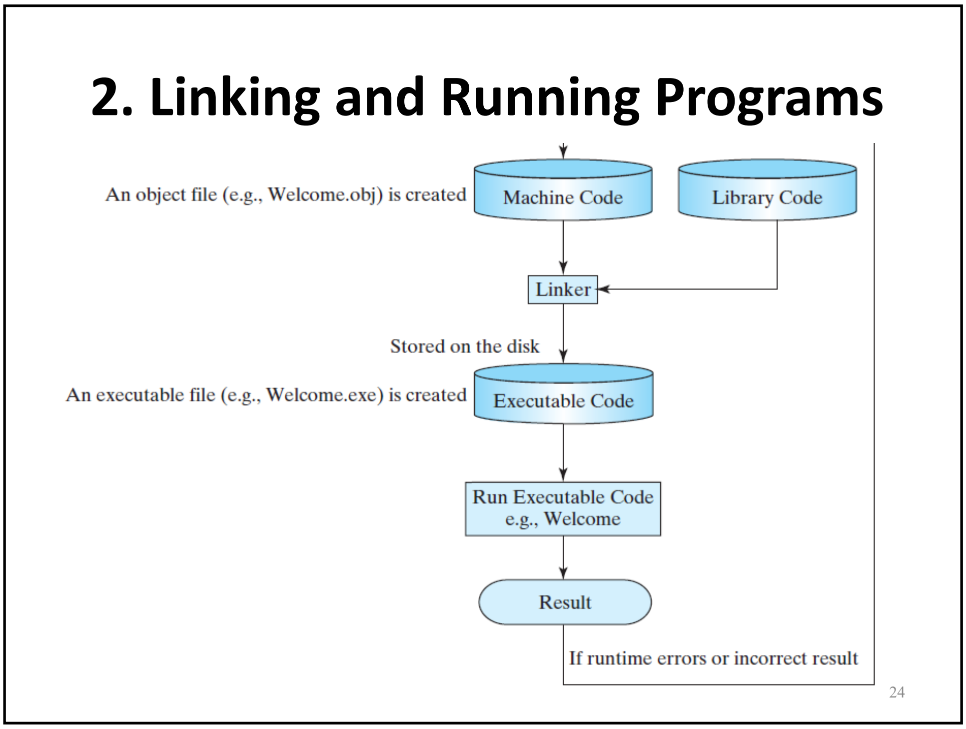
Stored on the disk

Modified Source Code

Compiler

If compilation errors

Stored on the disk



2. Linking and Running Programs

An object file (e.g., Welcome.obj) is created

Machine Code

Library Code

Linker

Stored on the disk

An executable file (e.g., Welcome.exe) is created

Executable Code

Run Executable Code
e.g., Welcome

Result

If runtime errors or incorrect result

C++ IDE Tutorial

You can develop a C++ program from a command window or from an IDE. An IDE is software that provides an *integrated development environment* (*IDE*) for rapidly developing C++ programs. Editing, compiling, building, debugging, and online help are integrated in one graphical user interface. Just enter source code or open an existing file in a window, then click a button, menu item, or function key to compile and run the program. Examples of popular IDEs are **Microsoft Visual Studio**, Dev-C++, Eclipse, and NetBeans. All these IDEs can be downloaded free.

Outline

- Introduction and Computers (§§1.1–1.2)
- Programming languages (§§1.3)
- A simple C++ program for console output (§1.6)
- C++ program-development cycle (§1.7)
- Programming style and documentation (§1.8)
- Programming errors (§1.9)

Programming Style and Documentation

- Appropriate Comments
- Proper Indentation and Spacing Lines
- Block Styles

```
#include <iostream>
using namespace std;
int main()
{
   cout << "(10.5 + 2 * 3) / (45 - 3.5) = ";
   cout << (10.5 + 2 * 3) / (45 - 3.5) << endl;

   return 0;
}
```

Outline

- Introduction and Computers (§§1.1–1.2)
- Programming languages (§§1.3)
- A simple C++ program for console output (§1.6)
- C++ program-development cycle (§1.7)
- Programming style and documentation (§1.8)
- Programming errors (§1.9)

Programming Errors

1. Syntax Errors
2. Runtime Errors
3. Logic Errors

Syntax Errors

```
#include <iostream>
using namespace std

int main()
{
  cout << "Programming is fun << endl;

  return 0;
}
```

ShowSyntaxErrors

Runtime Errors

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

int main()
{
    int i = 4;
    int j = 0;
    cout << i / j << endl;

    return 0;
}
```

ShowRuntimeErrors Run

Logic Errors

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

int main()
{
  cout << "Celsius 35 is Fahrenheit degree " << endl;
  cout << (9 / 5) * 35 + 32 << endl;

  return 0;
}
```

ShowLogicErrors Run

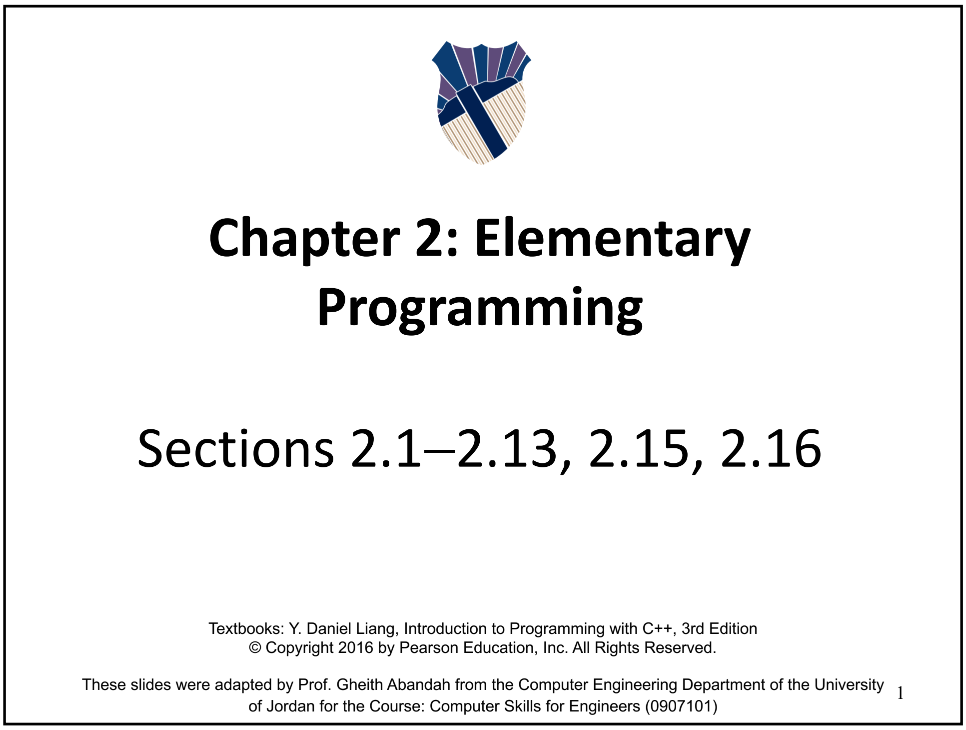
Common Errors

1. Missing Braces
2. Missing Semicolons
3. Missing Quotation Marks
4. Misspelling Names

```
int man()
{
      cout << "Programming is fun!" << endl;
      cout << "Fundamentals First" << endl;
      cout << "Problem Driven  << endl
```

Outline

- Introduction and Computers (§§1.1–1.2)
- Programming languages (§§1.3)
- A simple C++ program for console output (§1.6)
- C++ program-development cycle (§1.7)
- Programming style and documentation (§1.8)
- Programming errors (§1.9)

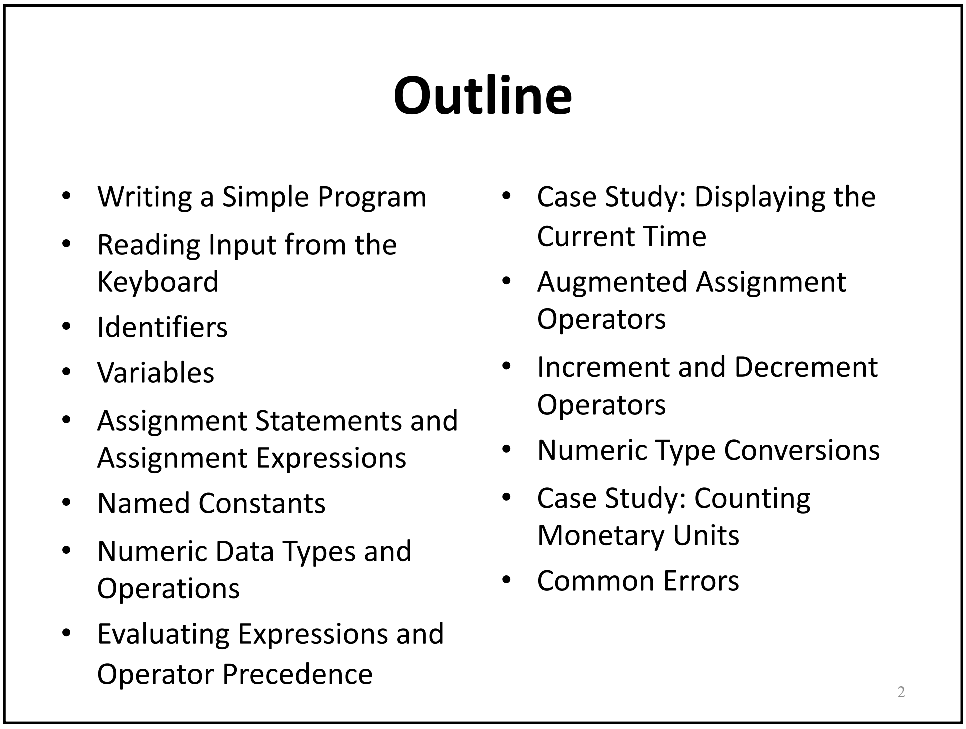


Chapter 2: Elementary Programming

Sections 2.1–2.13, 2.15, 2.16

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)



Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Writing a Simple Program

A program that computes the area of the circle.

ComputeArea

Run

Note: Clicking the green button displays the source code with interactive animation. You can also run the code in a browser. Internet connection is needed for this button.

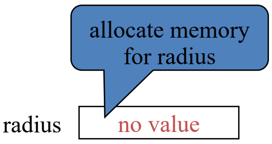
Note: Clicking the blue button runs the code from Windows. If you cannot run the buttons, see

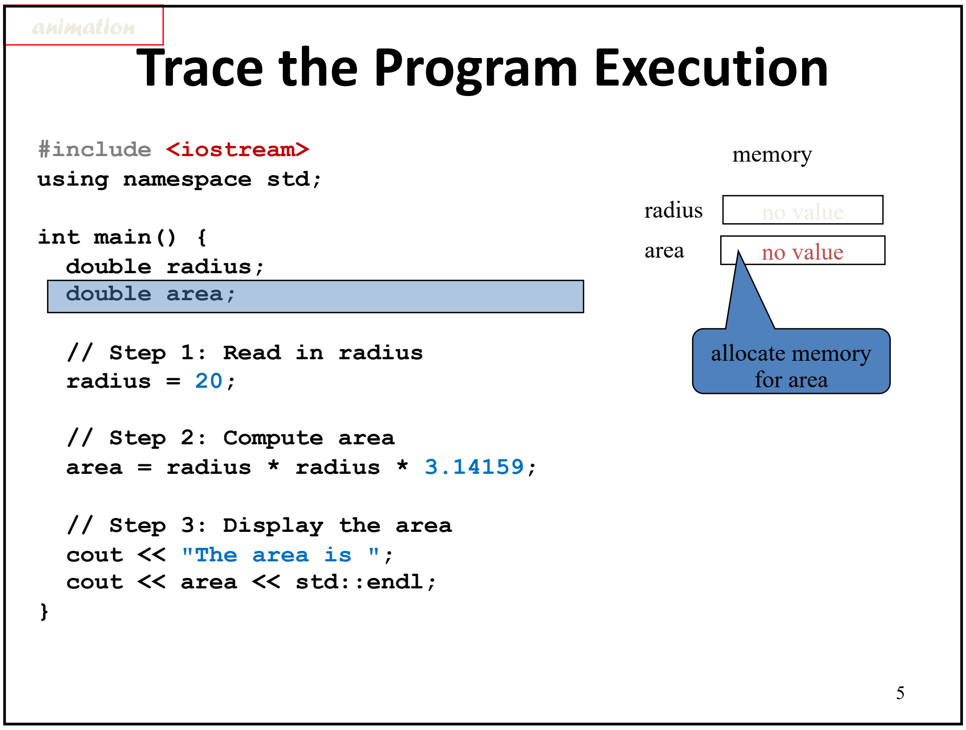
IMPORTANT NOTE: If you cannot run the buttons, see www.cs.armstrong.edu/liang/javaslidenote.doc.

animation

Trace the Program Execution

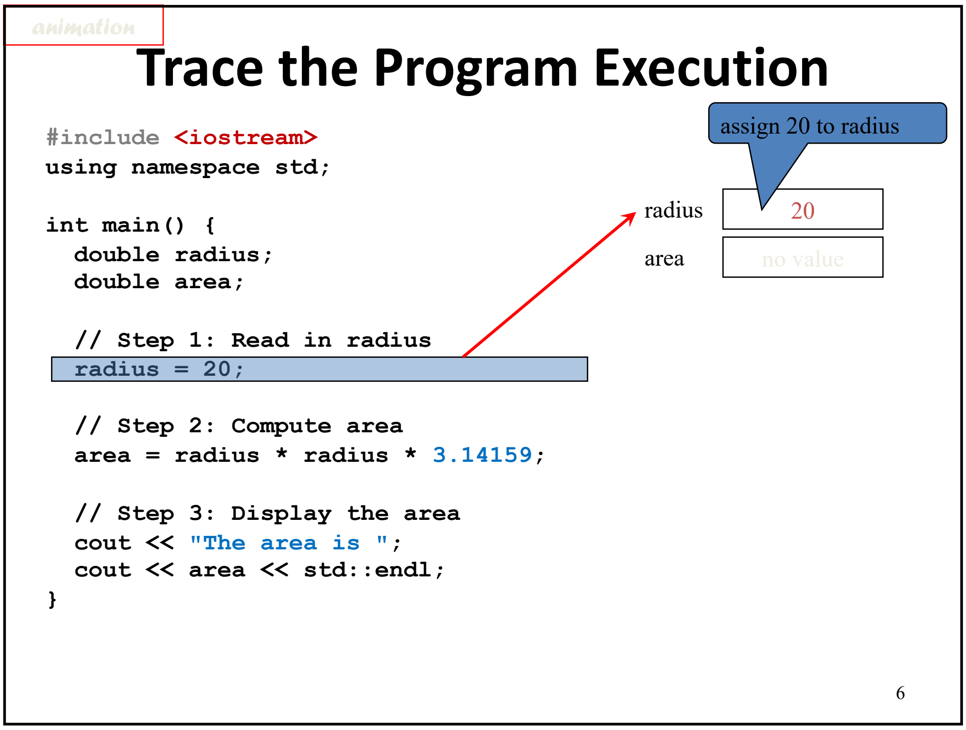
```
#include <iostream>
using namespace std;

int main() {
  double radius;
  double area;

  // Step 1: Read in radius
  radius = 20;

  // Step 2: Compute area
  area = radius * radius * 3.14159;

  // Step 3: Display the area
  cout << "The area is ";
  cout << area << endl;
}
```





Trace the Program Execution

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

int main() {
  double radius;
  double area;

  // Step 1: Read in radius
  radius = 20;

  // Step 2: Compute area
  area = radius * radius * 3.14159;

  // Step 3: Display the area
  cout << "The area is ";
  cout << area << std::endl;
}
```



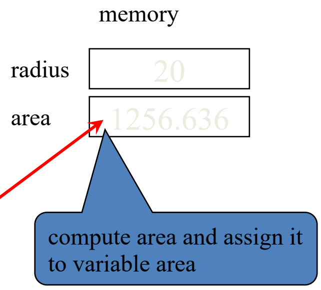
Trace the Program Execution

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

int main() {
  double radius;
  double area;

  // Step 1: Read in radius
  radius = 20;

  // Step 2: Compute area
  area = radius * radius * 3.14159;

  // Step 3: Display the area
  cout << "The area is ";
  cout << area << std::endl;
}
```

animation

Trace the Program Execution

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

int main() {
  double radius;
  double area;

  // Step 1: Read in radius
  radius = 20;

  // Step 2: Compute area
  area = radius * radius * 3.14159;

  // Step 3: Display the area
  cout << "The area is ";
  cout << area << std::endl;
}
```



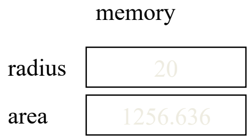
compute area and assign it to variable area

---

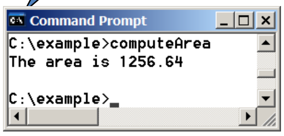
animation

Trace the Program Execution

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

int main() {
  double radius;
  double area;

  // Step 1: Read in radius
  radius = 20;

  // Step 2: Compute area
  area = radius * radius * 3.14159;

  // Step 3: Display the area
  cout << "The area is ";
  cout << area << std::endl;
}
```



print a message to the console



```
C:\example>computeArea
The area is 1256.64

C:\example>_
```

Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Reading Input from the Keyboard

You can use the **`cin`** object to read input from the keyboard.

```
cin >> radius;
```

ComputeAreaWithConsoleInput Run

Reading Multiple Input in One Statement

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

int main()
{
  // Prompt the user to enter three numbers
  double number1, number2, number3;
  cout << "Enter three numbers: ";
  cin >> number1 >> number2 >> number3;

  // Compute average
  double average = (number1 + number2 + number3) / 3;

  // Display result
  cout << "The average of " << number1 << " " << number2
      << " " << number3 << " is " << average << endl;

  return 0;
}
```

ComputeAverage Run

Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Identifiers

Identifiers are the names that identify elements such as variables and functions in a program.

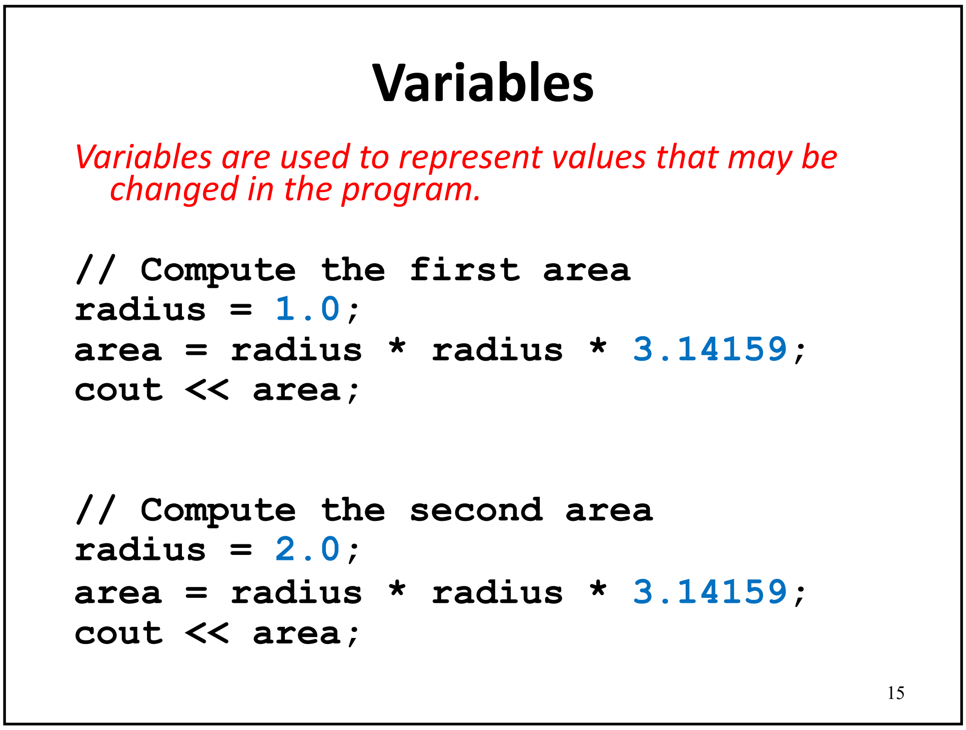
- An identifier is a sequence of characters that consists of letters, digits, and underscores (_).
- An identifier must start with a letter or an underscore. It cannot start with a digit.
- An identifier cannot be a reserved word. (See Appendix A, “C++ Keywords,” for a list of reserved words.)
- An identifier can be of any length, but your C++ compiler may impose some restriction. Use identifiers of 31 characters or fewer to ensure portability.

Which of the following identifiers are valid? Which are C++ keywords?

```
miles, Test, a++, --a, 4#R, $4, #44, apps
main, double, int, x, y, radius
```

Outline

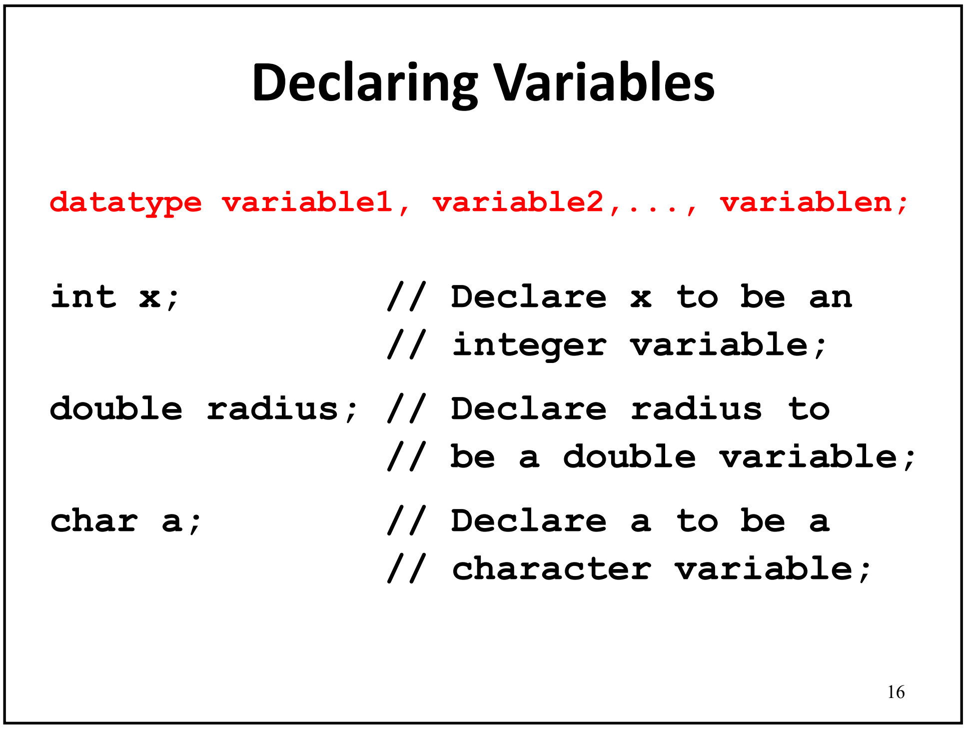
- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors



Variables

Variables are used to represent values that may be changed in the program.

```
// Compute the first area
radius = 1.0;
area = radius * radius * 3.14159;
cout << area;


// Compute the second area
radius = 2.0;
area = radius * radius * 3.14159;
cout << area;
```



Declaring Variables

```
datatype variable1, variable2,..., variablen;

int x;            // Declare x to be an
                  // integer variable;
double radius;    // Declare radius to
                  // be a double variable;
char a;           // Declare a to be a
                  // character variable;
```

Declaring Variables

```
int i, j, k;    // Declare three integers

int i = 10;     // Declare and initialize

int i(1), j(2); // Is equivalent to
int i = 1, j = 2;
```

Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Assignment Statements

An assignment statement designates a value for a variable. An assignment statement can be used as an expression in C++.

```
x = 1;            // Assign 1 to x;

y = x + 1;        // Assign 2 to y;
radius = 1.0;     // Assign 1.0 to radius;
a = 'A';          // Assign 'A' to a;
```

Assignment Statements

An assignment statement designates a value for a variable.

```
i = j = k = 1; // Assigns 1 to the three
               // variables


cout << x = 1; // Assigns 1 to x and
               // outputs 1
```

Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Named Constants

A named constant is an identifier that represents a permanent value.

```
const datatype CONSTANTNAME = VALUE;
```

```
const double PI = 3.14159;
```

ComputeAreaConstant Run

Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Numerical Data Types

- **Signed integers**
  - 16 bits: `short` `-3`
  - 32 bits: `int` `100000`
  - 64 bits: `long long` `-2147483648`
- **Unsigned integers**
  - 16 bits: `unsigned short` `4`
  - 32 bits: `unsigned`
  - 64 bits: `unsigned long long`

Synonymous Types

`short int` is synonymous to `short`. For example,

```
    short int i = 2;
```

is same as

```
    short i = 2;
```

```
unsigned short int    ≡ unsigned short
unsigned int          ≡ unsigned
long int              ≡ long
unsigned long int     ≡ unsigned long
```

Numerical Data Types

- **Floating-point numbers**
  - 32 bits: `float` `1.5`
  - 64 bits: `double` `-1.23456E+2`
  - 80 bits: `long double` `9.1e-1000`

- When a number such as 50.534 is converted into scientific notation such as 5.0534e+1, its decimal point is moved (i.e., floated) to a new position.

`double` vs. `float`

The double type values are more accurate than the float type values. For example,

```
cout << "1.0 / 3.0 is " << 1.0 / 3.0 << endl;
```

```
1.0 / 3.0 is 0.3333333333333331
```

16 digits

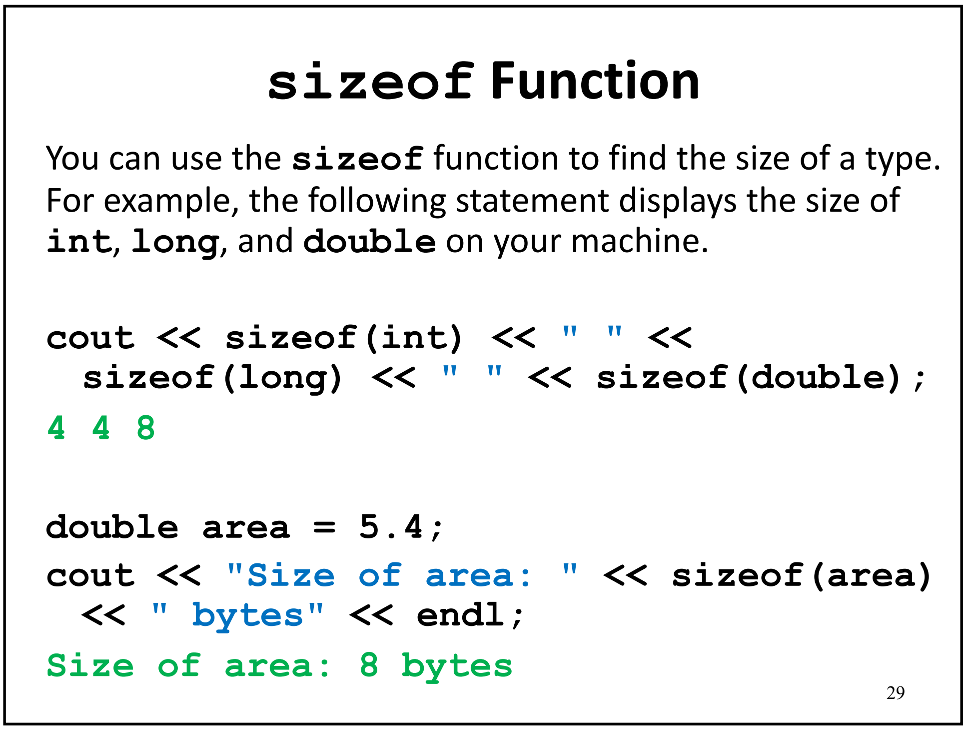
```
cout << "1.0F / 3.0F is " << 1.0F / 3.0F << endl
```

```
1.0F / 3.0F is 0.3333333432674408
```

7 digits

Numerical Data Types

Name	**Synonymy**	**Range**	**Storage Size**
short	short int	-2^{15} to $2^{15}-1$ (-32,768 to 32,767)	16-bit signed
unsigned short	unsigned short int	0 to $2^{16}-1$ (65535)	16-bit unsigned
int	signed	-2^{31} to $2^{31}-1$ (-2147483648 to 2147483647)	32-bit
unsigned	unsigned int	0 to $2^{32}-1$ (4294967295)	32-bit unsigned
long	long int	-2^{31} (-2147483648) to $2^{31}-1$ (2147483647)	32-bit signed
unsigned long	unsigned long int	0 to $2^{32}-1$ (4294967295)	32-bit unsigned
long long		-2^{63} (-9223372036854775808) to 263-1 (9223372036854775807)	64-bit signed
float		Negative range: -3.4028235E+38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38	32-bit IEEE 754
double		Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308	64-bit IEEE 754
long double		Negative range: -1.18E+4932 to -3.37E-4932 Positive range: 3.37E-4932 to 1.18E+4932 Significant decimal digits: 19	80-bit

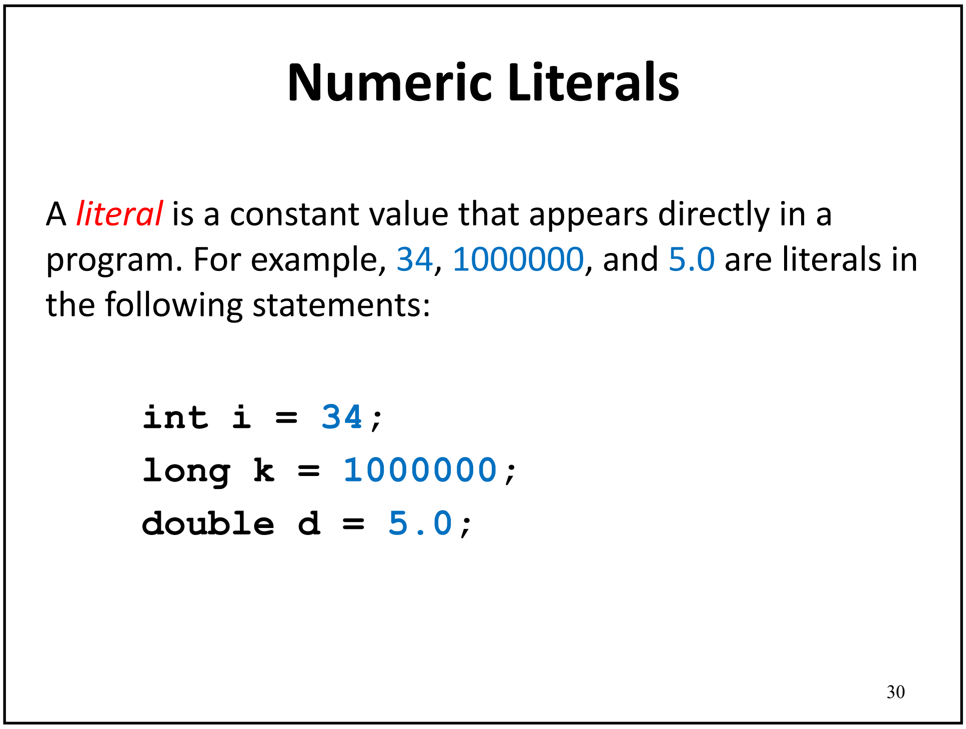


`sizeof` Function

You can use the **`sizeof`** function to find the size of a type. For example, the following statement displays the size of **`int`**, **`long`**, and **`double`** on your machine.

```
cout << sizeof(int) << " " <<
  sizeof(long) << " " << sizeof(double);
4 4 8
```

```
double area = 5.4;
cout << "Size of area: " << sizeof(area)
  << " bytes" << endl;
Size of area: 8 bytes
```



Numeric Literals

A *literal* is a constant value that appears directly in a program. For example, 34, 1000000, and 5.0 are literals in the following statements:

```
int i = 34;
long k = 1000000;
double d = 5.0;
```

octal and hex literals

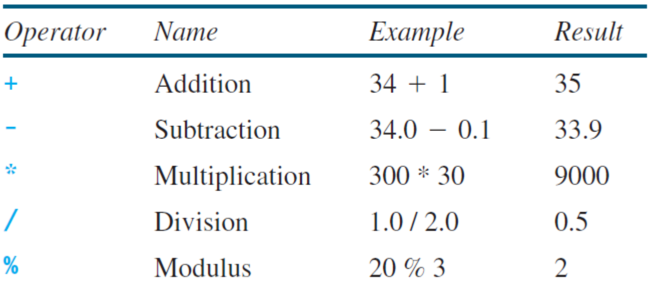
- By default, an integer literal is a *decimal* number.
- To denote a *binary* integer literal, use a leading `0b` or `0B` (zero b).
- To denote an *octal* integer literal, use a leading `0` (zero)
- To denote a *hexadecimal* integer literal, use a leading `0x` or `0X` (zero x).

```
cout << 10 << " " << 0b10 << " " << 010
  << " " << 0x10;
10 2 8 16
```

Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Numeric Operators



Operator	Name	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 − 0.1	33.9
*	Multiplication	300 * 30	9000
/	Division	1.0 / 2.0	0.5
%	Modulus	20 % 3	2

Integer Division

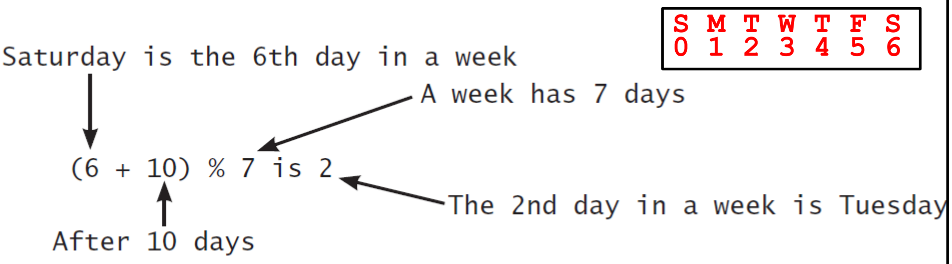
5 / 3 yields an integer 1.

5.0 / 2 yields a double value 2.5

5 % 2 yields 1 (the remainder of the division)

Remainder Operator

Remainder is very useful in programming. For example, an even number % 2 is always 0 and an odd number % 2 is always 1. So you can use this property to determine whether a number is even or odd.
Suppose today is Saturday and you and your friends are going to meet in 10 days. What day is in 10 days? You can find that day is Tuesday using the following expression:



S	M	T	W	T	F	S
0	1	2	3	4	5	6

Saturday is the 6th day in a week

A week has 7 days

(6 + 10) % 7 is 2

The 2nd day in a week is Tuesday

After 10 days

Example: Displaying Time

A program that obtains minutes from seconds.

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

int main()
{
  // Prompt the user for input
  int seconds;
  cout << "Enter an integer for seconds: ";
  cin >> seconds;
  int minutes = seconds / 60;
  int remainingSeconds = seconds % 60;
  cout << seconds << " seconds is " << minutes <<
    " minutes and " << remainingSeconds << " seconds " << endl;

  return 0;
}
```

DisplayTime Run

Exponent Operations

`pow(a, b)` = a^b

```
cout << pow(2.0, 3) << endl;
8
cout << pow(4.0, 0.5) << endl;
2
cout << pow(2.5, 2) << endl;
6.25
cout << pow(2.5, -2) << endl;
0.16
```

Overflow

When a variable is assigned a value that is too large to be stored, it causes *overflow*.

For example, executing the following statement causes overflow, because the largest value that can be stored in a variable of the **`short`** type is 32767. 32768 is too large.

```
short value = 32767 + 1;
```

Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Arithmetic Expressions

$$\frac{3 + 4x}{5} - \frac{10(y - 5)(a + b + c)}{x} + 9\left(\frac{4}{x} + \frac{9 + x}{y}\right)$$

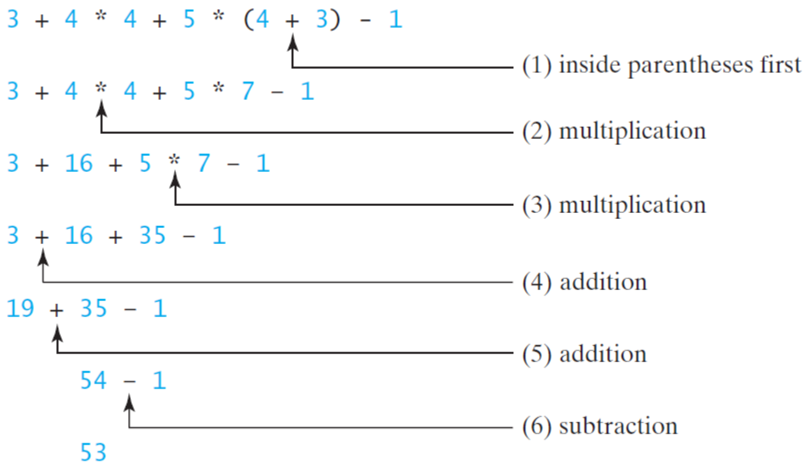
is translated to

```
(3+4*x)/5 – 10*(y-5)*(a+b+c)/x + 9*(4/x +
   (9+x)/y)
```

Precedence

`()`	Operators contained within pairs of parentheses are evaluated first.
`* / %`	Multiplication, division, and remainder operators are applied next.
`+ -`	Addition and subtraction operators are applied last.
→	If an expression contains several similar operators, they are applied from left to right.

Precedence Example



`3 + 4 * 4 + 5 * (4 + 3) - 1`

(1) inside parentheses first

`3 + 4 * 4 + 5 * 7 - 1`

(2) multiplication

`3 + 16 + 5 * 7 - 1`

(3) multiplication

`3 + 16 + 35 - 1`

(4) addition

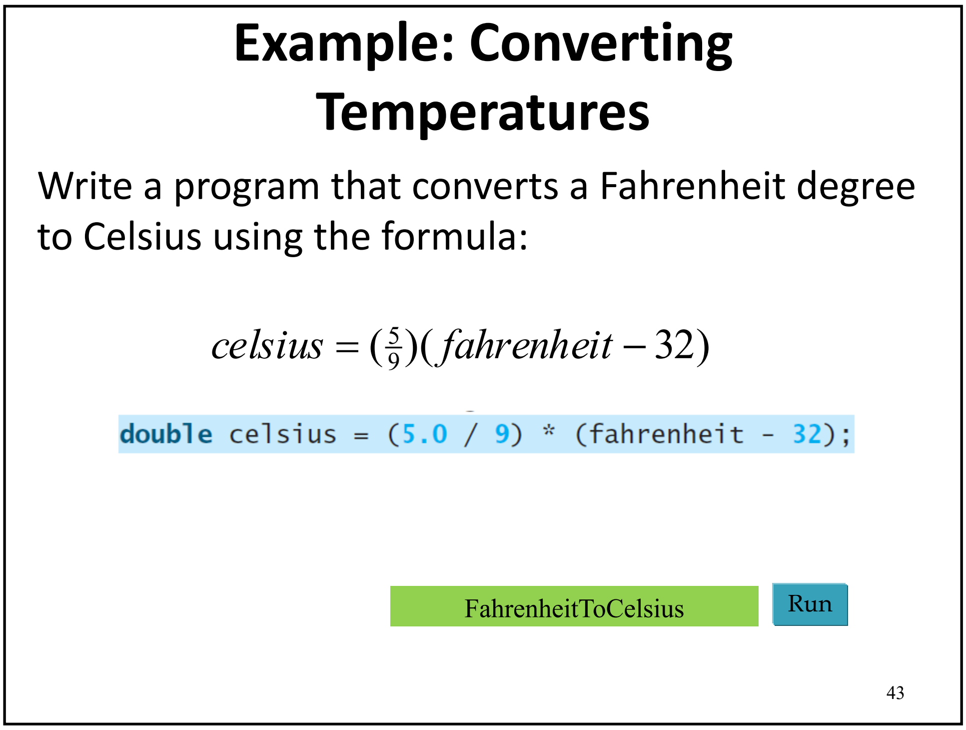
`19 + 35 - 1`

(5) addition

`54 - 1`

(6) subtraction

`53`

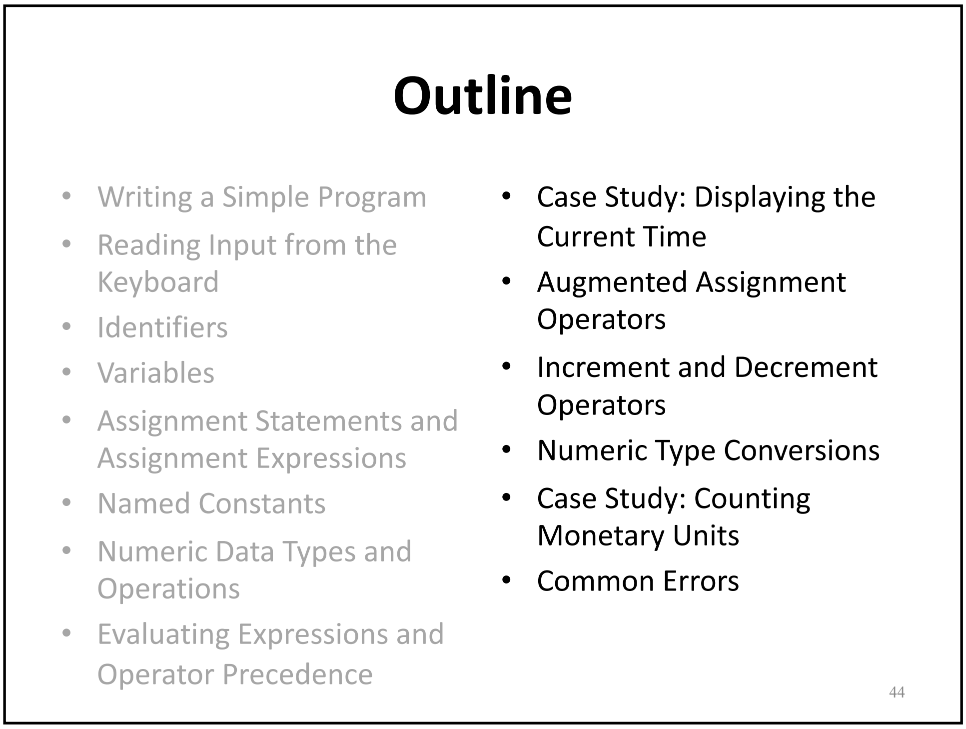


Example: Converting Temperatures

Write a program that converts a Fahrenheit degree to Celsius using the formula:

$$celsius = (\tfrac{5}{9})(fahrenheit - 32)$$

```
double celsius = (5.0 / 9) * (fahrenheit - 32);
```



Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Displaying the Current Time

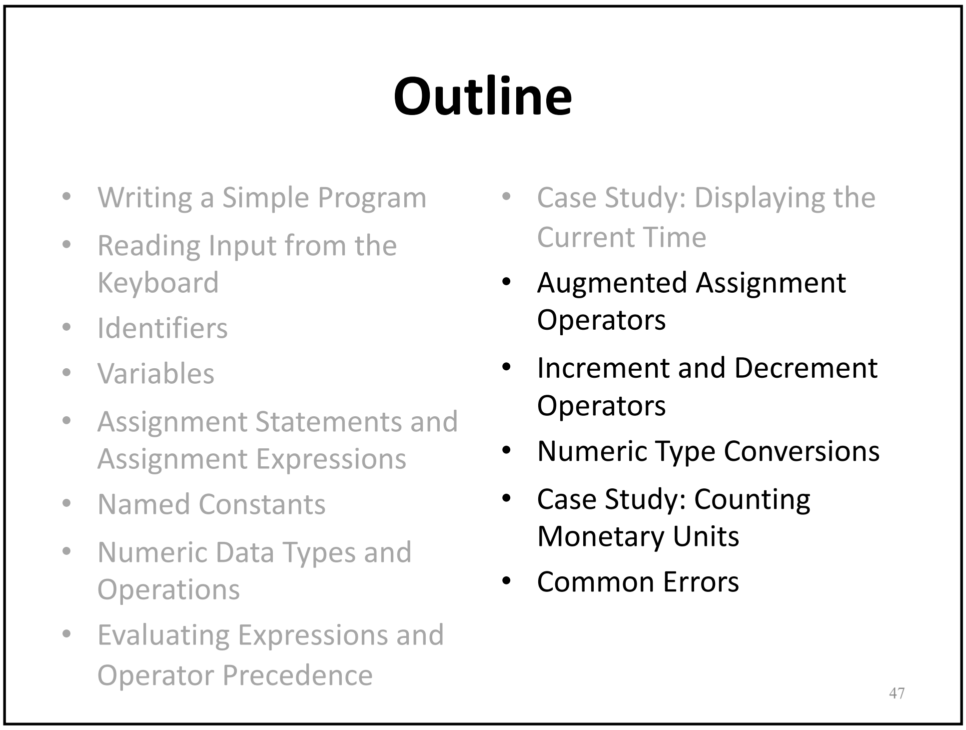
Write a program that displays current time in GMT in the format hour:minute:second such as 1:45:19.

The **`time(0)`** function in the **`ctime`** header file returns the current time in seconds elapsed since the time 00:00:00 on January 1, 1970 GMT, as shown in Figure 2.1. This time is known as the Unix epoch because 1970 was the year when the Unix operating system was formally introduced.

ShowCurrentTime Run

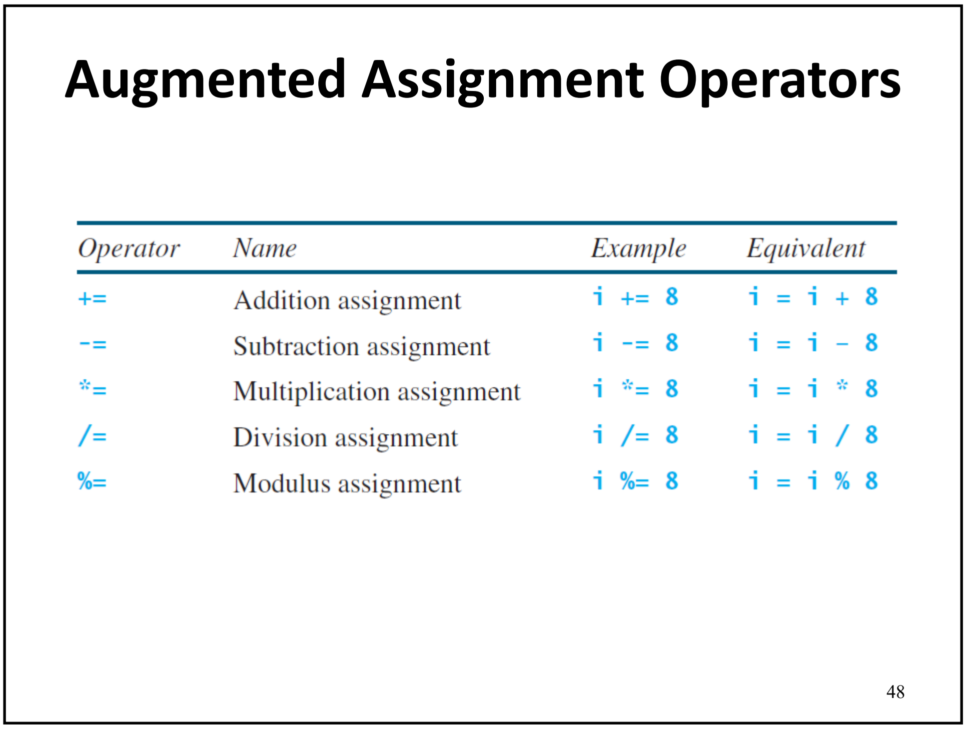
ShowCurrentTime.cpp

```
#include <iostream>
#include <ctime>
using namespace std;
int main() {
  // Obtain the total seconds since the midnight, Jan 1, 1970
  int totalSeconds = time(0);
  // Compute the current second in the minute in the hour
  int currentSecond = totalSeconds % 60;
  // Obtain the total minutes
  int totalMinutes = totalSeconds / 60;
  // Compute the current minute in the hour
  int currentMinute = totalMinutes % 60;
  // Obtain the total hours
  long totalHours = totalMinutes / 60;
  // Compute the current hour
  int currentHour = (int)(totalHours % 24);
  // Display results
  cout << "Current time is " << currentHour << ":"
    << currentMinute << ":" << currentSecond << " GMT" << endl;
  return 0;
}
```



Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors



Augmented Assignment Operators

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /= 8`	`i = i / 8`
`%=`	Modulus assignment	`i %= 8`	`i = i % 8`

Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Increment and Decrement Operators

Operator	Name	Description
`++var`	pre-increment	Increments **`var`** by 1 and evaluates to the new value in **`var`** after the increment.
`var++`	post-increment	Evaluates to the original value in **`var`** and increments **`var`** by 1.
`--var`	pre-decrement	Decrements **`var`** by 1 and evaluates to the new value in **`var`** after the decrement.
`var--`	post-decrement	Evaluates to the original value in **`var`** and decrements **`var`** by 1.

Increment and Decrement Operators, cont.

What is the output of the following two sequences?

```
int i = 10;
int newNum = 10 * i++;
cout << "i is " << i
  << ", newNum is " << newNum;
```

Same effect as

```
int newNum = 10 * i;
i = i + 1;
```

```
int i = 10;
int newNum = 10 * (++i);
cout << "i is " << i
  << ", newNum is " << newNum;
```

Same effect as

```
i = i + 1;
int newNum = 10 * i;
```

Increment and Decrement Operators, cont.

Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read. Avoid using these operators in expressions that modify multiple variables, or the same variable for multiple times such as this:

```
int k = ++i + i;   // Avoid!
```

Outline

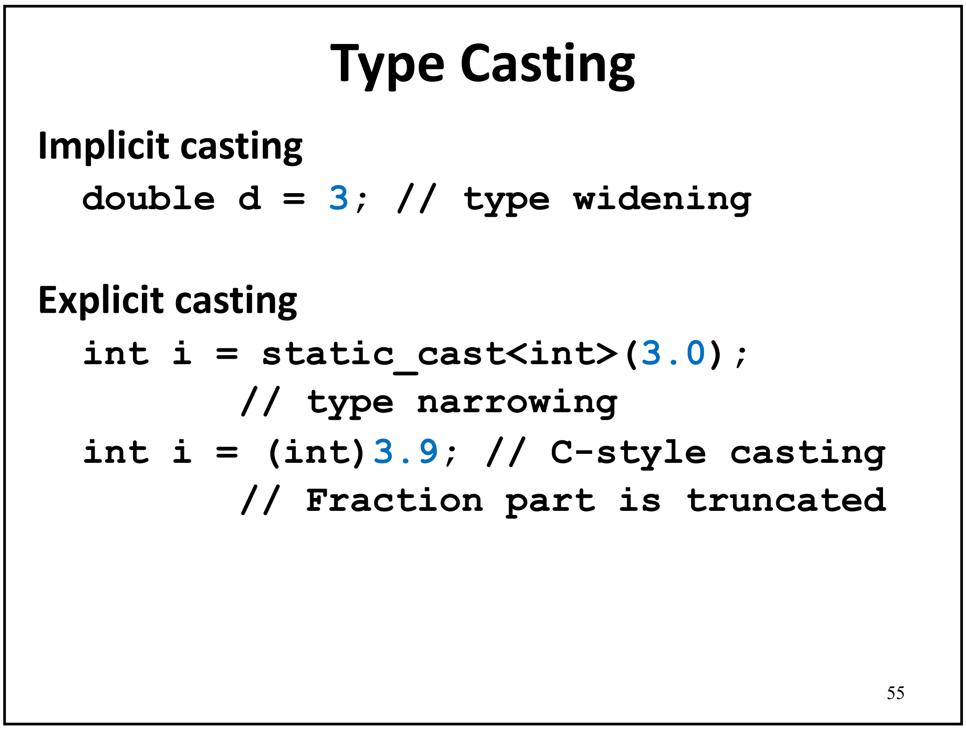
- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Numeric Type Conversion

Consider the following statements:

```
short i = 100;
long k = i * 3 + 4;
double d = i * 3.1 + k / 2;
```

```
int i = 34.7;        // i becomes 34
double f = i;        // f is now 34
double g = 34.3;     // g becomes 34.3
int j = g;           // j is now 34
```



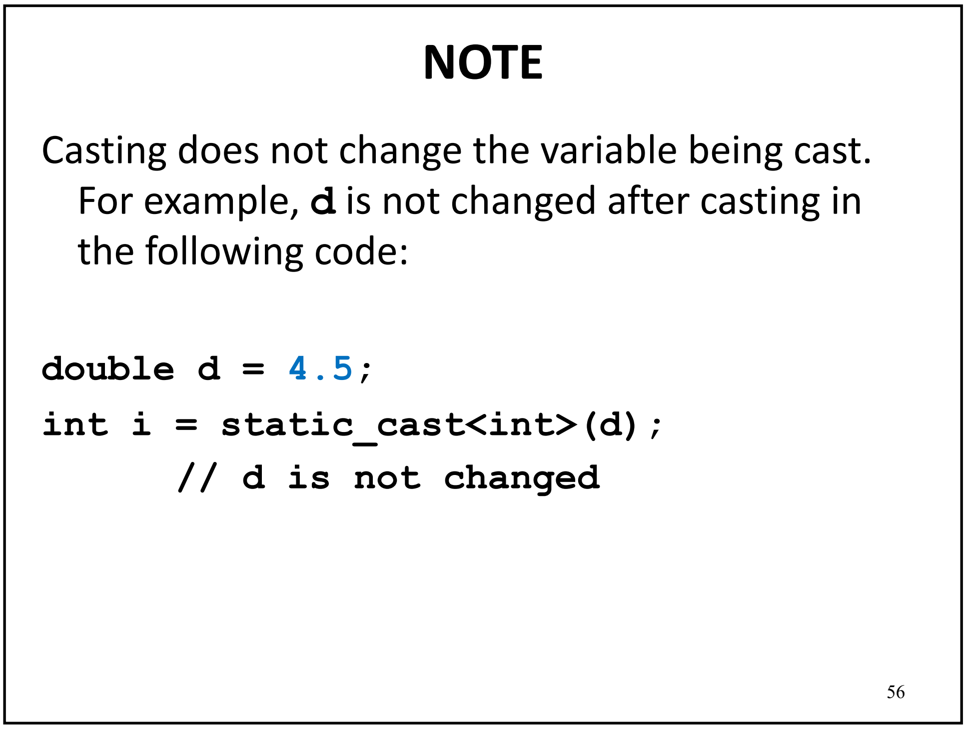
Type Casting

Implicit casting

```
double d = 3; // type widening
```

Explicit casting

```
int i = static_cast<int>(3.0);
       // type narrowing
int i = (int)3.9; // C-style casting
       // Fraction part is truncated
```



NOTE

Casting does not change the variable being cast. For example, **`d`** is not changed after casting in the following code:

```
double d = 4.5;
int i = static_cast<int>(d);
      // d is not changed
```

NOTE

The GNU and Visual C++ compilers will give a warning when you narrow a type unless you use `static_cast` to make the conversion explicit.

Example: Keeping Two Digits after Decimal Points

Write a program that displays the 6%-sales tax with two digits after the decimal point.

```
cout << "Sales tax is " <<
   static_cast<int>(tax * 100) / 100.0;
```

SalesTax Run

Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Case Study: Counting Monetary Units

This program lets the user enter the amount in decimal representing dollars and cents and output a report listing the monetary equivalent in single dollars, quarters, dimes, nickels, and pennies.

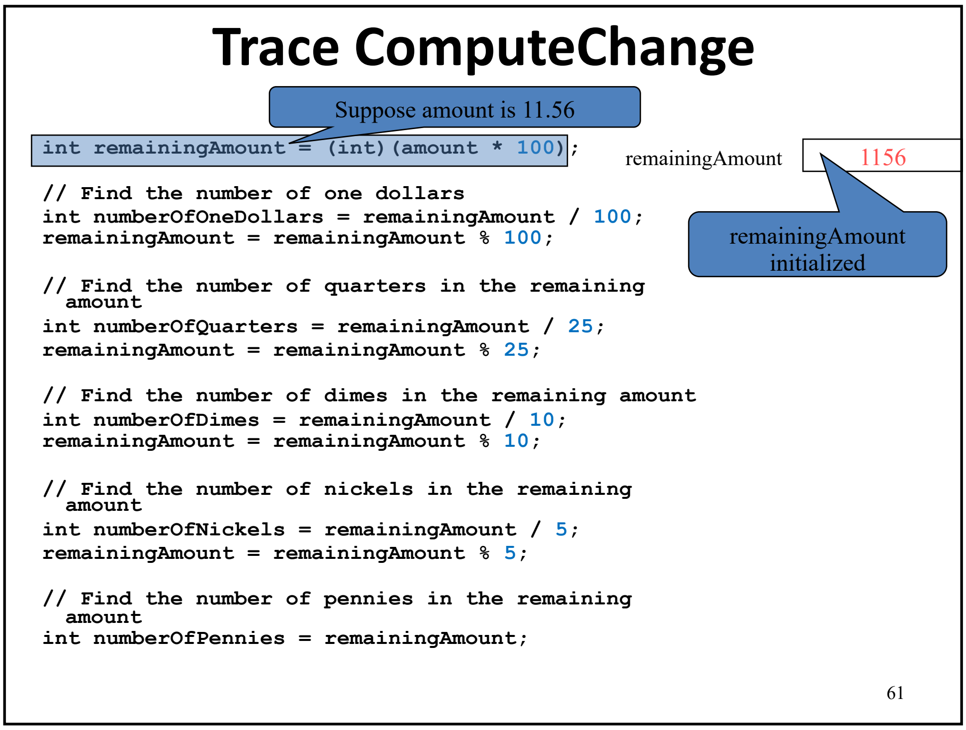
Dollar = 100 cents

Quarters = 25 cents

Dime = 10 cents

Nickel = 5 cents

ComputeChange Run



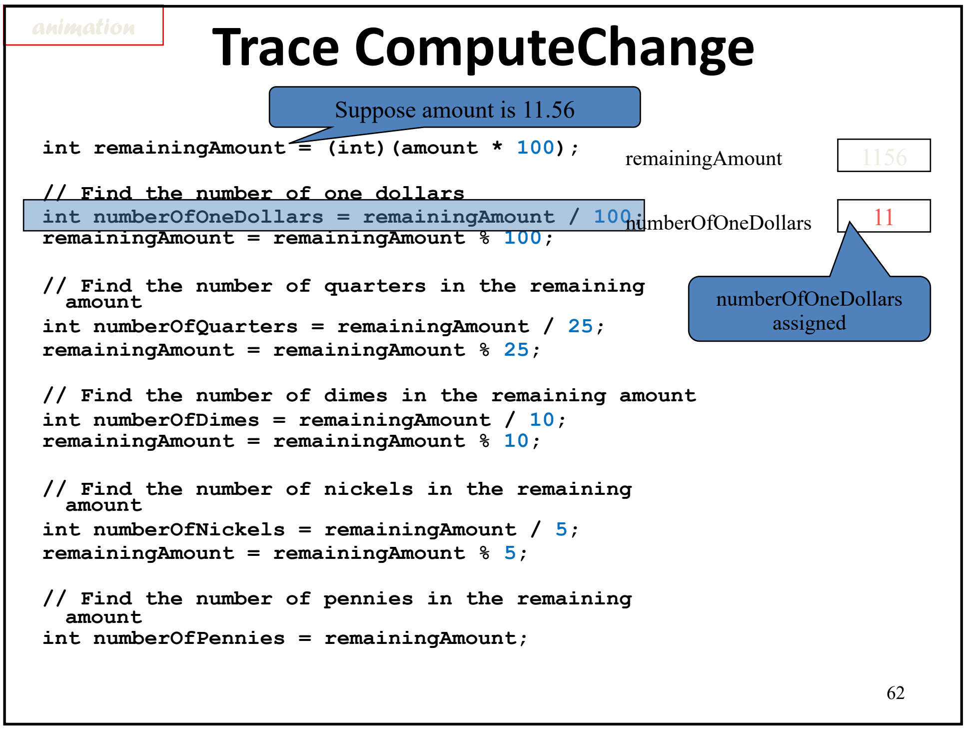
Trace ComputeChange

Suppose amount is 11.56

```
int remainingAmount = (int)(amount * 100);

// Find the number of one dollars
int numberOfOneDollars = remainingAmount / 100;
remainingAmount = remainingAmount % 100;

// Find the number of quarters in the remaining
  amount
int numberOfQuarters = remainingAmount / 25;
remainingAmount = remainingAmount % 25;

// Find the number of dimes in the remaining amount
int numberOfDimes = remainingAmount / 10;
remainingAmount = remainingAmount % 10;

// Find the number of nickels in the remaining
  amount
int numberOfNickels = remainingAmount / 5;
remainingAmount = remainingAmount % 5;

// Find the number of pennies in the remaining
  amount
int numberOfPennies = remainingAmount;
```

remainingAmount 1156

remainingAmount initialized



animation

Trace ComputeChange

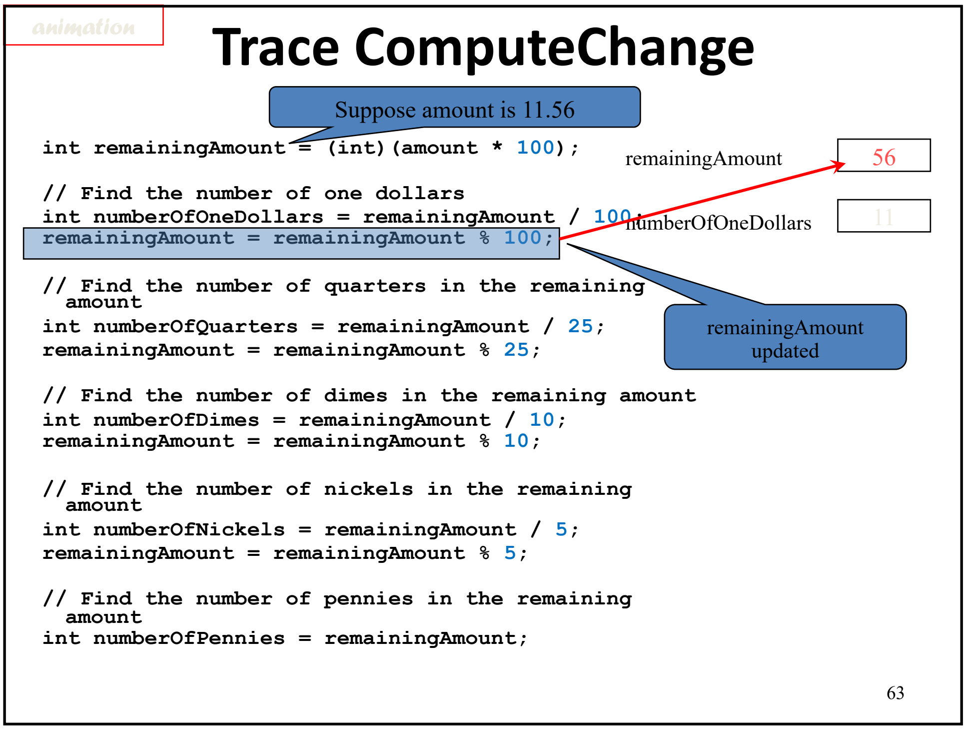
Suppose amount is 11.56

```
int remainingAmount = (int)(amount * 100);

// Find the number of one dollars
int numberOfOneDollars = remainingAmount / 100;
remainingAmount = remainingAmount % 100;

// Find the number of quarters in the remaining
  amount
int numberOfQuarters = remainingAmount / 25;
remainingAmount = remainingAmount % 25;

// Find the number of dimes in the remaining amount
int numberOfDimes = remainingAmount / 10;
remainingAmount = remainingAmount % 10;

// Find the number of nickels in the remaining
  amount
int numberOfNickels = remainingAmount / 5;
remainingAmount = remainingAmount % 5;

// Find the number of pennies in the remaining
  amount
int numberOfPennies = remainingAmount;
```

remainingAmount 1156

numberOfOneDollars 11

numberOfOneDollars assigned

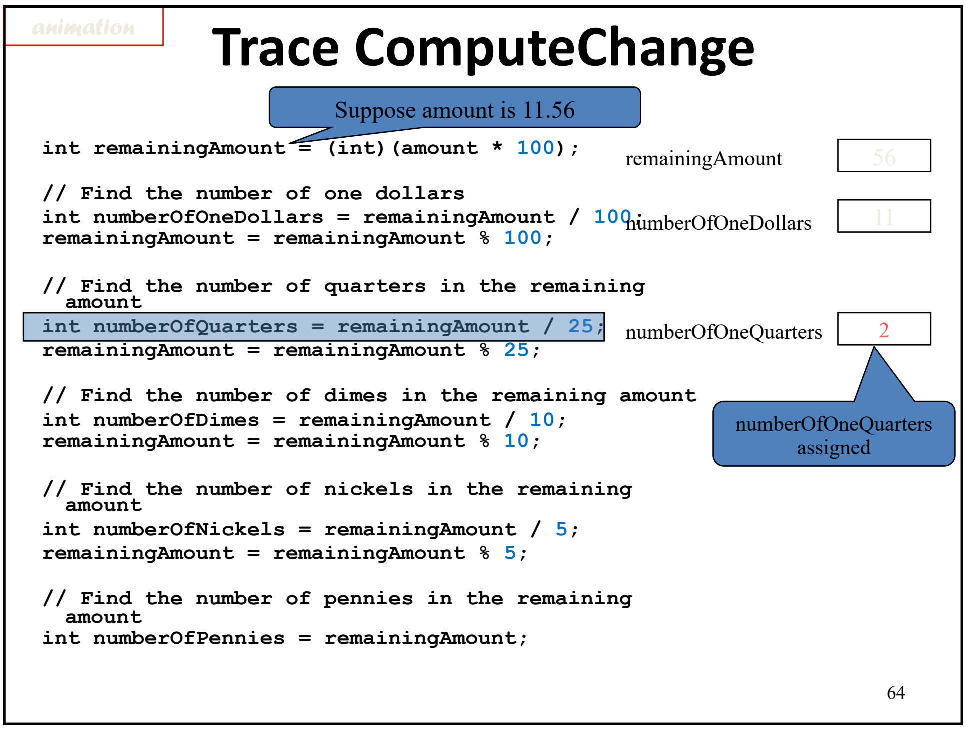


Trace ComputeChange

Suppose amount is 11.56

```
int remainingAmount = (int)(amount * 100);

// Find the number of one dollars
int numberOfOneDollars = remainingAmount / 100;
remainingAmount = remainingAmount % 100;

// Find the number of quarters in the remaining
  amount
int numberOfQuarters = remainingAmount / 25;
remainingAmount = remainingAmount % 25;

// Find the number of dimes in the remaining amount
int numberOfDimes = remainingAmount / 10;
remainingAmount = remainingAmount % 10;

// Find the number of nickels in the remaining
  amount
int numberOfNickels = remainingAmount / 5;
remainingAmount = remainingAmount % 5;

// Find the number of pennies in the remaining
  amount
int numberOfPennies = remainingAmount;
```

remainingAmount updated

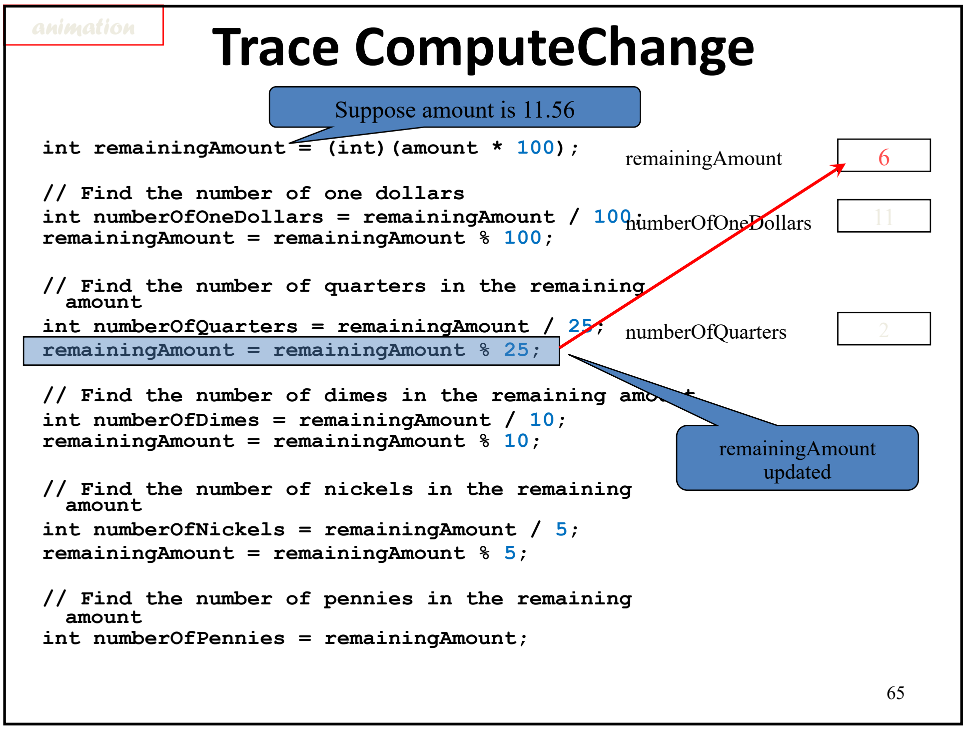


Trace ComputeChange

Suppose amount is 11.56

```
int remainingAmount = (int)(amount * 100);

// Find the number of one dollars
int numberOfOneDollars = remainingAmount / 100;
remainingAmount = remainingAmount % 100;

// Find the number of quarters in the remaining
  amount
int numberOfQuarters = remainingAmount / 25;
remainingAmount = remainingAmount % 25;

// Find the number of dimes in the remaining amount
int numberOfDimes = remainingAmount / 10;
remainingAmount = remainingAmount % 10;

// Find the number of nickels in the remaining
  amount
int numberOfNickels = remainingAmount / 5;
remainingAmount = remainingAmount % 5;

// Find the number of pennies in the remaining
  amount
int numberOfPennies = remainingAmount;
```

numberOfOneQuarters assigned



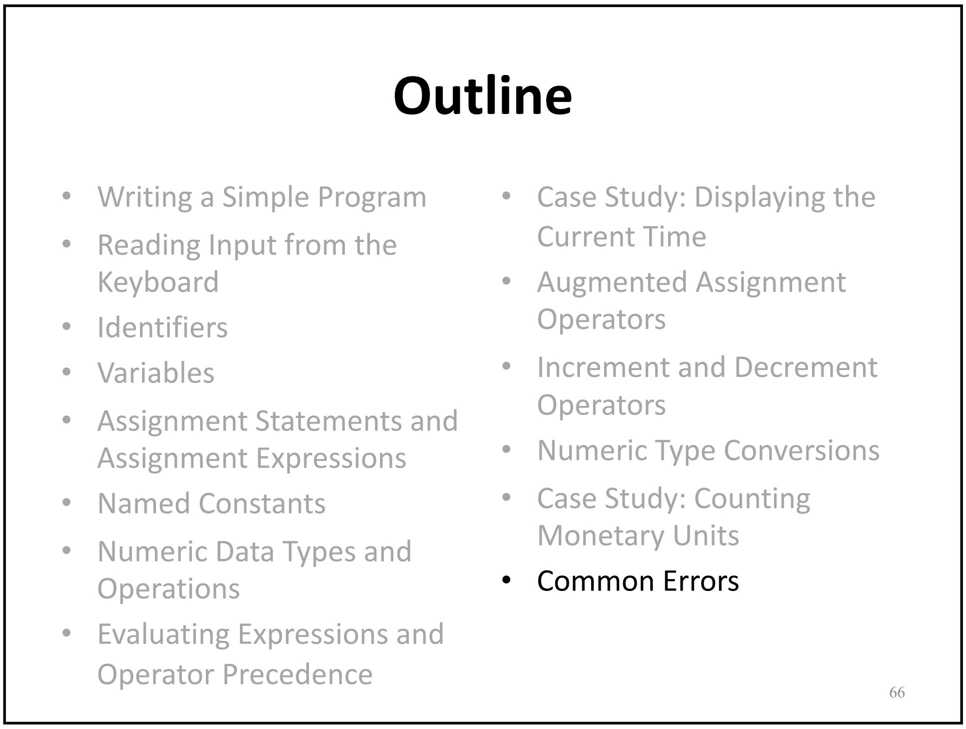
animation

Trace ComputeChange

Suppose amount is 11.56

```
int remainingAmount = (int)(amount * 100);

// Find the number of one dollars
int numberOfOneDollars = remainingAmount / 100;
remainingAmount = remainingAmount % 100;

// Find the number of quarters in the remaining
  amount
int numberOfQuarters = remainingAmount / 25;
remainingAmount = remainingAmount % 25;

// Find the number of dimes in the remaining amount
int numberOfDimes = remainingAmount / 10;
remainingAmount = remainingAmount % 10;

// Find the number of nickels in the remaining
  amount
int numberOfNickels = remainingAmount / 5;
remainingAmount = remainingAmount % 5;

// Find the number of pennies in the remaining
  amount
int numberOfPennies = remainingAmount;
```

remainingAmount updated



Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Common Errors

1. Undeclared or Uninitialized Variables

```
double interestRate = 0.05;
double interest = interestrate * 45;
```

2. Integer Overflow

```
short value = 32767 + 1; // is -32768
```

3. Round-off Errors

```
float a = 1000.43;
float b = 1000.0;
cout << a - b << endl;
```

displays `0.429993`, not `0.43`

Common Errors

4. Unintended Integer Division

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2;
cout << average << endl;
```

(a)

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2.0;
cout << average << endl;
```

(b)

(a) displays 1, (b) displays 1.5

5. Forgetting Header Files

```
#include <cmath>   // needed for pow()
#include <ctime>   // needed for time()
```

Outline

- Writing a Simple Program
- Reading Input from the Keyboard
- Identifiers
- Variables
- Assignment Statements and Assignment Expressions
- Named Constants
- Numeric Data Types and Operations
- Evaluating Expressions and Operator Precedence
- Case Study: Displaying the Current Time
- Augmented Assignment Operators
- Increment and Decrement Operators
- Numeric Type Conversions
- Case Study: Counting Monetary Units
- Common Errors

Chapter 3: Selections

Sections 3.1–3.16

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)

Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Introduction

If you assigned a negative value for **`radius`** in Listing 2.1, ComputeArea.cpp, the program would print an invalid result. If the radius is negative, you don't want the program to compute the area. How can you deal with this situation?

Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

The `bool` Type and Operators

Often in a program you need to compare two values, such as whether **`i`** is greater than **`j`**. C++ provides six *relational operators* (also known as *comparison operators*):

Operator	*Mathematics Symbol*	*Name*	*Example (radius is 5)*	*Result*
`<`	<	less than	`radius < 0`	`false`
`<=`	≤	less than or equal to	`radius <= 0`	`false`
`>`	>	greater than	`radius > 0`	`true`
`>=`	≥	greater than or equal to	`radius >= 0`	`true`
`==`	=	equal to	`radius == 0`	`false`
`!=`	≠	not equal to	`radius != 0`	`true`

The `bool` Type and Operators

A variable that holds a Boolean value is known as a *Boolean variable*, which holds **`true`** or **`false`**.

```
bool lightsOn = true;
cout << lightsOn; // Displays 1
cout << (4 < 5);  // Displays 1
cout << (4 > 5);  // Displays 0
```

Any nonzero value evaluates to **`true`** and zero value evaluates to **`false`**.

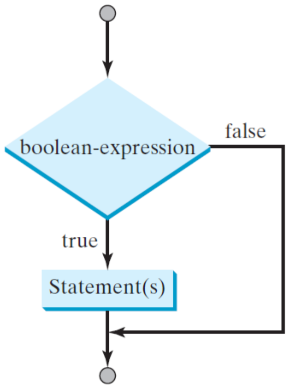
```
bool b1 = -1.5; // ≡ bool b1 = true;
bool b2 = 0;    // ≡ bool b2 = false;
bool b3 = 1.5;  // ≡ bool b3 = true;
```

Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

One-way `if` Statements

```
if (booleanExpression)
{
  statement(s);
}
```



```
if (radius >= 0)
{
  area = radius * radius * PI;
  cout << "The area for the circle of " <<
    " radius " << radius << " is " << area;
}
```

Notes

- The boolean-expression must be enclosed in parentheses.

```
if i > 0
{
  cout << "i is positive" << endl;
}
```

(a) Wrong

```
if (i > 0)
{
  cout << "i is positive" << endl;
}
```

(b) Correct

- The braces can be omitted if they enclose a single statement.

```
if (i > 0)
{
  cout << "i is positive" << endl;
}
```

(a)

Equivalent

```
if (i > 0)
  cout << "i is positive" << endl;
```

(b)

Simple if Demo

A program that prompts the user to enter an integer. If the number is a multiple of **5**, displays **HiFive**. If the number is even, displays **HiEven**.

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

int main()
{
  // Prompt the user to enter an integer
  int number;
  cout << "Enter an integer: ";
  cin >> number;

  if (number % 5 == 0)
    cout << "HiFive" << endl;

  if (number % 2 == 0)
    cout << "HiEven" << endl;

  return 0;
}
```

SimpleIfDemo

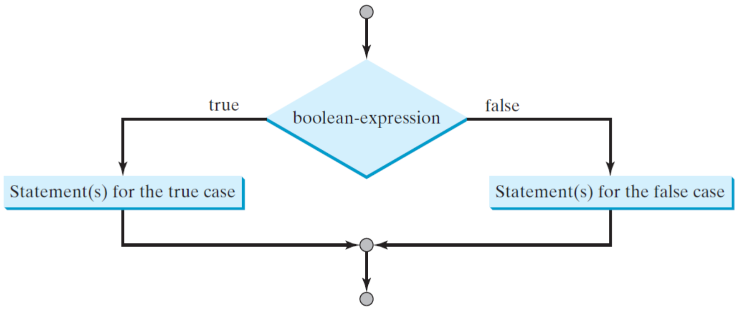
Run

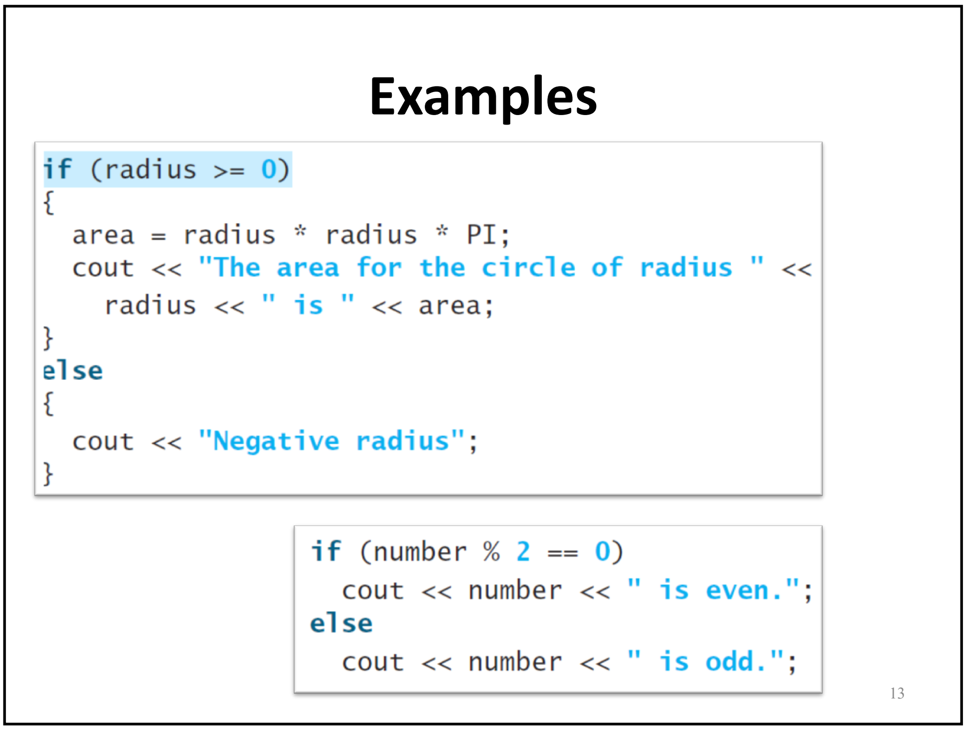
Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Two-Way `if-else` Statement

```
if (booleanExpression)
{
  statement(s)-for-the-true-case;
}
else
{
  statement(s)-for-the-false-case;
}
```

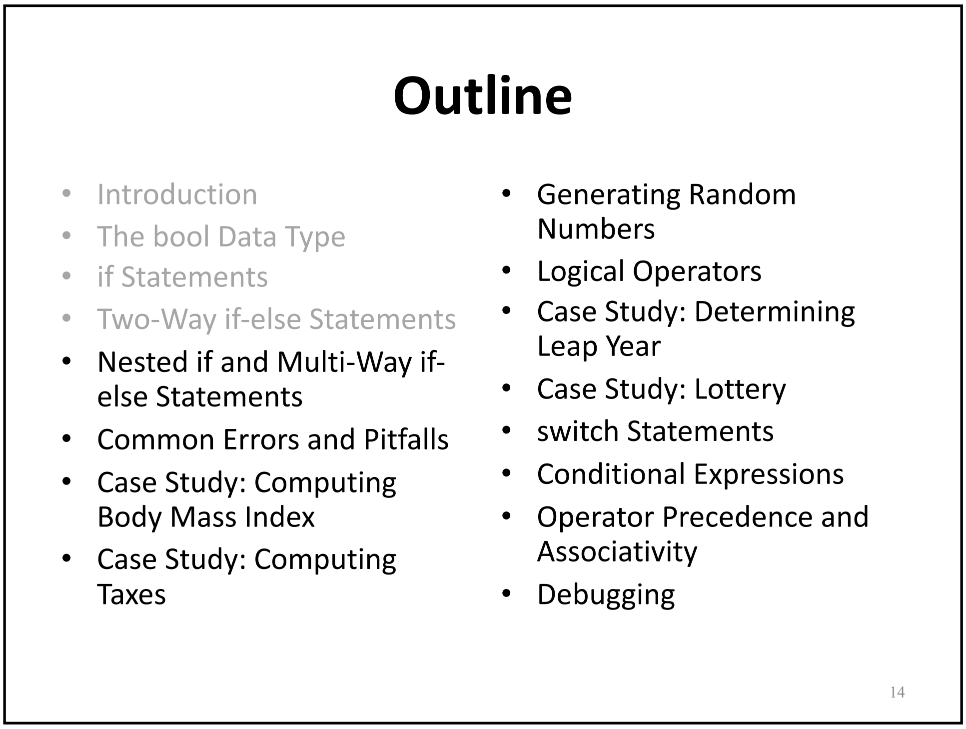




Examples

```
if (radius >= 0)
{
  area = radius * radius * PI;
  cout << "The area for the circle of radius " <<
    radius << " is " << area;
}
else
{
  cout << "Negative radius";
}
```

```
if (number % 2 == 0)
  cout << number << " is even.";
else
  cout << number << " is odd.";
```



Outline

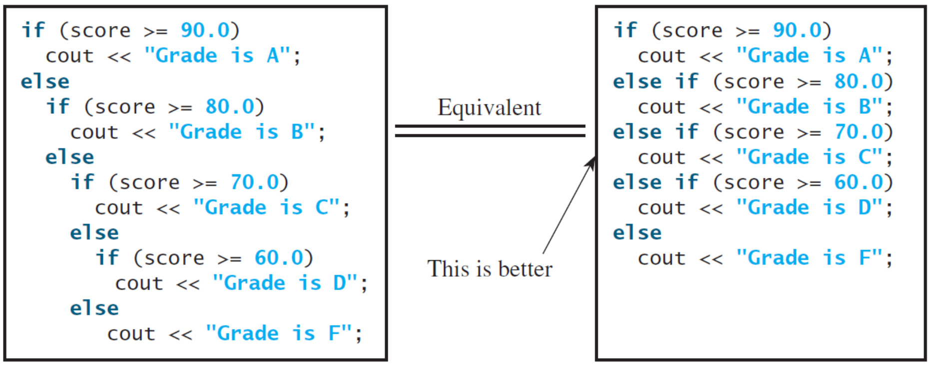
- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Nested `if` Statements

You can nest multiple `if` statements

```
if (i > k)
{
  if (j > k)
    cout << "i and j are greater than k";
}
else
  cout << "i is less than or equal to k";
```

Multiple Alternative `if` Statements



```
if (score >= 90.0)
  cout << "Grade is A";
else
  if (score >= 80.0)
    cout << "Grade is B";
  else
    if (score >= 70.0)
      cout << "Grade is C";
    else
      if (score >= 60.0)
        cout << "Grade is D";
    else
       cout << "Grade is F";
```

(a)

```
if (score >= 90.0)
  cout << "Grade is A";
else if (score >= 80.0)
  cout << "Grade is B";
else if (score >= 70.0)
  cout << "Grade is C";
else if (score >= 60.0)
  cout << "Grade is D";
else
  cout << "Grade is F";
```

(b)

Trace if-else statement

Suppose score is 70.0

The condition is false

```
if (score >= 90.0)
  cout << "Grade is A";
else if (score >= 80.0)
  cout << "Grade is B";
else if (score >= 70.0)
  cout << "Grade is C";
else if (score >= 60.0)
  cout << "Grade is D";
else
  cout << "Grade is F";
```

Trace if-else statement

Suppose score is 70.0

The condition is false

```
if (score >= 90.0)
  cout << "Grade is A";
else if (score >= 80.0)
  cout << "Grade is B";
else if (score >= 70.0)
  cout << "Grade is C";
else if (score >= 60.0)
  cout << "Grade is D";
else
  cout << "Grade is F";
```

animation

Trace if-else statement

Suppose score is 70.0

The condition is true

```
if (score >= 90.0)
  cout << "Grade is A";
else if (score >= 80.0)
  cout << "Grade is B";
else if (score >= 70.0)
  cout << "Grade is C";
else if (score >= 60.0)
  cout << "Grade is D";
else
  cout << "Grade is F";
```

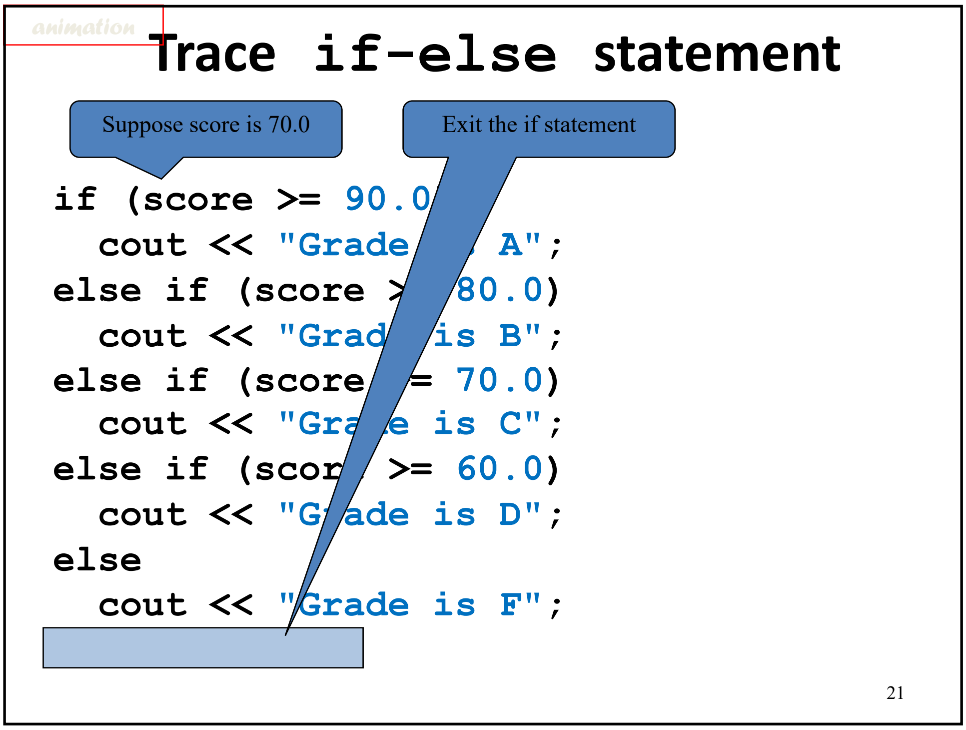
animation

Trace if-else statement

Suppose score is 70.0

grade is C

```
if (score >= 90.0)
  cout << "Grade is A";
else if (score >= 80.0)
  cout << "Grade is B";
else if (score >= 70.0)
  cout << "Grade is C";
else if (score >= 60.0)
  cout << "Grade is D";
else
  cout << "Grade is F";
```



Trace if-else statement

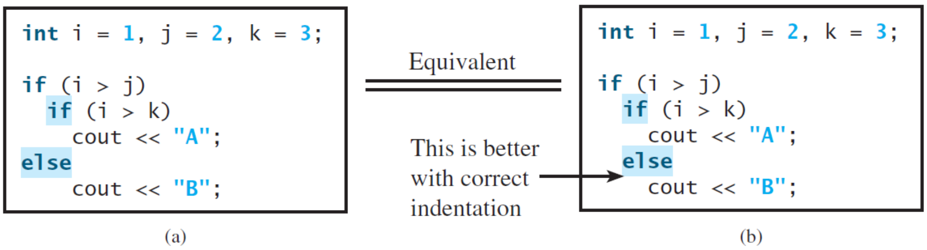
Suppose score is 70.0

Exit the if statement

```
if (score >= 90.0)
   cout << "Grade is A";
else if (score >= 80.0)
   cout << "Grade is B";
else if (score >= 70.0)
   cout << "Grade is C";
else if (score >= 60.0)
   cout << "Grade is D";
else
   cout << "Grade is F";
```

Note

The **`else`** clause matches the most recent **`if`** clause in the same block.



```
int i = 1, j = 2, k = 3;

if (i > j)
  if (i > k)
    cout << "A";
else
    cout << "B";
```

(a)

Equivalent

This is better with correct indentation

```
int i = 1, j = 2, k = 3;

if (i > j)
  if (i > k)
    cout << "A";
  else
    cout << "B";
```

(b)

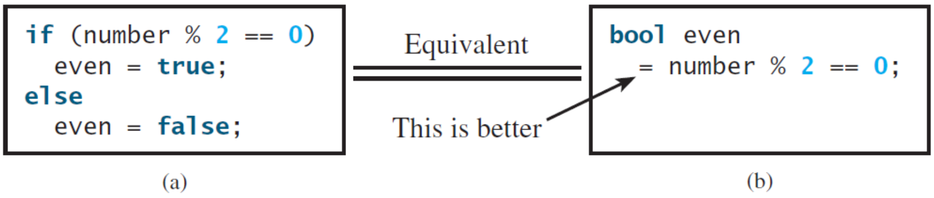
Note, cont.

Nothing is printed from the Statement (a) above. To force the **`else`** clause to match the first **`if`** clause, you must add a pair of braces:

```
int i = 1, j = 2, k = 3;
if (i > j)
{
  if (i > k)
    cout << "A";
}
else
  cout << "B";
```

This statement prints **B**.

TIP



```
if (number % 2 == 0)
  even = true;
else
  even = false;
```

(a)

```
bool even
  = number % 2 == 0;
```

(b)

Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Common Errors

1: Forgetting Necessary Braces

```
if (radius >= 0)
  area = radius * radius * PI;
  cout << "The area "
    << " is " << area;
```

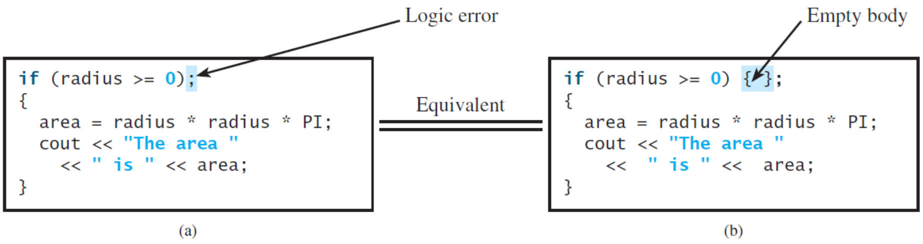
(a) Wrong

```
if (radius >= 0)
{
  area = radius * radius * PI;
  cout << "The area "
    <<  " is " <<  area;
}
```

(b) Correct

Common Errors

2: Wrong Semicolon at the if Line



```
if (radius >= 0);
{
  area = radius * radius * PI;
  cout << "The area "
    << " is " << area;
}
```

(a)

```
if (radius >= 0) { };
{
  area = radius * radius * PI;
  cout << "The area "
    <<  " is " <<  area;
}
```

(b)

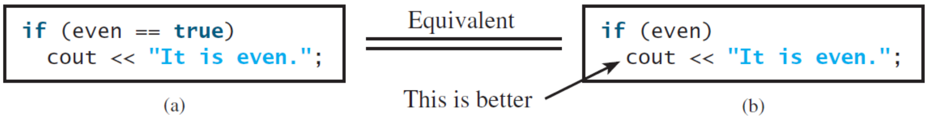
Common Errors

3: Mistakenly Using = for ==

```
if (count = 1)
   cout << "count is zero" << endl;
else
   cout << "count is not zero" << endl;
```

Common Errors

4: Redundant Testing of Boolean Values



```
if (even == true)
  cout << "It is even.";
```

(a)

Equivalent

This is better

```
if (even)
  cout << "It is even.";
```

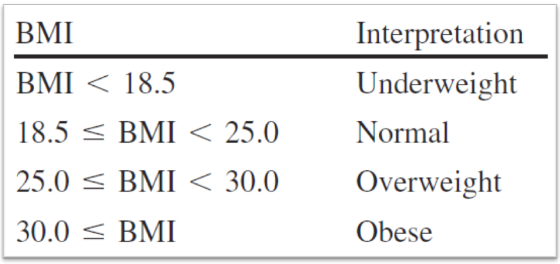
(b)

Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Case Study: Body Mass Index

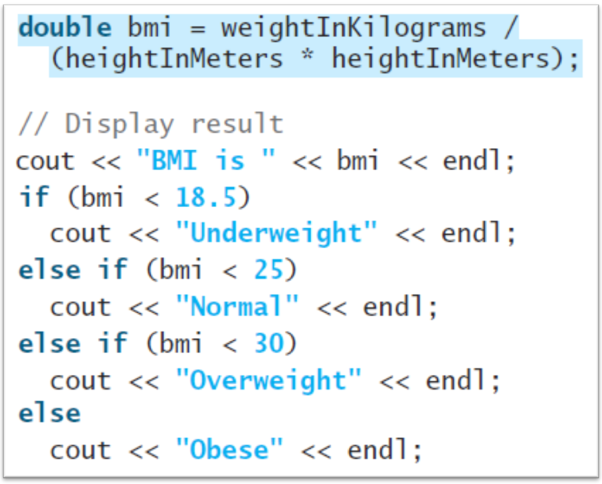
The Body Mass Index (BMI) is a measure of health on weight. It can be calculated by taking your weight in kilograms and dividing by the square of your height in meters ($BMI = {}^{m}/_{h^2}$). The interpretation of BMI for people 16 years or older is as follows:



BMI	Interpretation
BMI $<$ 18.5	Underweight
18.5 $\leq$ BMI $<$ 25.0	Normal
25.0 $\leq$ BMI $<$ 30.0	Overweight
30.0 $\leq$ BMI	Obese

ComputeBMI Run

Case Study: Body Mass Index



```
double bmi = weightInKilograms /
  (heightInMeters * heightInMeters);

// Display result
cout << "BMI is " << bmi << endl;
if (bmi < 18.5)
  cout << "Underweight" << endl;
else if (bmi < 25)
  cout << "Normal" << endl;
else if (bmi < 30)
  cout << "Overweight" << endl;
else
  cout << "Obese" << endl;
```

Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Case Study: Computing Taxes

The US federal personal income tax is calculated based on the filing status and taxable income. There are four filing statuses: single filers, married filing jointly, married filing separately, and head of household. The tax rates for 2002 are shown below.

Tax rate	Single filers	Married filing jointly or qualifying widow/widower	Married filing separately	Head of household
10%	Up to $6,000	Up to $12,000	Up to $6,000	Up to $10,000
15%	$6,001 - $27,950	$12,001 - $46,700	$6,001 - $23,350	$10,001 - $37,450
27%	$27,951 - $67,700	$46,701 - $112,850	$23,351 - $56,425	$37,451 - $96,700
30%	$67,701 - $141,250	$112,851 - $171,950	$56,426 - $85,975	$96,701 - $156,600
35%	$141,251- $307,050	$171,951 - $307,050	$85,976 - $153,525	$156,601 - $307,050
38.6%	$307,051 or more	$307,051 or more	$153,526 or more	$307,051 or more

Computing Taxes: Skeleton Code

```
if (status == 0)
{
  // Compute tax for single filers
}
else if (status == 1)
{
  // Compute tax for married file jointly
}
else if (status == 2)
{
  // Compute tax for married file separately
}
else if (status == 3)
{
  // Compute tax for head of household
}
else
{
  // Display wrong status
}
```

ComputeTax Run

Computing Taxes: First Case Details

```
if (status == 0)
  {
    // Compute tax for single filers
    if (income <= 6000)
      tax = income * 0.10;
    else if (income <= 27950)
      tax = 6000 * 0.10 + (income - 6000) * 0.15;
    else if (income <= 67700)
      tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
            (income - 27950) * 0.27;
    else if (income <= 141250)
      …
  }
  else if (status == 1)
```

Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Generating Random Numbers

- You can use the **`rand()`** function to obtain a random integer.
- This function returns a random integer between 0 and RAND_MAX (32,767 in Visual C++).
- To start with a different seed at each execution, use

```
srand(time(0));
```

- To obtain a random integer between 0 and 9, use

```
rand() % 10
```

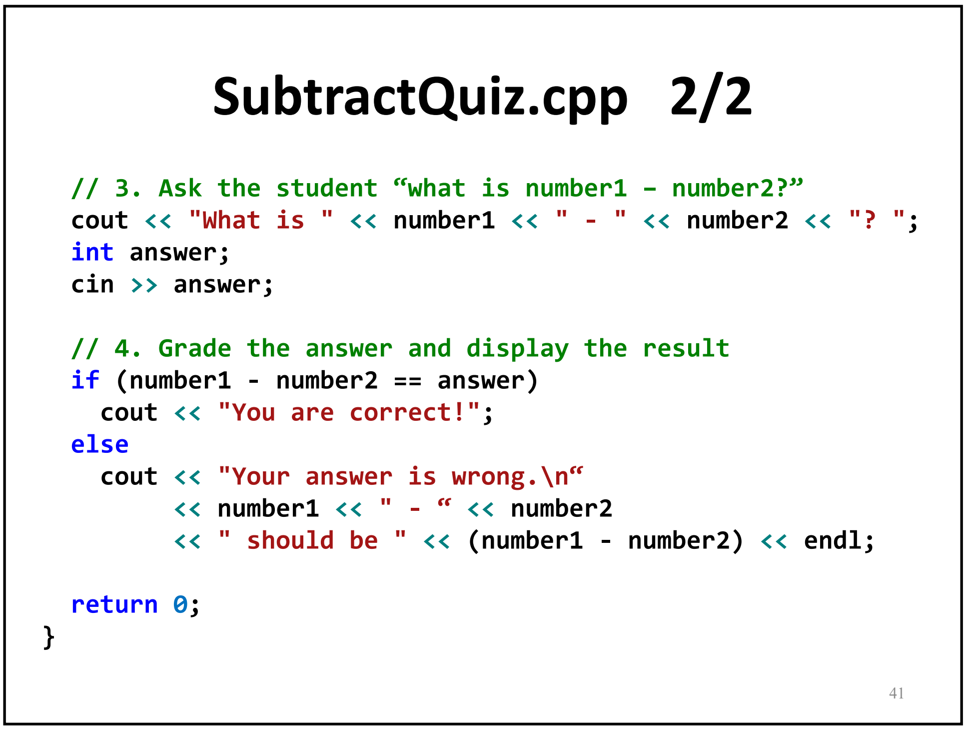
Example: A Simple Math Learning Tool

- This example creates a program for a first grader to practice subtractions.
- The program randomly generates two single-digit integers **`number1`** and **`number2`** with **`number1 >= number2`** and displays a question such as **"What is 9 – 2?"** to the student.
- After the student types the answer, the program displays a message to indicate whether the answer is correct.

SubtractionQuiz Run

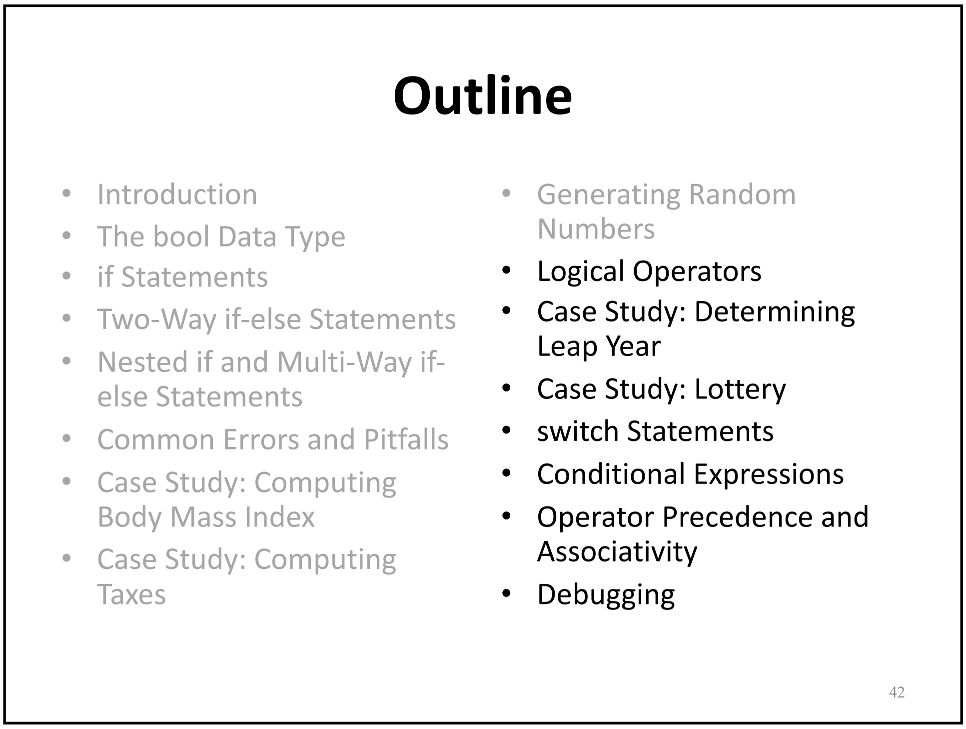
SubtractQuiz.cpp 1/2

```
#include <iostream>
#include <ctime> // for time function
#include <cstdlib> // for rand and srand functions
using namespace std;

int main()
{
  // 1. Generate two random single-digit integers
  srand(time(0));
  int number1 = rand() % 10;
  int number2 = rand() % 10;

  // 2. If number1 < number2, swap number1 with number2
  if (number1 < number2)
  {
    int temp = number1;
    number1 = number2;
    number2 = temp;
  }
```



SubtractQuiz.cpp 2/2

```
  // 3. Ask the student “what is number1 – number2?”
  cout << "What is " << number1 << " - " << number2 << "? ";
  int answer;
  cin >> answer;

  // 4. Grade the answer and display the result
  if (number1 - number2 == answer)
    cout << "You are correct!";
  else
    cout << "Your answer is wrong.\n“
         << number1 << " - “ << number2
         << " should be " << (number1 - number2) << endl;

  return 0;
}
```



Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Logical Operators

- The logical operators **`!`**, **`&&`**, and **`||`** can be used to create a compound Boolean expression.

TABLE 3.3 Boolean Operators

Operator	*Name*	*Description*
`!`	not	logical negation
`&&`	and	logical conjunction
`||`	or	logical disjunction

TABLE 3.4 Truth Table for Operator `!`

p	*!p*	*Example (assume age = 24, weight = 140)*
`true`	`false`	`!(age > 18)` is `false`, because `(age > 18)` is `true`.
`false`	`true`	`!(weight == 150)` is `true`, because `(weight == 150)` is `false`.

TABLE 3.5 Truth Table for Operator `&&`

p1	*p2*	*p1 && p2*	*Example (assume age = 24, weight = 140)*
`false`	`false`	`false`	`(age > 18) && (weight <= 140)` is `true`, because `(age > 18)` and `(weight <= 140)` are both `true`.
`false`	`true`	`false`	
`true`	`false`	`false`	`(age > 18) && (weight > 140)` is `false`, because `(weight > 140)` is `false`.
`true`	`true`	`true`	

TABLE 3.6 Truth Table for Operator `||`

p1	*p2*	*p1 || p2*	*Example (assume age = 24, weight = 140)*
`false`	`false`	`false`	`(age > 34) || (weight <= 140)` is `true`, because `(weight <= 140)` is `true`.
`false`	`true`	`true`	
`true`	`false`	`true`	`(age > 34) || (weight >= 150)` is `false`, because `(age > 34)` and `(weight >= 150)` are both `false`.
`true`	`true`	`true`	

Examples

A program that checks whether a number is divisible by 2 and 3, whether a number is divisible by 2 or 3, and whether a number is divisible by 2 or 3 but not both:

TestBooleanOperators

Run

TestBooleanOperators.cpp

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

int main()
{
  int number;
  cout << "Enter an integer: ";
  cin >> number;

  if (number % 2 == 0 && number % 3 == 0)
    cout << number << " is divisible by 2 and 3." << endl;
  if (number % 2 == 0 || number % 3 == 0)
    cout << number << " is divisible by 2 or 3." << endl;
  if ((number % 2 == 0 || number % 3 == 0) &&
       !(number % 2 == 0 && number % 3 == 0))
    cout << number << " divisible by 2 or 3, but not both." << endl;

  return(0);
}
```

Short-Circuit Operator

- When evaluating **`p1 && p2`**, C++ first evaluates **`p1`** and then evaluates **`p2`** if **`p1`** is **`true`**; if **`p1`** is **`false`**, it does not evaluate **`p2`**.
- When evaluating **`p1 || p2`**, C++ first evaluates **`p1`** and then evaluates **`p2`** if **`p1`** is **`false`**; if **`p1`** is **`true`**, it does not evaluate **`p2`**.
- Therefore, **`&&`** is referred to as the *conditional* or *short-circuit AND* operator, and **`||`** is referred to as the *conditional* or *short-circuit OR* operator.

Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Case Study: Determining Leap Year

A program that lets the user enter a year and checks whether it is a leap year.

A year is a *leap year* if it is divisible by 4 but not by 100 or if it is divisible by 400. So you can use the following Boolean expression to check whether a year is a leap year:

```
(year % 4 == 0 && year % 100 != 0) ||
(year % 400 == 0)
```

LeapYear Run

Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Case Study: Lottery

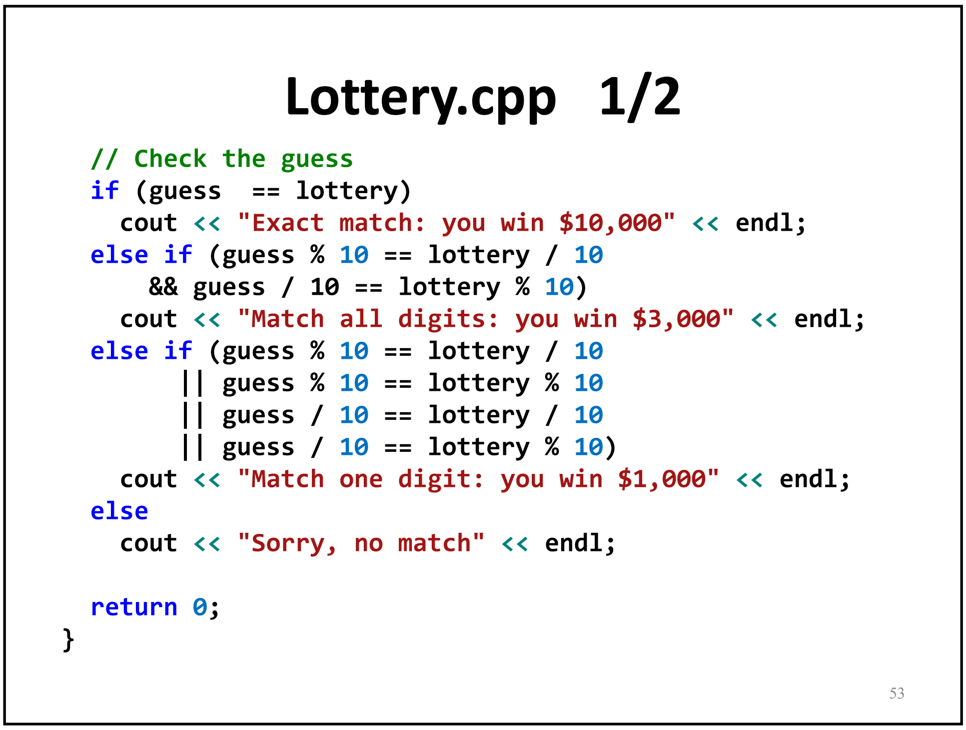
Randomly generates a lottery of a two-digit number, prompts the user to enter a two-digit number, and determines whether the user wins according to the following rule:

- If the user input matches the lottery in exact order, the award is $10,000.
- If the user input matches the lottery, the award is $3,000.
- If one digit in the user input matches a digit in the lottery, the award is $1,000.

Lottery Run

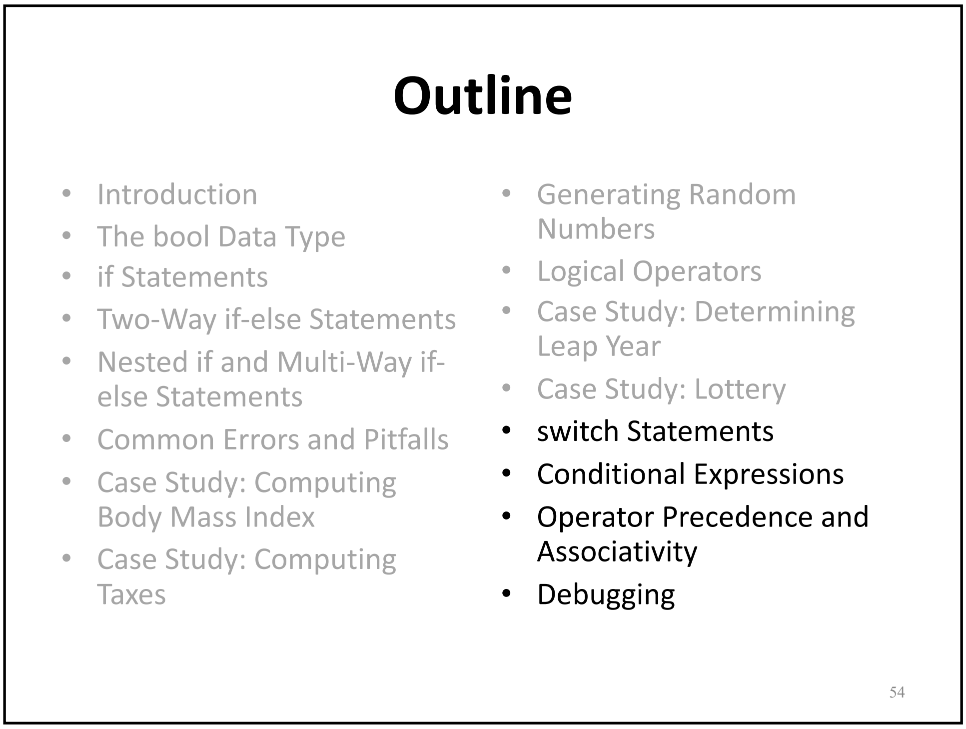
Lottery.cpp 1/2

```
#include <iostream>
#include <ctime>   // for time function
#include <cstdlib> // for rand and srand functions
using namespace std;

int main()
{
  // Generate a lottery
  srand(time(0));
  int lottery = rand() % 100;

  // Prompt the user to enter a guess
  cout << "Enter your lottery pick (two digits): ";
  int guess;
  cin >> guess;
```



Lottery.cpp 1/2

```
  // Check the guess
  if (guess  == lottery)
    cout << "Exact match: you win $10,000" << endl;
  else if (guess % 10 == lottery / 10
      && guess / 10 == lottery % 10)
    cout << "Match all digits: you win $3,000" << endl;
  else if (guess % 10 == lottery / 10
        || guess % 10 == lottery % 10
        || guess / 10 == lottery / 10
        || guess / 10 == lottery % 10)
    cout << "Match one digit: you win $1,000" << endl;
  else
    cout << "Sorry, no match" << endl;

  return 0;
}
```



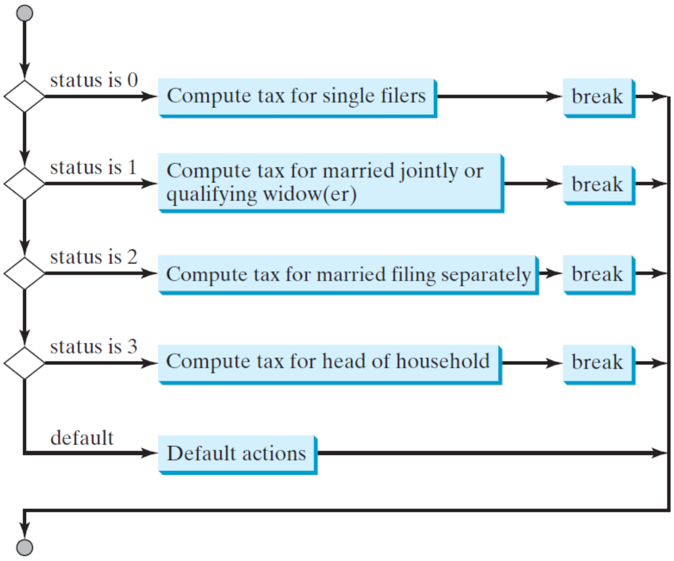
Outline

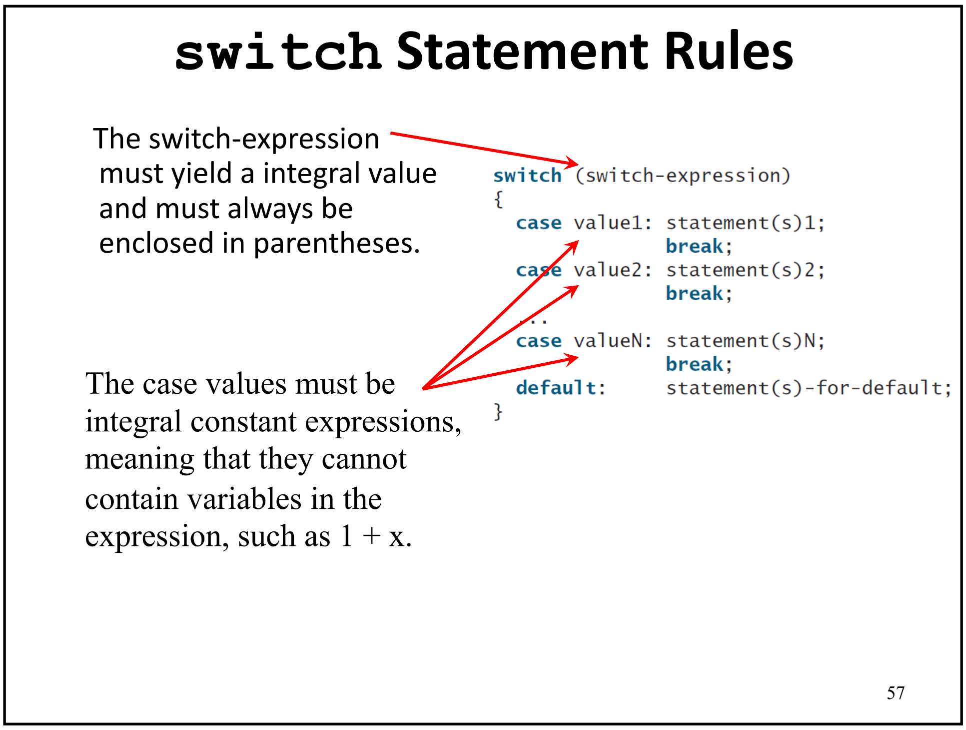
- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

switch Statements

```
switch (status)
{
  case 0:  compute taxes for single filers;
           break;
  case 1:  compute taxes for married file jointly;
           break;
  case 2:  compute taxes for married file separately;
           break;
  case 3:  compute taxes for head of household;
           break;
  default: cout << "Errors: invalid status" << endl;
}
```

switch Statement Flow Chart



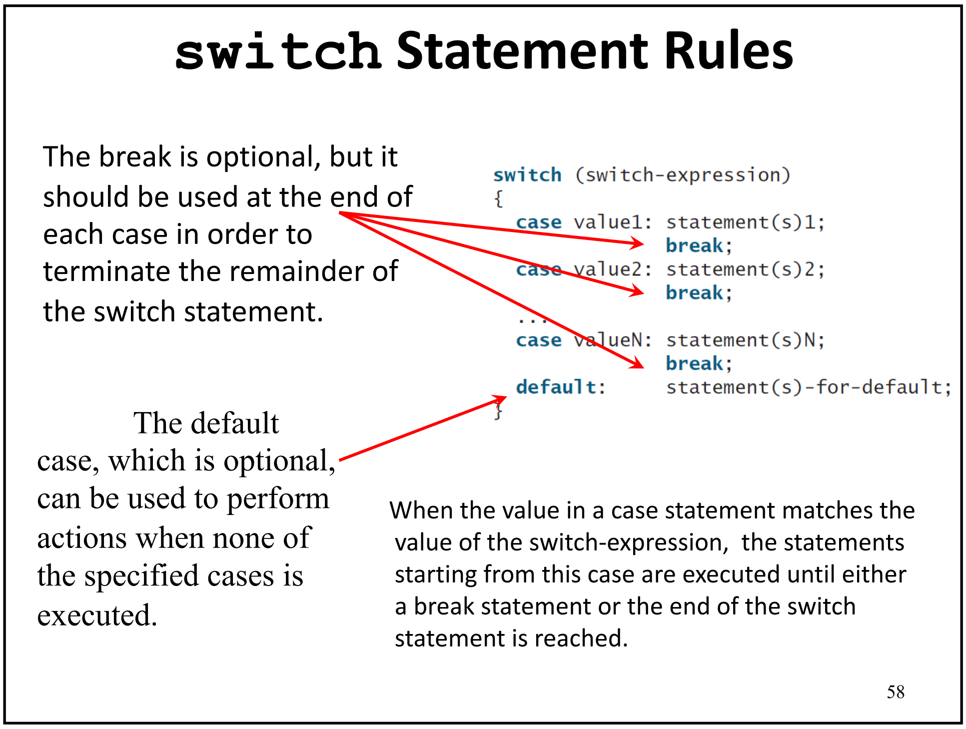


`switch` Statement Rules

The switch-expression must yield a integral value and must always be enclosed in parentheses.

```
switch (switch-expression)
{
  case value1: statement(s)1;
               break;
  case value2: statement(s)2;
               break;
  ...
  case valueN: statement(s)N;
               break;
  default:     statement(s)-for-default;
}
```

The case values must be integral constant expressions, meaning that they cannot contain variables in the expression, such as 1 + x.



`switch` Statement Rules

The break is optional, but it should be used at the end of each case in order to terminate the remainder of the switch statement.

```
switch (switch-expression)
{
  case value1: statement(s)1;
               break;
  case value2: statement(s)2;
               break;
  ...
  case valueN: statement(s)N;
               break;
  default:     statement(s)-for-default;
}
```

The default case, which is optional, can be used to perform actions when none of the specified cases is executed.

When the value in a case statement matches the value of the switch-expression, the statements starting from this case are executed until either a break statement or the end of the switch statement is reached.

animation

Trace `switch` statement

Suppose day is 3:

```
switch (day)
{
  case 1: // Fall to through to the next case
  case 2: // Fall to through to the next case
  case 3: // Fall to through to the next case
  case 4: // Fall to through to the next case
  case 5: cout << "Weekday"; break;
  case 0: // Fall to through to the next case
  case 6: cout << "Weekend";
}
```

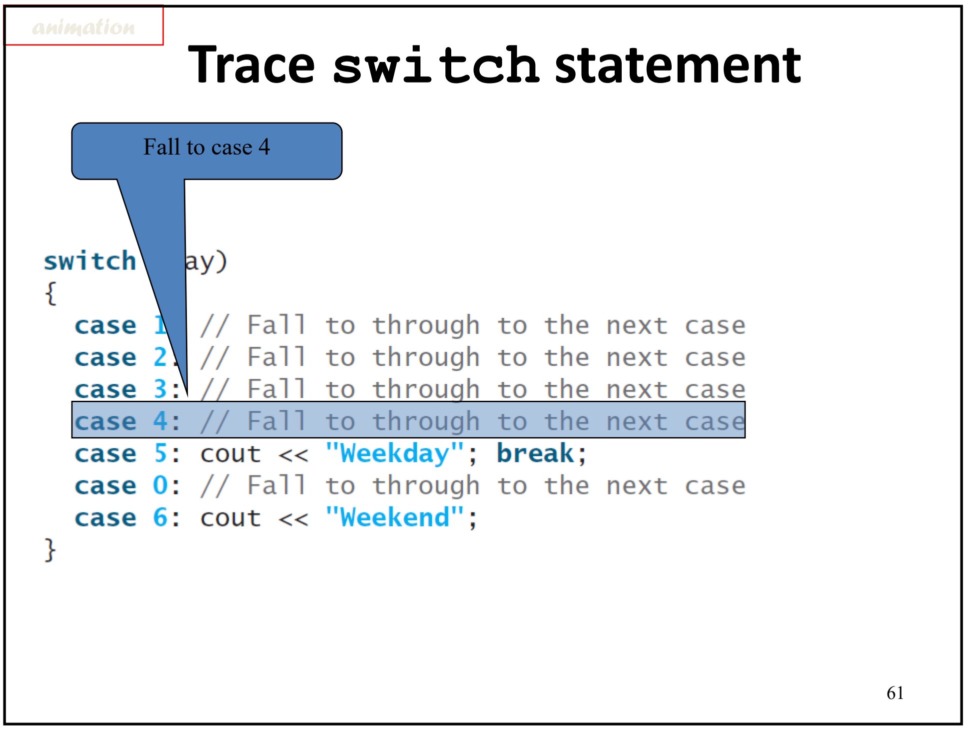
---

animation

Trace `switch` statement

Execute case 3

```
switch     ay)
{
  case 1  // Fall to through to the next case
  case 2: // Fall to through to the next case
  case 3: // Fall to through to the next case
  case 4: // Fall to through to the next case
  case 5: cout << "Weekday"; break;
  case 0: // Fall to through to the next case
  case 6: cout << "Weekend";
}
```

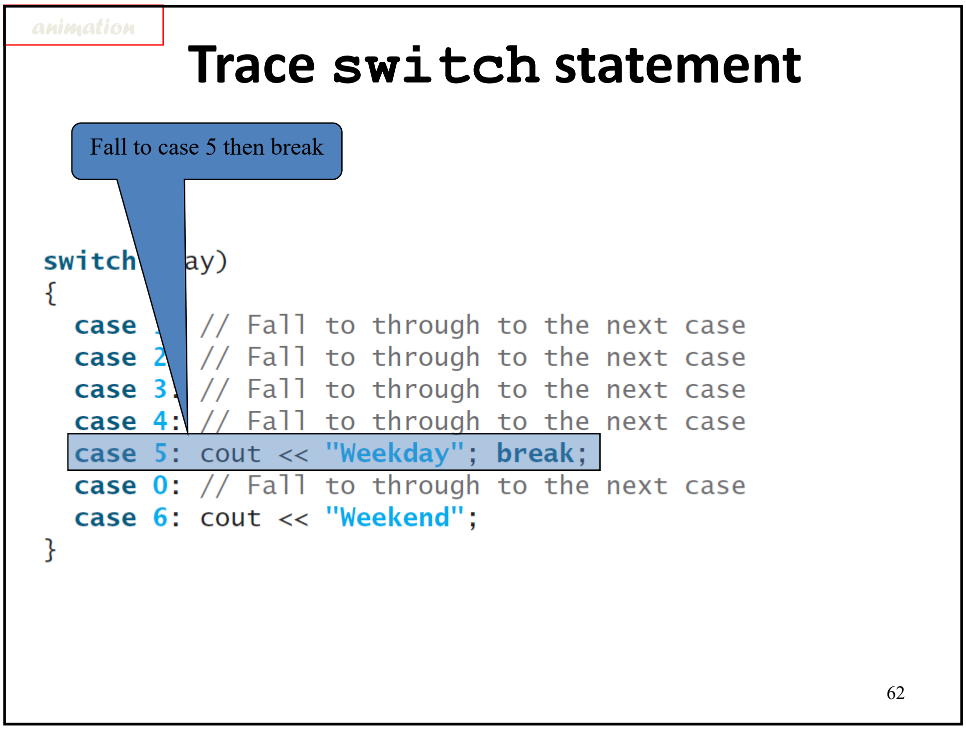


animation

Trace `switch` statement

Fall to case 4

```
switch    ay)
{
  case 1  // Fall to through to the next case
  case 2  // Fall to through to the next case
  case 3: // Fall to through to the next case
  case 4: // Fall to through to the next case
  case 5: cout << "Weekday"; break;
  case 0: // Fall to through to the next case
  case 6: cout << "Weekend";
}
```



animation

Trace `switch` statement

Fall to case 5 then break

```
switch    ay)
{
  case    // Fall to through to the next case
  case 2  // Fall to through to the next case
  case 3  // Fall to through to the next case
  case 4: // Fall to through to the next case
  case 5: cout << "Weekday"; break;
  case 0: // Fall to through to the next case
  case 6: cout << "Weekend";
}
```

animation

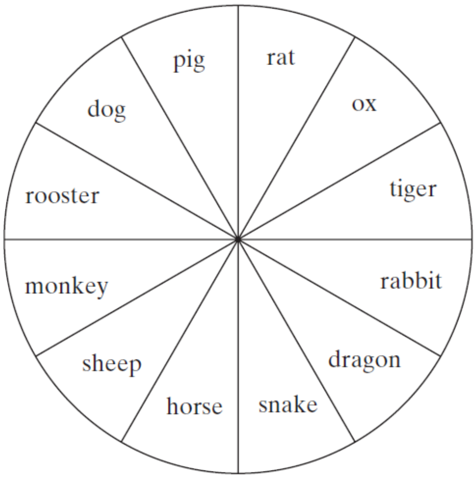
Trace `switch` statement

Execute what is next

```
switc    day)
{
  cas   1: // Fall to through to the next case
  cas   2: // Fall to through to the next case
  cas   3: // Fall to through to the next case
  cas   4: // Fall to through to the next case
  ca  e 5: cout << "Weekday"; break;
  ca  e 0: // Fall to through to the next case
  ca  e 6: cout << "Weekend";
}
```

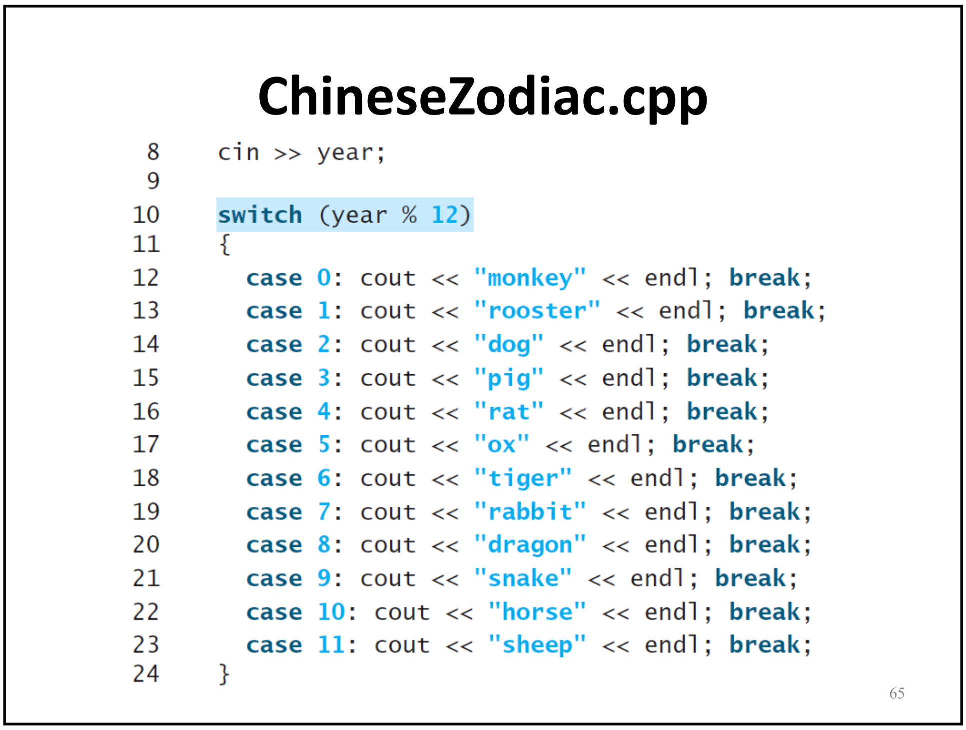
Example: Chinese Zodiac

A program that prompts the user to enter a year and displays the animal for the year.



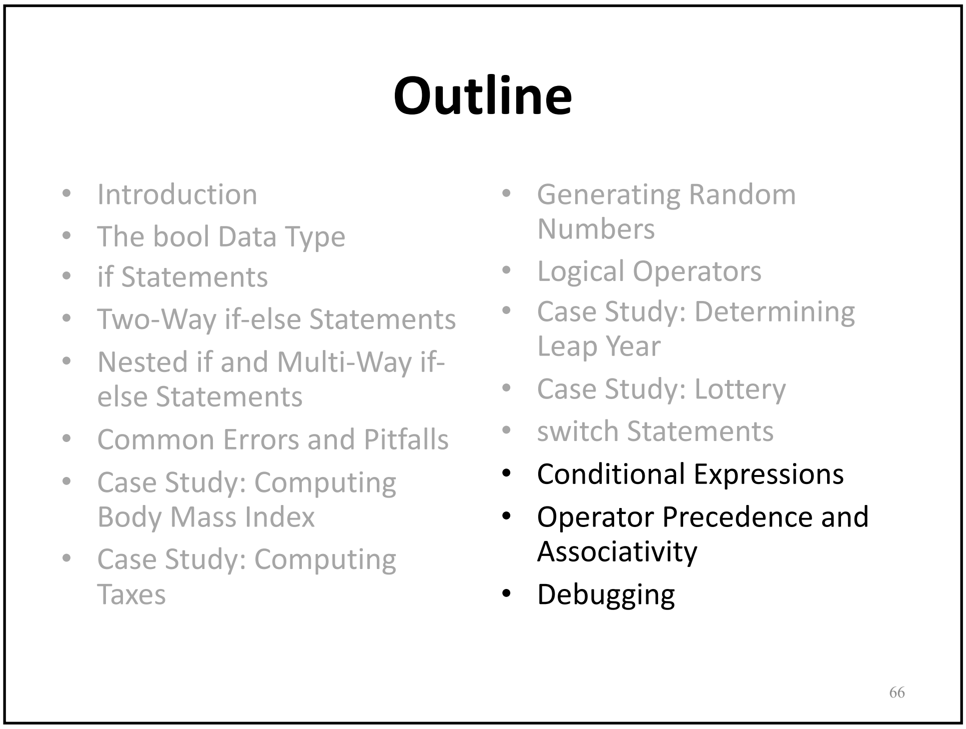
year % 12 =
0: monkey
1: rooster
2: dog
3: pig
4: rat
5: ox
6: tiger
7: rabbit
8: dragon
9: snake
10: horse
11: sheep

ChineseZodiac Run



ChineseZodiac.cpp

```
  cin >> year;

  switch (year % 12)
  {
    case 0: cout << "monkey" << endl; break;
    case 1: cout << "rooster" << endl; break;
    case 2: cout << "dog" << endl; break;
    case 3: cout << "pig" << endl; break;
    case 4: cout << "rat" << endl; break;
    case 5: cout << "ox" << endl; break;
    case 6: cout << "tiger" << endl; break;
    case 7: cout << "rabbit" << endl; break;
    case 8: cout << "dragon" << endl; break;
    case 9: cout << "snake" << endl; break;
    case 10: cout << "horse" << endl; break;
    case 11: cout << "sheep" << endl; break;
  }
```



Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Conditional Expressions

A conditional expression evaluates an expression based on a condition.

Syntax:
(booleanExpression) ? expression1 : expression2

The result of this conditional expression is expression1 if boolean-expression is true; otherwise, the result is expression2.

Examples

- Equivalent statements:

```
if (x > 0)
  y = 1;
else
  y = -1;
```

≡

```
y = x > 0 ? 1 : -1;
```

- Finding the max:

```
max = num1 > num2 ? num1 : num2;
```

- Odd of even:

```
cout << (num % 2 == 0 ? "num is even" : "num is odd") << endl;
```

Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Operator Precedence and Associativity

Operator precedence and associativity determine the order in which operators are evaluated.

How to evaluate 3 + 4 * 4 > 5 * (4 + 3) – 1?

false?

3 + 4 * 4 > 5 * (4 + 3) – 1 && (4 – 3 > 5)?

false?

Operator Precedence

Precedence	*Operator*
↓	`var++` and `var--` (Postfix)
	`+`, `-` (Unary plus and minus), `++var` and `--var` (Prefix)
	`static_cast`<type>(v), (type) (Casting)
	`!` (Not)
	`*`, `/`, `%` (Multiplication, division, and remainder)
	`+`, `-` (Binary addition and subtraction)
	`<`, `<=`, `>`, `>=` (Relational)
	`==`, `!=` (Equality)
	`&&` (AND)
	`||` (OR)
	`=`, `+=`, `-=`, `*=`, `/=`, `%=` (Assignment operator)

Operator Associativity

- All binary operators except assignment operators are *left associative*.
- Assignment operators are *right associative*.

`a - b + c - d` is equivalent to `((a - b) + c) - d`

`a = b += c = 5` is equivalent to `a = (b += (c = 5))`

Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Debugging

- Debugging is the process of finding and fixing errors in a program.
- Visual Studio supports debugging:
  - Executing a single statement at a time
  - Tracing into or stepping over a function
  - Setting breakpoints
  - Displaying variables
  - Displaying call stacks
  - Modifying variables
- Show demo on Visual Studio 2019.

Outline

- Introduction
- The bool Data Type
- if Statements
- Two-Way if-else Statements
- Nested if and Multi-Way if-else Statements
- Common Errors and Pitfalls
- Case Study: Computing Body Mass Index
- Case Study: Computing Taxes
- Generating Random Numbers
- Logical Operators
- Case Study: Determining Leap Year
- Case Study: Lottery
- switch Statements
- Conditional Expressions
- Operator Precedence and Associativity
- Debugging

Chapter 4: Mathematical Functions, Characters, and Strings

Sections 4.1–4.11

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

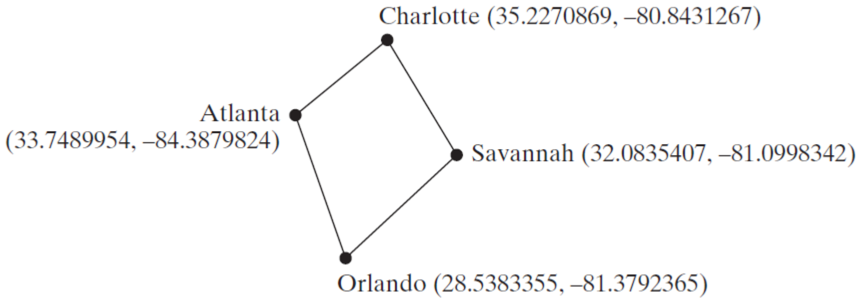
These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)

Outline

- Introduction
- Mathematical Functions
- Character Data Type and Operations
- Case Study: Generating Random Characters
- Case Study: Guessing Birthdays
- Character Functions
- Case Study: Converting Hexadecimal Decimal
- The string Type
- Case Study: Revising the Lottery Program Using Strings
- Formatting Console Output
- Simple File Input and Output

Introduction

Suppose you need to estimate the area enclosed by four cities, given the GPS locations (latitude and longitude) of these cities, as shown in the following diagram. How would you write a program to solve this problem? You will be able to write such a program after completing this chapter.



Outline

- Introduction
- Mathematical Functions
- Character Data Type and Operations
- Case Study: Generating Random Characters
- Case Study: Guessing Birthdays
- Character Functions
- Case Study: Converting Hexadecimal Decimal
- The string Type
- Case Study: Revising the Lottery Program Using Strings
- Formatting Console Output
- Simple File Input and Output

Mathematical Functions

C++ provides many useful functions in the **cmath** header for performing common mathematical functions.

1. Trigonometric functions
2. Exponent functions
3. Service functions

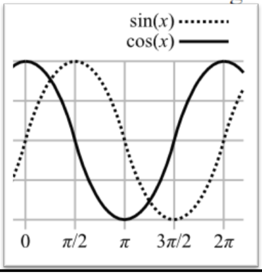
To use them, you need to include:

```
#include <cmath>
```

Trigonometric Functions

Function	*Description*
`sin(radians)`	Returns the trigonometric sine of an angle in radians.
`cos(radians)`	Returns the trigonometric cosine of an angle in radians
`tan(radians)`	Returns the trigonometric tangent of an angle in radians.
`asin(a)`	Returns the angle in radians for the inverse of sine.
`acos(a)`	Returns the angle in radians for the inverse of cosine.
`atan(a)`	Returns the angle in radians for the inverse of tangent.

`sin(0)` returns `0.0`
`sin(PI / 2)` returns `1.0`
`cos(0)` returns `1.0`
`atan(1.0)` returns `0.785398` (same as π/4)



Exponent Functions

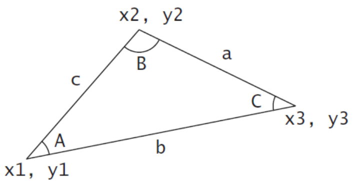
Function	*Description*
`exp(x)`	Returns e raised to power of x (e^x).
`log(x)`	Returns the natural logarithm of x ($\ln(x) = \log_e(x)$).
`log10(x)`	Returns the base 10 logarithm of x ($\log_{10}(x)$).
`pow(a, b)`	Returns a raised to the power of b (a^b).
`sqrt(x)`	Returns the square root of x ($\sqrt{x}$) for x >= 0.

```
exp(1.0) returns 2.71828
log(E) returns 1.0
log10(10.0) returns 1.0
pow(2.0, 3) returns 8.0
sqrt(4.0) returns 2.0
sqrt(10.5) returns 3.24
```

Service Functions

Function	Description	Example
`ceil(x)`	x is rounded up to its nearest integer. This integer is returned as a double value.	ceil(2.1) returns 3.0 ceil(-2.1) returns -2.0
`floor(x)`	x is rounded down to its nearest integer. This integer is returned as a double value.	floor(2.1) returns 2.0 floor(-2.1) returns -3.0
`min(x, y)`	Returns the minimum of x and y.	max(2, 3) returns 3
`max(x, y)`	Returns the maximum of x and y.	min(2.5, 4.6) returns 2.5
`abs(x)`	Returns the absolute value of x.	abs(-2.1) returns 2.1

Case Study: Computing Angles of a Triangle



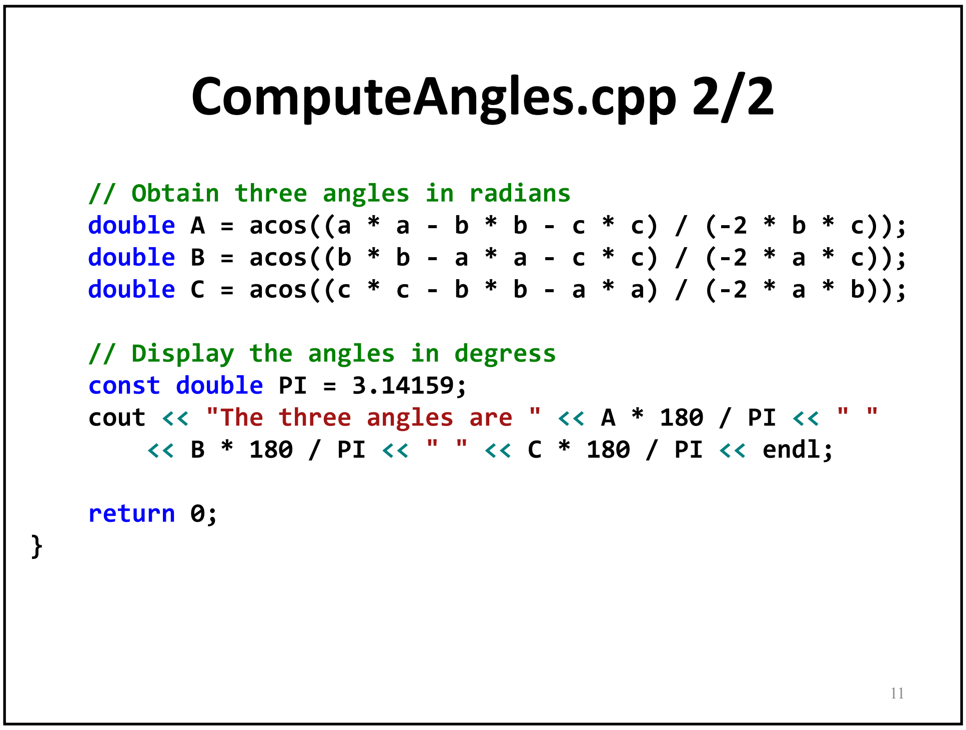
```
A = acos((a * a - b * b - c * c) / (-2 * b * c))
B = acos((b * b - a * a - c * c) / (-2 * a * c))
C = acos((c * c - b * b - a * a) / (-2 * a * b))
```

A program that prompts the user to enter the x- and y-coordinates of the three corner points in a triangle and then displays the triangle’s angles.

ComputeAngles Run

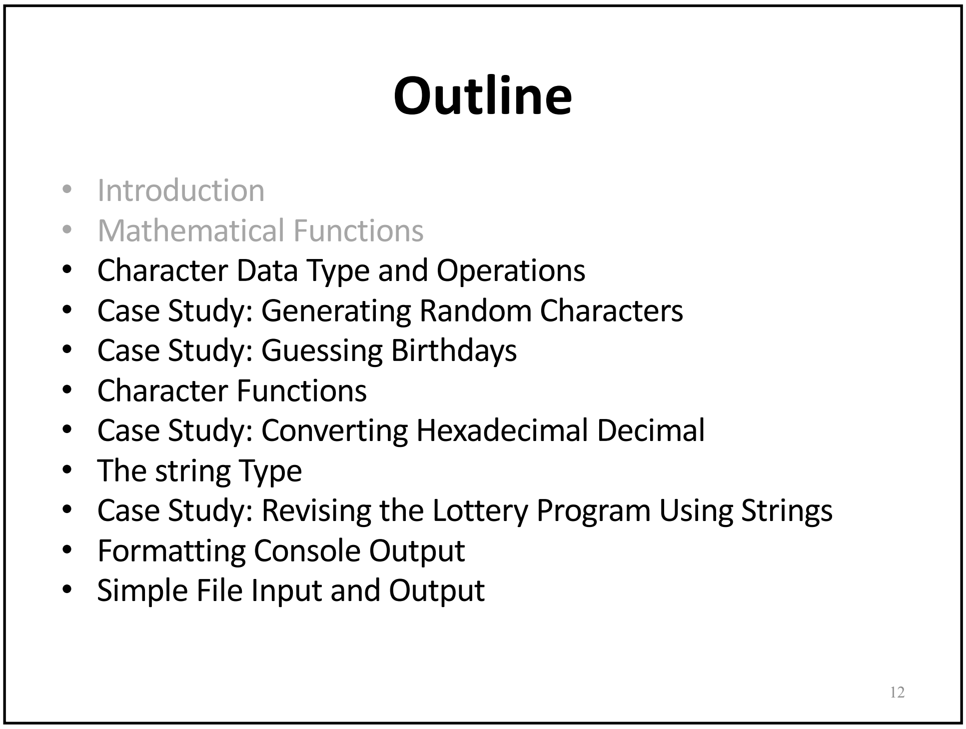
ComputeAngles.cpp 1/2

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

int main()
{
    // Prompt the user to enter three points
    cout << "Enter three points: ";
    double x1, y1, x2, y2, x3, y3;
    cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;

    // Compute three sides
    double a = sqrt((x2 - x3) * (x2 - x3) + (y2 - y3) * (y2 - y3));
    double b = sqrt((x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3));
    double c = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
```



ComputeAngles.cpp 2/2

```
    // Obtain three angles in radians
    double A = acos((a * a - b * b - c * c) / (-2 * b * c));
    double B = acos((b * b - a * a - c * c) / (-2 * a * c));
    double C = acos((c * c - b * b - a * a) / (-2 * a * b));

    // Display the angles in degress
    const double PI = 3.14159;
    cout << "The three angles are " << A * 180 / PI << " "
        << B * 180 / PI << " " << C * 180 / PI << endl;

    return 0;
}
```



Outline

- Introduction
- Mathematical Functions
- Character Data Type and Operations
- Case Study: Generating Random Characters
- Case Study: Guessing Birthdays
- Character Functions
- Case Study: Converting Hexadecimal Decimal
- The string Type
- Case Study: Revising the Lottery Program Using Strings
- Formatting Console Output
- Simple File Input and Output

Character Data Type

- *A character data type represents a single character.*

```
char letter = 'A'; (ASCII)
char numChar = '4'; (ASCII)
```

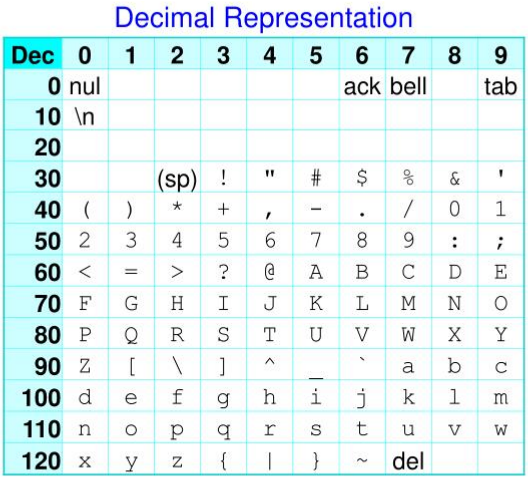
- The increment and decrement operators can also be used on **`char`** variables to get the next or preceding character. For example, the following statements display character **`b`**.

```
char ch = 'a';
cout << ++ch;
```

- The characters are encoded into numbers using the ASCII code.

Appendix B: ASCII Character Set

ASCII Character Set is a subset of the Unicode from \u0000 to \u007f



Decimal Representation

Dec	0	1	2	3	4	5	6	7	8	9
0	nul						ack	bell		tab
10	\n									
20										
30			(sp)	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	del		

ASCII Character Set in the Hexadecimal Index

Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Read Characters

To read a character from the keyboard, use

```
cout << "Enter a character: ";
char ch;
cin >> ch; // Read a character
```

Escape Sequences

C++ uses a special notation to represent special character.

Escape Sequence	*Name*	*ASCII Code*
`\b`	Backspace	8
`\t`	Tab	9
`\n`	Linefeed	10
`\f`	Formfeed	12
`\r`	Carriage Return	13
`\\`	Backslash	92
`\"`	Double Quote	34

```
cout << "He said \"Hi\".\n";
```

The output is: `He said "Hi".`

Casting between `char` and Numeric Types

- A **`char`** can be cast into any numeric type, and vice versa.
- When an integer is cast into a char, only its lower 8 bits of data are used; the other part is ignored.

```
int i = 'a';
// Same as int i = static_cast<int>('a');

char c = 97;
// Same as char c = static_cast<char>(97);
```

Numeric Operators on Characters

The **char** type is treated as if it is an integer of the byte size. All numeric operators can be applied to **char** operands.

```
// The ASCII code for '2' is 50 and for '3' is 51
int i = '2' + '3';
cout << "i is " << i << endl; // i is now 101

int j = 2 + 'a'; // The ASCII code for 'a' is 97
cout << "j is " << j << endl;
cout << j << " is the ASCII code for character " <<
  static_cast<char>(j) << endl;
```

Display

```
i is 101
j is 99
99 is the ASCII code for character c
```

Example: Converting a Lowercase to Uppercase

A program that prompts the user to enter a lowercase letter and finds its corresponding uppercase letter.

```
char uppercaseLetter =
  static_cast<char>('A' + (lowercaseLetter - 'a'));
```

ToUppercase Run

Comparing and Testing Characters

- The ASCII for lowercase letters are consecutive integers starting from the code for 'a', then for 'b', 'c', ..., and 'z'. The same is true for the uppercase letters.
- The lower case of a letter is larger than its upper case by 32.
- Two characters can be compared using the comparison operators just like comparing two numbers.
- `'a' < 'b'` is true because the ASCII code for `'a'` (97) is less than the ASCII code for 'b' (98).
- `'a' < 'A'` is false.
- `'1' < '8'` is true.

Outline

- Introduction
- Mathematical Functions
- Character Data Type and Operations
- Case Study: Generating Random Characters
- Case Study: Guessing Birthdays
- Character Functions
- Case Study: Converting Hexadecimal Decimal
- The string Type
- Case Study: Revising the Lottery Program Using Strings
- Formatting Console Output
- Simple File Input and Output

Case Study: Generating Random Characters

The **`rand()`** function returns a random integer. You can use it to write a simple expression to generate random numbers in any range.

`rand() % 10`	→	Returns a random integer between 0 and 9.
`50 + rand() % 50`	→	Returns a random integer between 50 and 99.

In general,

`a + rand() % b`	→	Returns a random number between a and a + b, excluding a + b.

Case Study: Generating Random Characters, cont.

Every character has a unique ASCII code between 0 and 127. To generate a random character is to generate a random integer between 0 and 127. The **`srand(seed)`** function is used to set a seed.

```
// Get a random character
srand(time(0));
char randomChar = static_cast<char>(startChar + rand() %
 (endChar - startChar + 1));

cout << "The random character between " << startChar << " and "
     << endChar << " is " << randomChar << endl;
```

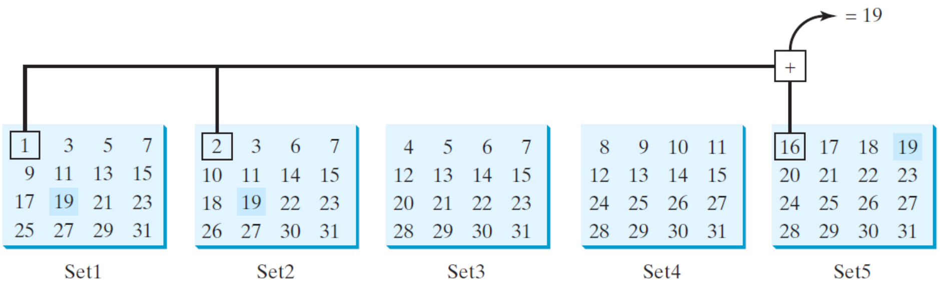
DisplayRandomCharacter Run

Outline

- Introduction
- Mathematical Functions
- Character Data Type and Operations
- Case Study: Generating Random Characters
- Case Study: Guessing Birthdays
- Character Functions
- Case Study: Converting Hexadecimal Decimal
- The string Type
- Case Study: Revising the Lottery Program Using Strings
- Formatting Console Output
- Simple File Input and Output

Case Study: Guessing Birthdays

- The program can find your birth date. The program prompts you to answer whether your birth date is in the following five sets of numbers:



1	3	5	7
9	11	13	15
17	19	21	23
25	27	29	31

Set1

2	3	6	7
10	11	14	15
18	19	22	23
26	27	30	31

Set2

4	5	6	7
12	13	14	15
20	21	22	23
28	29	30	31

Set3

8	9	10	11
12	13	14	15
24	25	26	27
28	29	30	31

Set4

16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31

Set5

GuessBirthday Run

Case Study: Guessing Birthdays

```
// Prompt the user for Set1
cout << "Is your birthday in Set1?" << endl;
cout << " 1  3  5  7\n" <<
        " 9 11 13 15\n" <<
        "17 19 21 23\n" <<
        "25 27 29 31" << endl;
cout << "Enter N/n for No and Y/y for Yes: ";
cin >> answer;

if (answer == 'Y' || answer == 'y')
  day += 1;
```

Outline

- Introduction
- Mathematical Functions
- Character Data Type and Operations
- Case Study: Generating Random Characters
- Case Study: Guessing Birthdays
- Character Functions
- Case Study: Converting Hexadecimal Decimal
- The string Type
- Case Study: Revising the Lottery Program Using Strings
- Formatting Console Output
- Simple File Input and Output

Character Functions

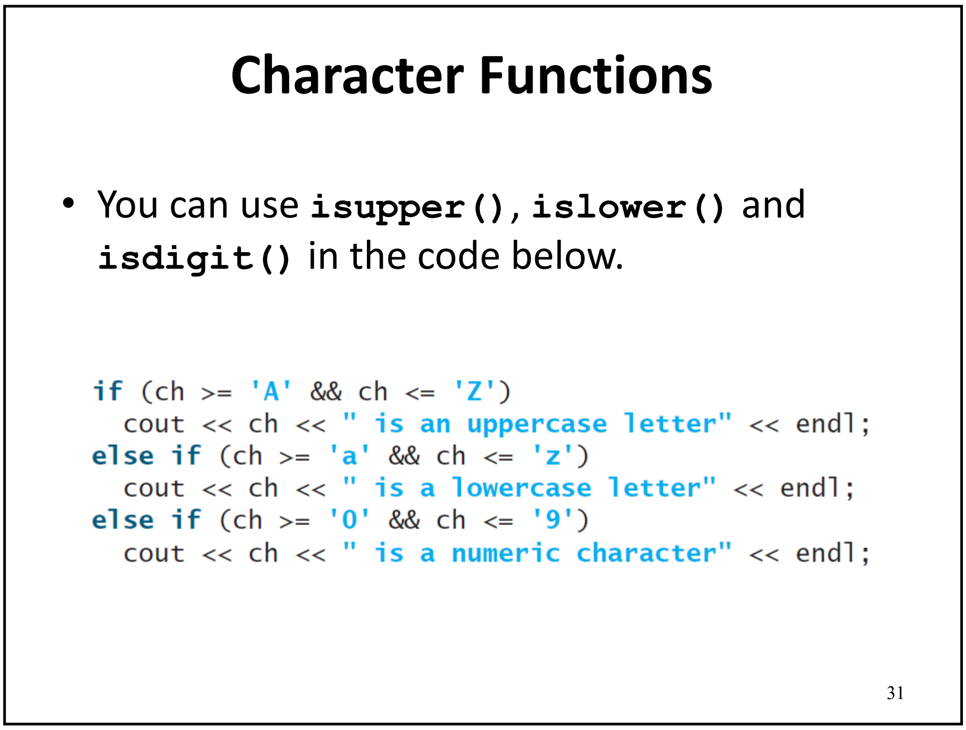
C++ contains functions for working with characters.

Function	*Description*
`isdigit(ch)`	Returns true if the specified character is a digit.
`isalpha(ch)`	Returns true if the specified character is a letter.
`isalnum(ch)`	Returns true if the specified character is a letter or digit.
`islower(ch)`	Returns true if the specified character is a lowercase letter.
`isupper(ch)`	Returns true if the specified character is an uppercase letter.
`isspace(ch)`	Returns true if the specified character is a whitespace character.
`tolower(ch)`	Returns the lowercase of the specified character.
`toupper(ch)`	Returns the uppercase of the specified character.

Example using Character Functions

```
if (islower(ch))
{
  cout << "It is a lowercase letter " << endl;
  cout << "Its equivalent uppercase letter is " <<
    static_cast<char>(toupper(ch)) << endl;
}
```

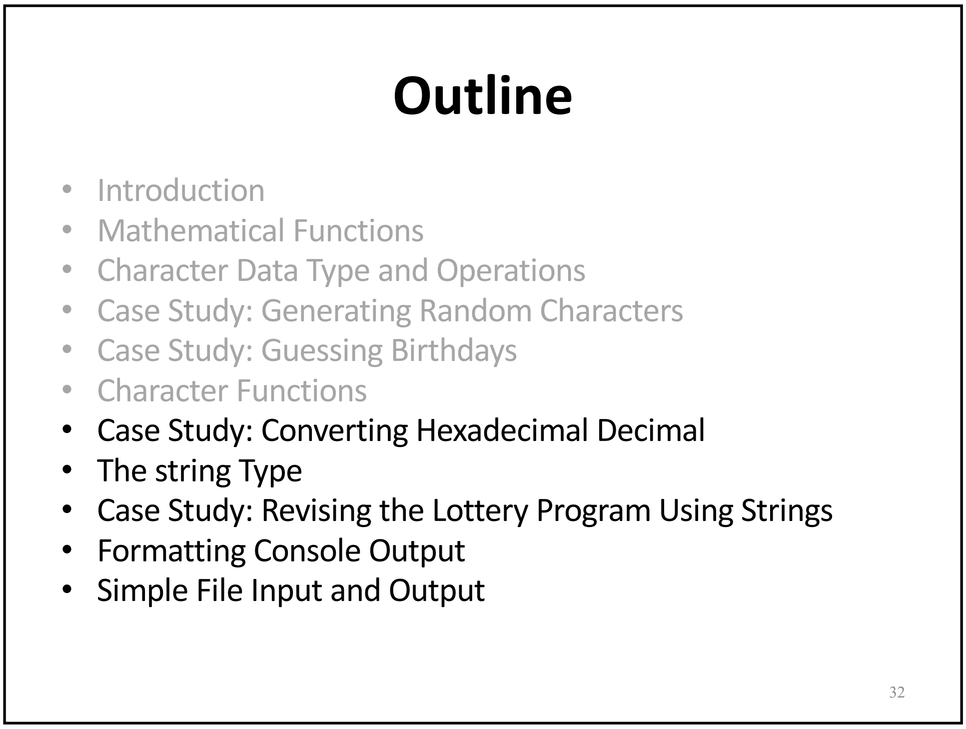
CharacterFunctions Run



Character Functions

- You can use **`isupper()`**, **`islower()`** and **`isdigit()`** in the code below.

```
if (ch >= 'A' && ch <= 'Z')
  cout << ch << " is an uppercase letter" << endl;
else if (ch >= 'a' && ch <= 'z')
  cout << ch << " is a lowercase letter" << endl;
else if (ch >= '0' && ch <= '9')
  cout << ch << " is a numeric character" << endl;
```



Outline

- Introduction
- Mathematical Functions
- Character Data Type and Operations
- Case Study: Generating Random Characters
- Case Study: Guessing Birthdays
- Character Functions
- Case Study: Converting Hexadecimal Decimal
- The string Type
- Case Study: Revising the Lottery Program Using Strings
- Formatting Console Output
- Simple File Input and Output

Case Study: Converting a Hexadecimal Digit to a Decimal Value

A program that converts a hexadecimal digit to decimal.

DECIMAL	HEX	BINARY
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

HexDigit2Dec

Run

HexDigit2Dec.cpp

```
hexDigit = toupper(hexDigit);
if (hexDigit <= 'F' && hexDigit >= 'A')
{
  int value = 10 + hexDigit - 'A';
  cout << "The decimal value for hex digit "
    << hexDigit << " is " << value << endl;
}
else if (isdigit(hexDigit))
{
  cout << "The decimal value for hex digit "
    << hexDigit << " is " << hexDigit << endl;
}
```

Outline

- Introduction
- Mathematical Functions
- Character Data Type and Operations
- Case Study: Generating Random Characters
- Case Study: Guessing Birthdays
- Character Functions
- Case Study: Converting Hexadecimal Decimal
- The string Type
- Case Study: Revising the Lottery Program Using Strings
- Formatting Console Output
- Simple File Input and Output

The `string` Type

A string is a sequence of characters.

```
#include <string>
string s;
string message = "Programming is fun";
```

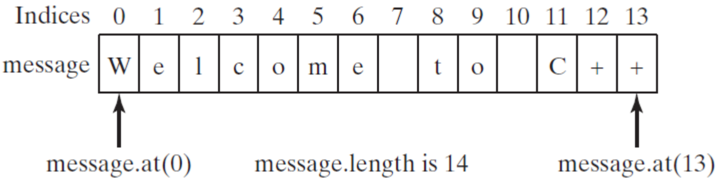
Function	*Description*
`length()`	Returns the number of characters in this string.
`size()`	Same as length().
`at(index)`	Returns the character at the specified index from this string.

String Subscript Operator

C++ provides the subscript operator for accessing the character at a specified index in a string using the syntax **`stringName[index]`**.

```
string s = "welcome to C++";
s.at(0) = 'W';
cout << s.length() << s[0] << endl;
```

```
14W
```



Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13
message	W	e	l	c	o	m	e		t	o		C	+	+

message.at(0) message.length is 14 message.at(13)

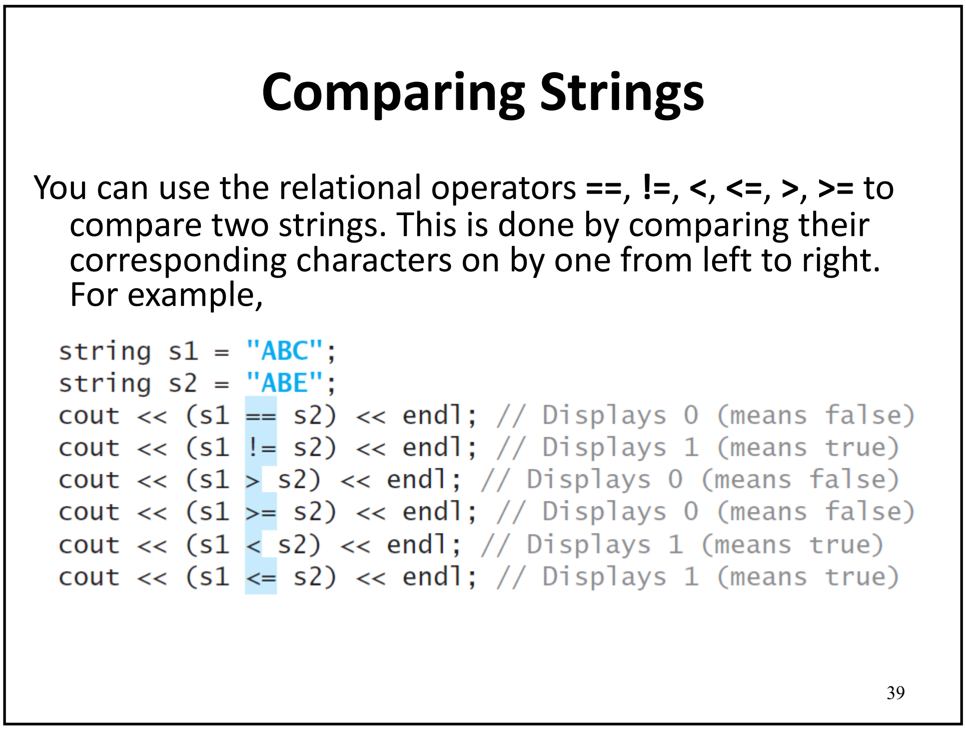
Concatenating Strings

C++ provides the + operator for concatenating two strings.

```
string s3 = s1 + s2;
```

```
string m = "Good";
m += " morning";
m += '!';
cout << m << endl;
```

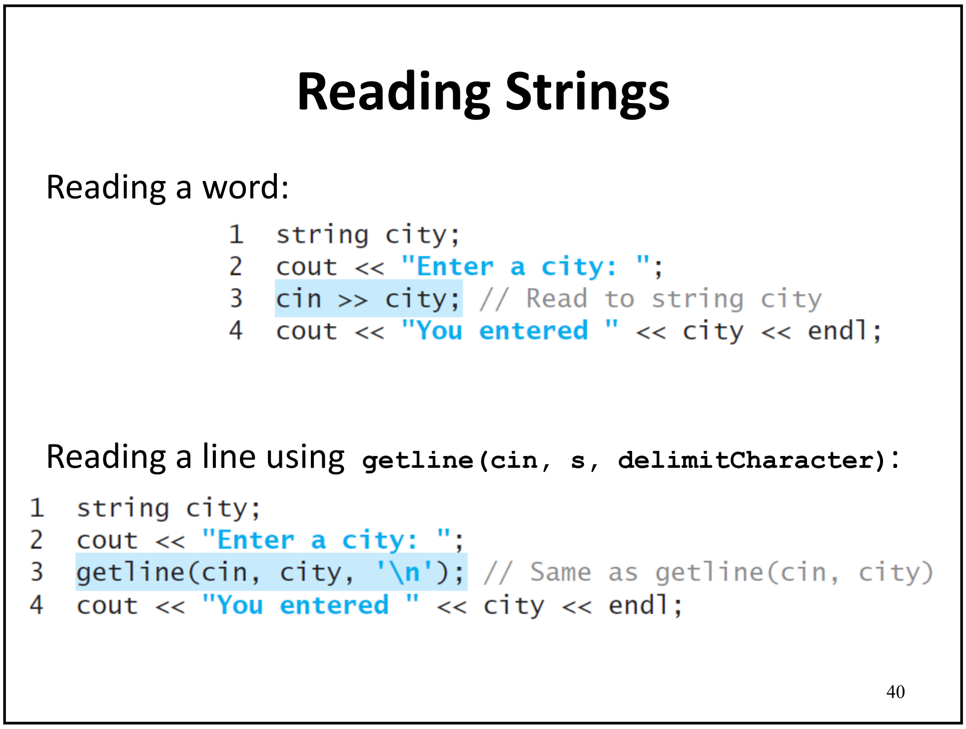
```
Good morning!
```



Comparing Strings

You can use the relational operators **==**, **!=**, **<**, **<=**, **>**, **>=** to compare two strings. This is done by comparing their corresponding characters on by one from left to right. For example,

```
string s1 = "ABC";
string s2 = "ABE";
cout << (s1 == s2) << endl; // Displays 0 (means false)
cout << (s1 != s2) << endl; // Displays 1 (means true)
cout << (s1 > s2) << endl; // Displays 0 (means false)
cout << (s1 >= s2) << endl; // Displays 0 (means false)
cout << (s1 < s2) << endl; // Displays 1 (means true)
cout << (s1 <= s2) << endl; // Displays 1 (means true)
```



Reading Strings

Reading a word:

```
string city;
cout << "Enter a city: ";
cin >> city; // Read to string city
cout << "You entered " << city << endl;
```

Reading a line using `getline(cin, s, delimitCharacter)`:

```
string city;
cout << "Enter a city: ";
getline(cin, city, '\n'); // Same as getline(cin, city)
cout << "You entered " << city << endl;
```

Example: Order Two Cities

A program that prompts the user to enter two cities and displays them in alphabetical order.

OrderTwoCities Run

OrderTwoCities.cpp

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

int main() {
    string city1, city2;
    cout << "Enter the first city: ";
    getline(cin, city1);
    cout << "Enter the second city: ";
    getline(cin, city2);

    cout << "The cities in alphabetical order are ";
    if (city1 < city2)
        cout << city1 << " " << city2 << endl;
    else
        cout << city2 << " " << city1 << endl;

    return 0;
}
```

Outline

- Introduction
- Mathematical Functions
- Character Data Type and Operations
- Case Study: Generating Random Characters
- Case Study: Guessing Birthdays
- Character Functions
- Case Study: Converting Hexadecimal Decimal
- The string Type
- Case Study: Revising the Lottery Program Using Strings
- Formatting Console Output
- Simple File Input and Output

Case Study: Revising the Lottery Program Using Strings

A problem can be solved using many different approaches. This section rewrites the lottery program in Listing 3.7 using strings. Using strings simplifies this program.

```
// Check the guess
if (guess == lottery)
  cout << "Exact match: you win $10,000" << endl;
else if (guess[1] == lottery[0] && guess[0] == lottery[1])
  cout << "Match all digits: you win $3,000" << endl;
else if (guess[0] == lottery[0] || guess[0] == lottery[1]
      || guess[1] == lottery[0] || guess[1] == lottery[1])
  cout << "Match one digit: you win $1,000" << endl;
else
  cout << "Sorry, no match" << endl;
```

LotteryUsingStrings Run

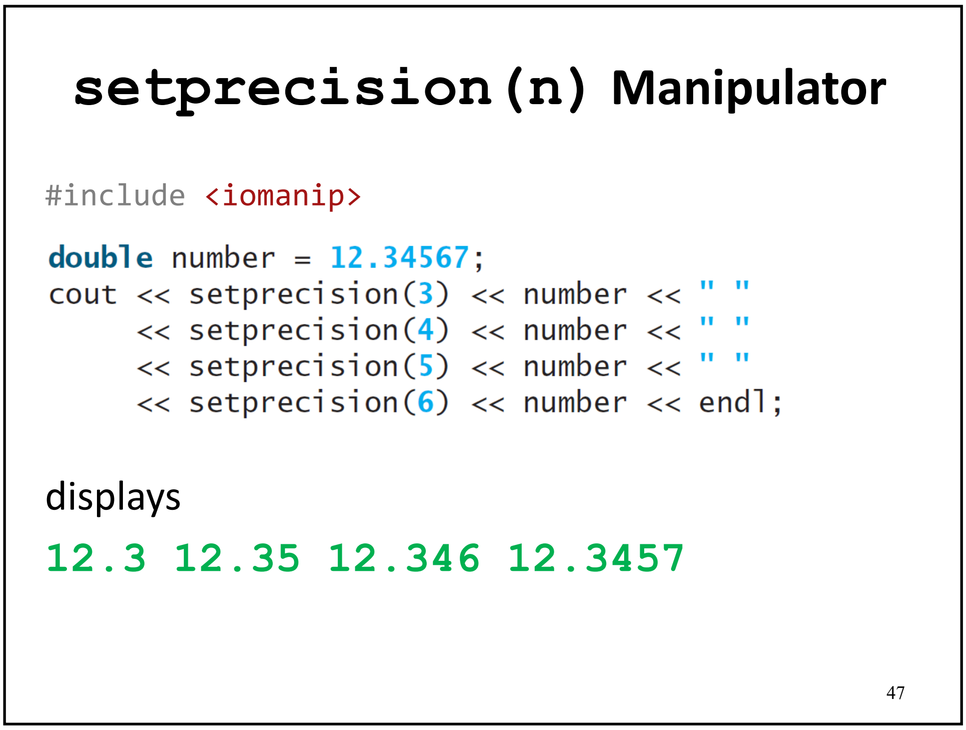
Outline

- Introduction
- Mathematical Functions
- Character Data Type and Operations
- Case Study: Generating Random Characters
- Case Study: Guessing Birthdays
- Character Functions
- Case Study: Converting Hexadecimal Decimal
- The string Type
- Case Study: Revising the Lottery Program Using Strings
- Formatting Console Output
- Simple File Input and Output

Formatting Console Output

You can use the stream manipulators to display formatted output on the console.

Operator	*Description*
`setprecision(n)`	sets the precision of a floating-point number
`fixed`	displays floating-point numbers in fixed-point notation
`showpoint`	causes a floating-point number to be displayed with a decimal point with trailing zeros even if it has no fractional part
`setw(width)`	specifies the width of a print field
`left`	justifies the output to the left
`right`	justifies the output to the right

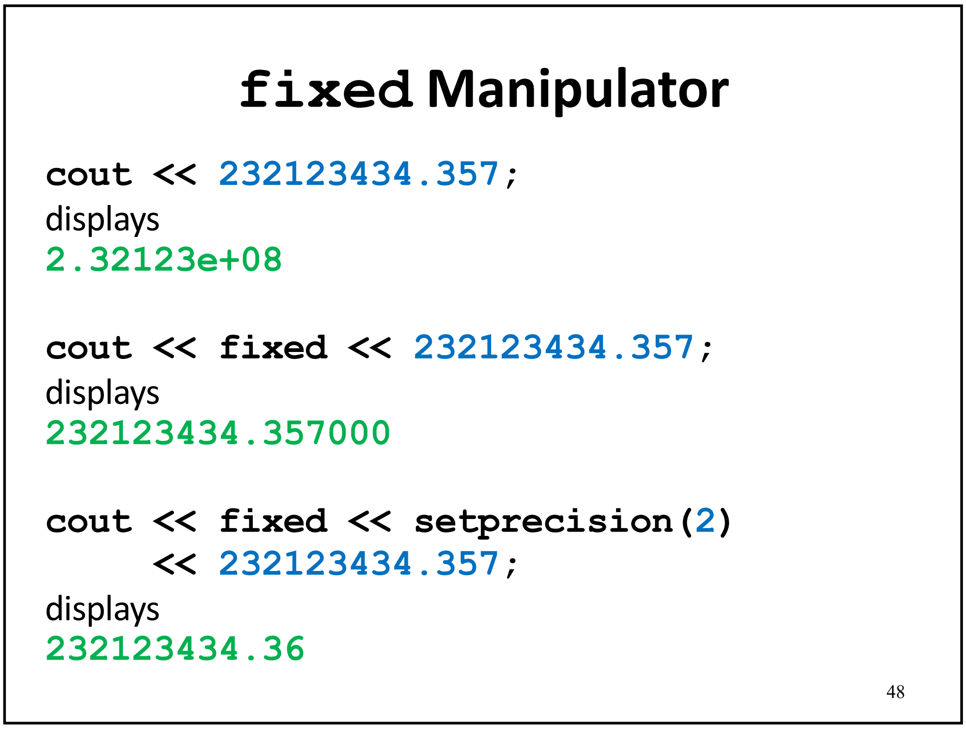


`setprecision(n)` Manipulator

```
#include <iomanip>

double number = 12.34567;
cout << setprecision(3) << number << " "
     << setprecision(4) << number << " "
     << setprecision(5) << number << " "
     << setprecision(6) << number << endl;
```

displays

```
12.3 12.35 12.346 12.3457
```



`fixed` Manipulator

```
cout << 232123434.357;
```

displays

```
2.32123e+08
```

```
cout << fixed << 232123434.357;
```

displays

```
232123434.357000
```

```
cout << fixed << setprecision(2)
     << 232123434.357;
```

displays

```
232123434.36
```

`showpoint` Manipulator

```
cout << setprecision(6);
cout << 1.23 << endl;
cout << showpoint << 1.23 << endl;
cout << showpoint << 123.0 << endl;
```

displays

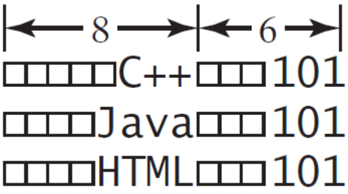
```
1.23
1.23000
123.000
```

`setw(width)` Manipulator

```
cout << setw(8) << "C++" << setw(6) << 101 << endl;
cout << setw(8) << "Java" << setw(6) << 101 << endl;
cout << setw(8) << "HTML" << setw(6) << 101 << endl;
```

displays

```
     C++   101
    Java   101
    HTML   101
```



```
cout << setw(8) << "Programming" << "#" << setw(2) << 101;
Prgramming#101
```

`left` and `right` Manipulators

```
cout << right;
cout << setw(8) << 1.23 << endl;
cout << setw(8) << 351.34 << endl;
```

displays

```
□□□□1.23
□□351.34
```

`left` and `right` Manipulators

```
cout << left;
cout << setw(8) << 1.23;
cout << setw(8) << 351.34 << endl;
```

displays

```
1.23□□□□351.34□□
```

Outline

- Introduction
- Mathematical Functions
- Character Data Type and Operations
- Case Study: Generating Random Characters
- Case Study: Guessing Birthdays
- Character Functions
- Case Study: Converting Hexadecimal Decimal
- The string Type
- Case Study: Revising the Lottery Program Using Strings
- Formatting Console Output
- Simple File Input and Output

Simple File Output

To write data to a file, first declare a variable of the **`ofstream`** type:

```
#include <fstream>
ofstream output;
```

To specify a file, invoke the **`open`** function from **`output`** object as follows:

```
output.open("numbers.txt");
```

Optionally, you can create a file output object and open the file in one statement like this:

```
ofstream output("numbers.txt");
```

To write data, use the stream insertion operator (**<<**) in the same way that you send data to the **cout** object. For example,

```
output << 95 << " " << 56 << " " << 34 << endl;
```

Finally:

```
output.close();
```

SimpleFileOutput Run

Simple File Input

To read data from a file, first declare a variable of the **`ifstream`** type:

```
#include <fstream>
ifstream input;
```

To specify a file, invoke the **`open`** function from **`input`** as follows:

```
input.open("numbers.txt");
```

Or:

```
ifstream input("numbers.txt");
```

To read data, use the stream extraction operator (**>>**) in the same way that you read data from the **`cin`** object. For example,

```
input >> score1 >> score2 >> score3;
```

Finally:

```
input.close();
```

SimpleFileInput Run

Outline

- Introduction
- Mathematical Functions
- Character Data Type and Operations
- Case Study: Generating Random Characters
- Case Study: Guessing Birthdays
- Character Functions
- Case Study: Converting Hexadecimal Decimal
- The string Type
- Case Study: Revising the Lottery Program Using Strings
- Formatting Console Output
- Simple File Input and Output

Chapter 5: Loops

Sections 5.1–5.6, 5.9

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)

Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

Introduction

Suppose that you need to print a string (e.g., "Welcome to C++!") a hundred times. It would be tedious to have to write the following statement a hundred times:

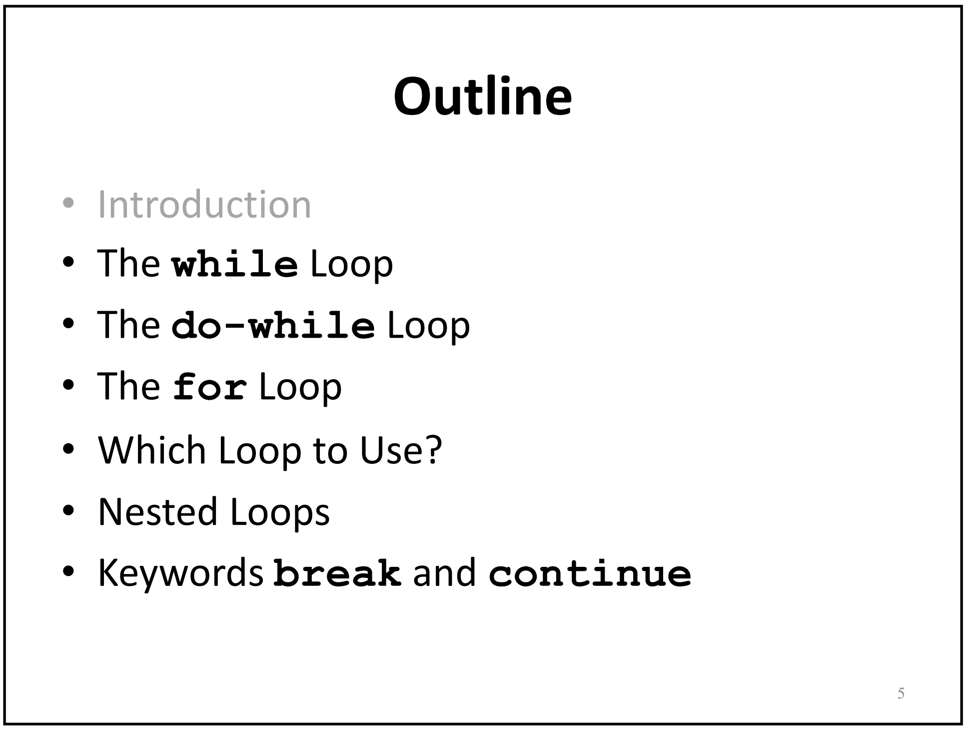
```
cout  << "Welcome to C++!" << endl;
```

Introduction

100 times

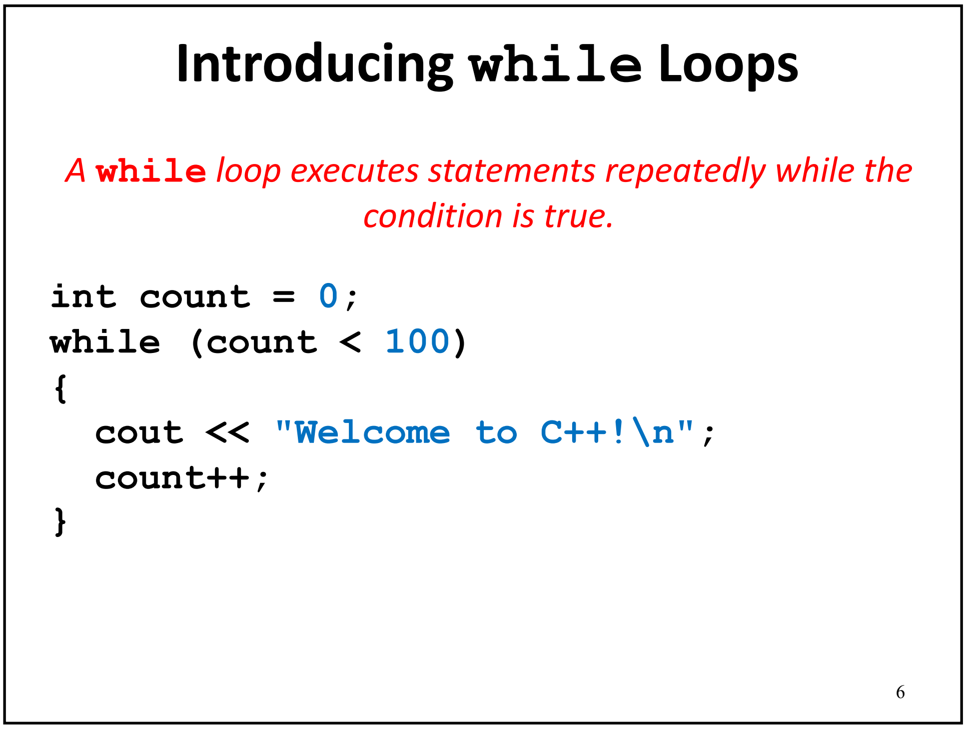
```
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;


…

cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
cout << "Welcome to Java!" << endl;
```

So, how do you solve this problem?



Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**



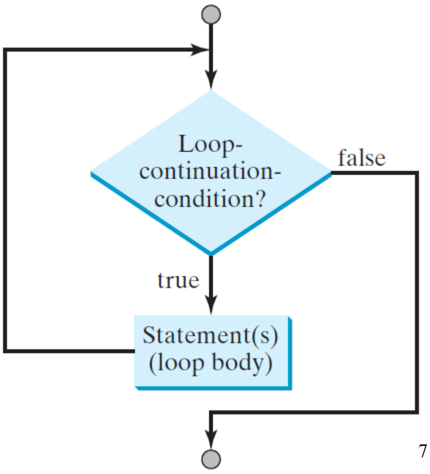
Introducing while Loops

*A **`while`** loop executes statements repeatedly while the condition is true.*

```
int count = 0;
while (count < 100)
{
   cout << "Welcome to C++!\n";
   count++;
}
```

`while` Loop Flow Chart

```
while (loop-continuation-condition)
{
  // Loop body
  Statement(s);
}
```



---

animation

Trace `while` Loop

```
int count = 0;
while (count < 2)
{
  cout << "Welcome to C++!";
  count++;
}
```

Initialize count

animation

Trace while Loop, cont.

(count < 2) is true

```
int count = 0;
while (count < 2)
{
   cout << "Welcome to C++!";
   count++;
}
```

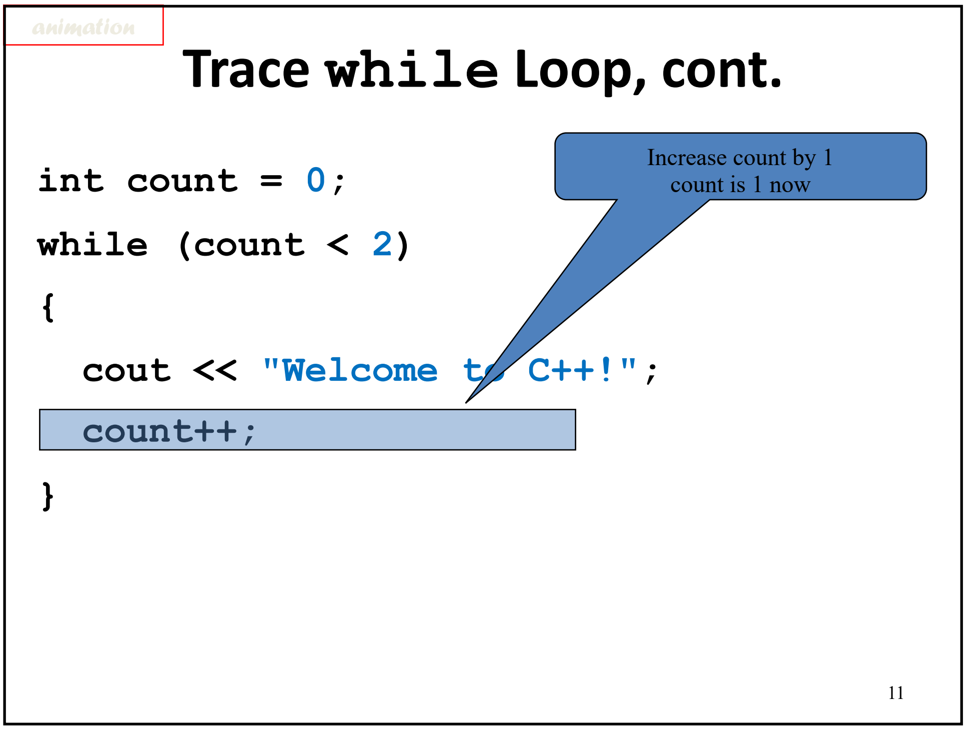
---

animation

Trace while Loop, cont.

Print Welcome to C++

```
int count = 0;
while (count < 2)
{
   cout << "Welcome to C++!";
   count++;
}
```

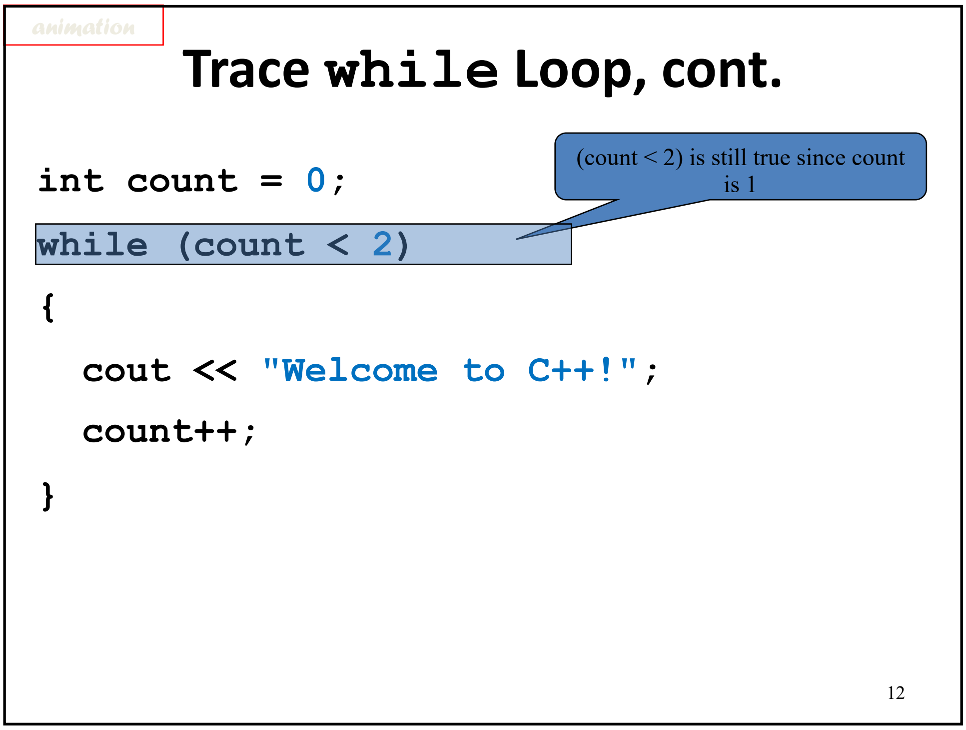


animation

Trace while Loop, cont.

```
int count = 0;
while (count < 2)
{
  cout << "Welcome to C++!";
  count++;
}
```

Increase count by 1
count is 1 now



animation

Trace while Loop, cont.

```
int count = 0;
while (count < 2)
{
  cout << "Welcome to C++!";
  count++;
}
```

(count < 2) is still true since count is 1

animation

Trace while Loop, cont.

Print Welcome to C++

```
int count = 0;
while (count < 2)
{
  cout << "Welcome to C++!";
  count++;
}
```

animation

Trace while Loop, cont.

Increase count by 1
count is 2 now

```
int count = 0;
while (count < 2)
{
  cout << "Welcome to C++!";
  count++;
}
```

animation

Trace while Loop, cont.

(count < 2) is false since count is 2 now

```
int count = 0;
while (count < 2)
{
   cout << "Welcome to C++!";
   count++;
}
```

animation

Trace while Loop

The loop exits. Execute the next statement after the loop.

```
int count = 0;
while (count < 2)
{
   cout << "Welcome to C++!";
   count++;
}
```

Case Study: Guessing Numbers

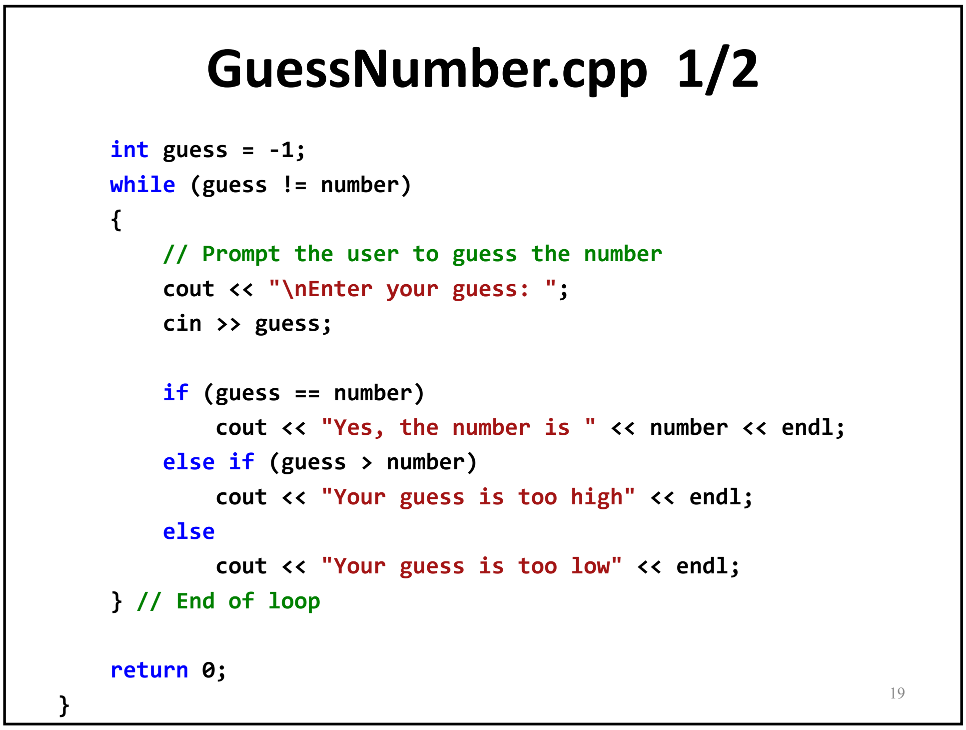
Write a program that randomly generates an integer between **0** and **100**, inclusive. The program prompts the user to enter a number continuously until the number matches the randomly generated number. For each user input, the program tells the user whether the input is too low or too high, so the user can choose the next input intelligently. Here is a sample run:

GuessNumberOneTime Run

GuessNumber Run

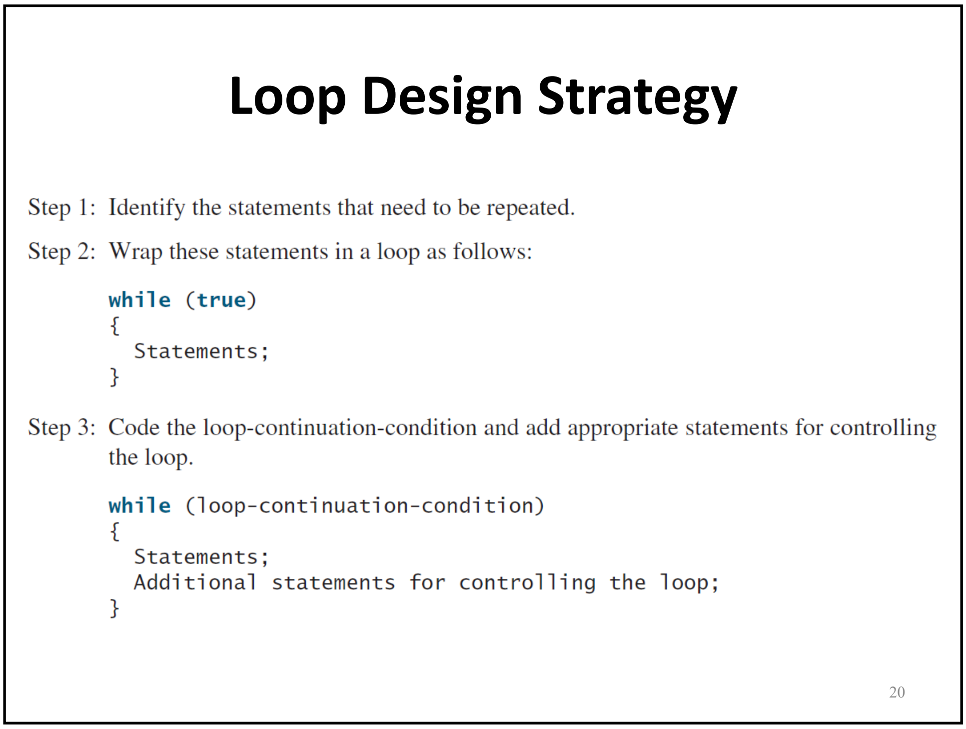
GuessNumber.cpp 1/2

```
#include <iostream>
#include <cstdlib>
#include <ctime> // Needed for the time function
using namespace std;

int main()
{
    // Generate a random number to be guessed
    srand(time(0));
    int number = rand() % 101;

    cout << "Guess a magic number between 0 and 100";
```



GuessNumber.cpp 1/2

```
    int guess = -1;
    while (guess != number)
    {
        // Prompt the user to guess the number
        cout << "\nEnter your guess: ";
        cin >> guess;

        if (guess == number)
            cout << "Yes, the number is " << number << endl;
        else if (guess > number)
            cout << "Your guess is too high" << endl;
        else
            cout << "Your guess is too low" << endl;
    } // End of loop

    return 0;
}
```



Loop Design Strategy

Step 1: Identify the statements that need to be repeated.

Step 2: Wrap these statements in a loop as follows:

```
while (true)
{
  Statements;
}
```

Step 3: Code the loop-continuation-condition and add appropriate statements for controlling the loop.

```
while (loop-continuation-condition)
{
  Statements;
  Additional statements for controlling the loop;
}
```

Case Study: Multiple Subtraction Quiz

Take the subtraction quiz 5 times.

Report number of correct answers and the quiz time.

SubtractionQuiz

Run

SubtractionQuizLoop.cpp 1/3

```
#include <iostream>
#include <ctime> // Needed for time function
#include <cstdlib> // Needed for the srand and rand functions
using namespace std;
int main()
{
    int correctCount = 0; // Count the number of correct answers
    int count = 0; // Count the number of questions
    long startTime = time(0);
    const int NUMBER_OF_QUESTIONS = 5;
    srand(time(0)); // Set a random seed
    while (count < NUMBER_OF_QUESTIONS)
    {      See next slides      }
    long endTime = time(0);
    long testTime = endTime - startTime;
    cout << "Correct count is " << correctCount << "\nTest time is "
        << testTime << " seconds\n";
    return 0;
}
```

SubtractionQuizLoop.cpp 2/3

```
while (count < NUMBER_OF_QUESTIONS)
    {
        // 1. Generate two random single-digit integers
        int number1 = rand() % 10;
        int number2 = rand() % 10;

        // 2. If number1 < number2, swap number1 with number2
        if (number1 < number2)
        {
            int temp = number1;
            number1 = number2;
            number2 = temp;
        }
```

SubtractionQuizLoop.cpp 3/3

```
        // 3. Prompt the student to answer “what is num1 – num2?”
        cout << "What is " << number1 << " - " << number2 << "? ";
        int answer;
        cin >> answer;
        // 4. Grade the answer and display the result
        if (number1 - number2 == answer)
        {
            cout << "You are correct!\n";
            correctCount++;
        }
        else
            cout << "Your answer is wrong.\n" << number1 << " - " <<
            number2 << " should be " << (number1 - number2) << endl;
        // Increase the count
        count++;
    }
```

Controlling a Loop with User Confirmation

```
char continueLoop = 'Y';
while (continueLoop == 'Y')
{
  // Execute the loop body once
  ...

  // Prompt the user for confirmation
  cout << "Enter Y to continue and N to quit: ";
  cin >> continueLoop;
}
```

Controlling a Loop with a Sentinel Value

You may use an input value to signify the end of the loop. Such a value is known as a *sentinel value*.

A program that reads and calculates the sum of an unspecified number of integers. The input 0 signifies the end of the input.

SentinelValue Run

SentinelValue.cpp

```
int data;
cin >> data;

// Keep reading data until the input is 0
int sum = 0;
while (data != 0)
{
    sum += data;

    // Read the next data
    cout << "Enter an integer (the input ends " <<
        "if it is 0): ";
    cin >> data;
}

cout << "The sum is " << sum << endl;
```

Input and Output Redirections

- If you have a large number of data to enter, it would be cumbersome to type from the keyboard.
- You may store the data separated by whitespaces in a text file, say `input.txt`, and run the program and redirecting input to the file.
- You can also redirect program output to a text file, say `outpu.txt`.

```
SentinelValue.exe < input.txt > output.txt
```

Reading Data from a File

- If you have many numbers to read from a file, you need to write a loop to read all these numbers.
- You can invoke the **`eof()`** function on the input object to detect the end of file.
- A program that reads all numbers from the file **`numbers.txt`**.

ReadAllData Run

ReadAllData.cpp

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    // Open a file
    ifstream input("numbers.txt");

    double sum = 0;
    double number;
    while (!input.eof()) // Read data to the end of file
    {
        input >> number; // Read data
        cout << number << " "; // Display data
        sum += number;
    }
    input.close();
    cout << "\nTotal is " << sum << endl;
    return 0;
}
```

Caution

- Don’t use floating-point values for equality checking in a loop control expression; they are approximations, using them can result in inaccurate results.
- The following loop does not stop.

```
double item = 1;
double sum = 0;
while (item != 0) // No guarantee it will be 0
{
    sum += item;
    item -= 0.1;
}
```

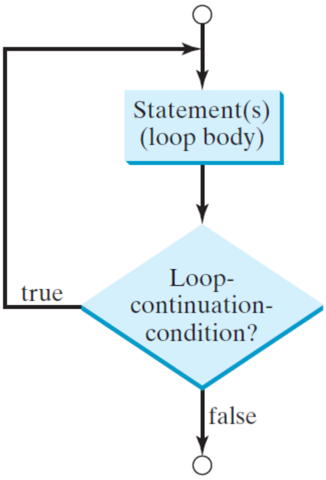
Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

`do-while` Loop

A ***`do-while`*** *loop is the same as a while loop except that it executes the loop body first and then checks the loop continuation condition.*

```
do
{
  // Loop body;
  Statement(s);
} while (loop-continuation-cond
```



TestDoWhile

Run

TestDoWhile.cpp

```
// Initialize data and sum
int data = 0;
int sum = 0;

do
{
    sum += data;

    // Read the next data
    cout << "Enter an integer (the input ends " <<
        "if it is 0): ";
    cin >> data; // Keep reading until the input is 0
} while (data != 0);

cout << "The sum is " << sum << endl;
```

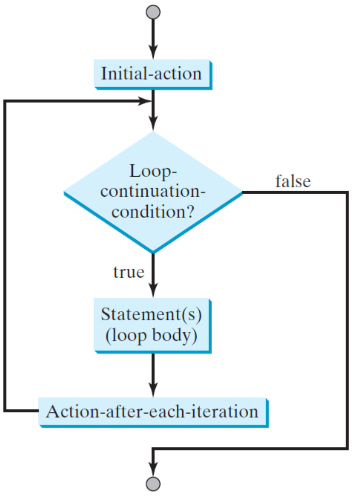
Outline

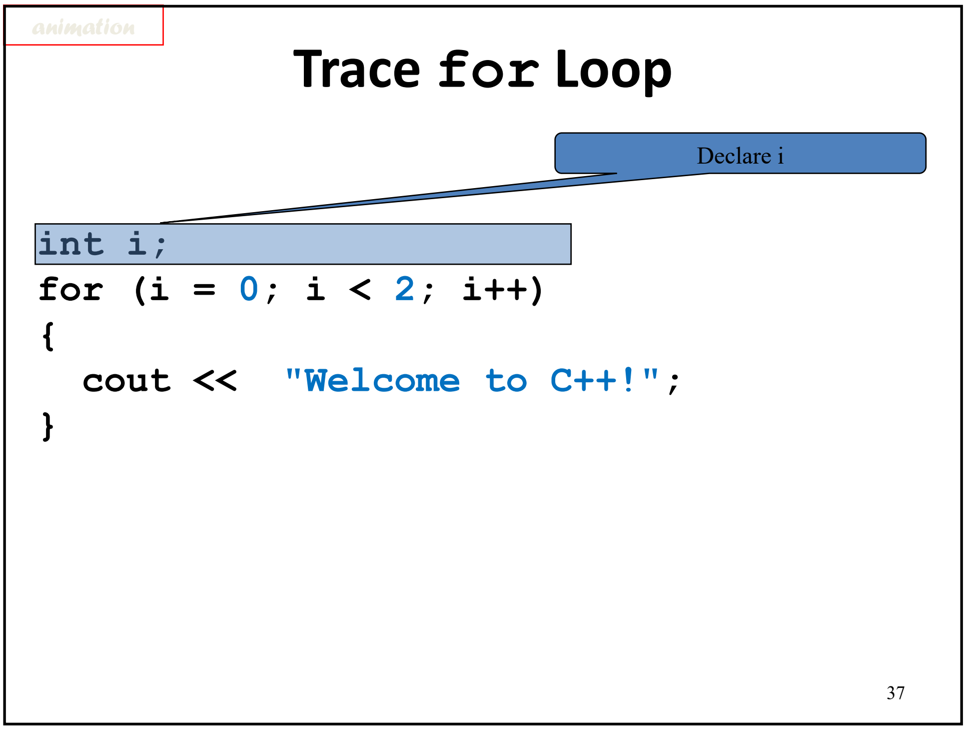
- Introduction
- The **while** Loop
- The **do-while** Loop
- The **for** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **break** and **continue**

for Loops

```
for (initial-action; loop-continuation-condition;
     action-after-each-iteration)
{
  // Loop body;
  Statement(s);
}
```

*A **for** loop has a concise syntax for writing loops.*



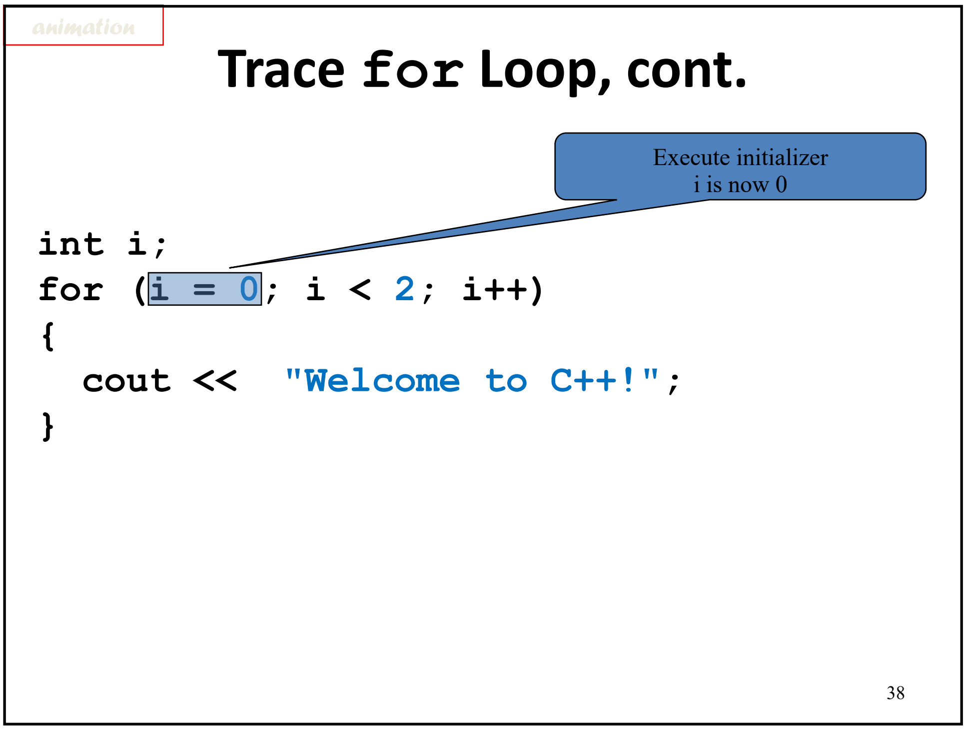


animation

Trace for Loop

Declare i

```
int i;
for (i = 0; i < 2; i++)
{
  cout <<  "Welcome to C++!";
}
```

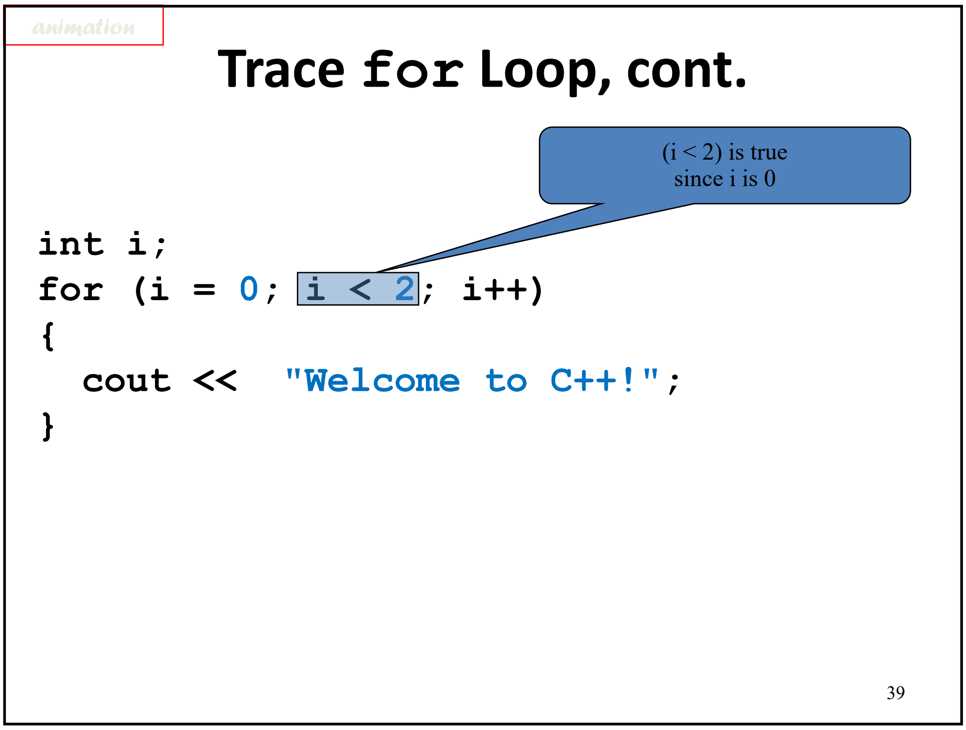


animation

Trace for Loop, cont.

Execute initializer
i is now 0

```
int i;
for (i = 0; i < 2; i++)
{
  cout <<  "Welcome to C++!";
}
```

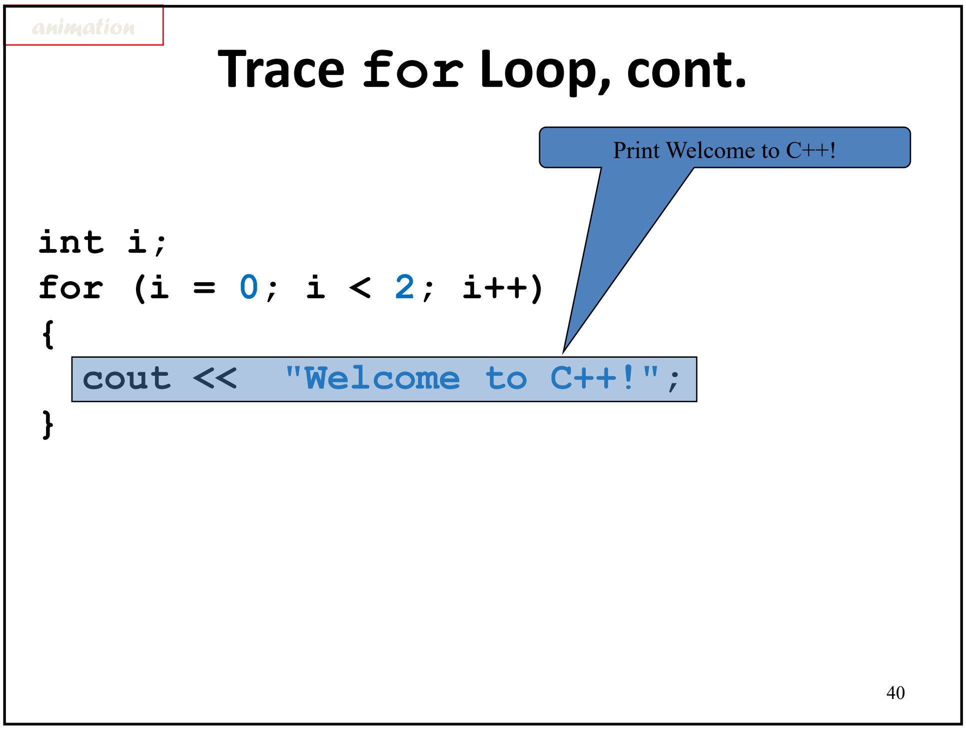


animation

Trace for Loop, cont.

(i < 2) is true
since i is 0

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```

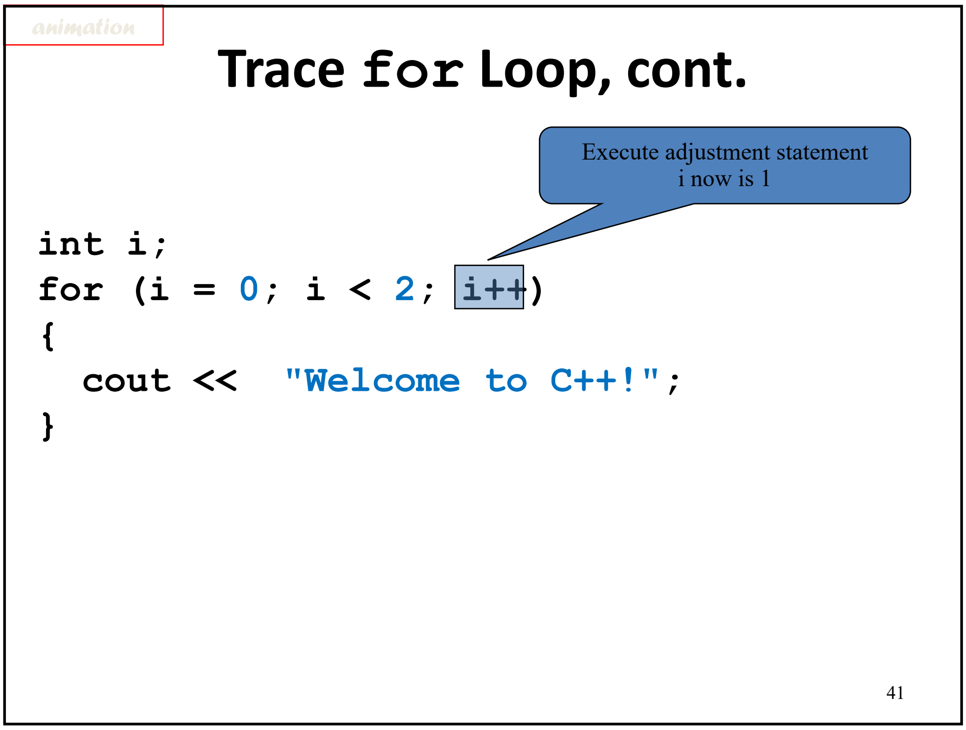


animation

Trace for Loop, cont.

Print Welcome to C++!

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```

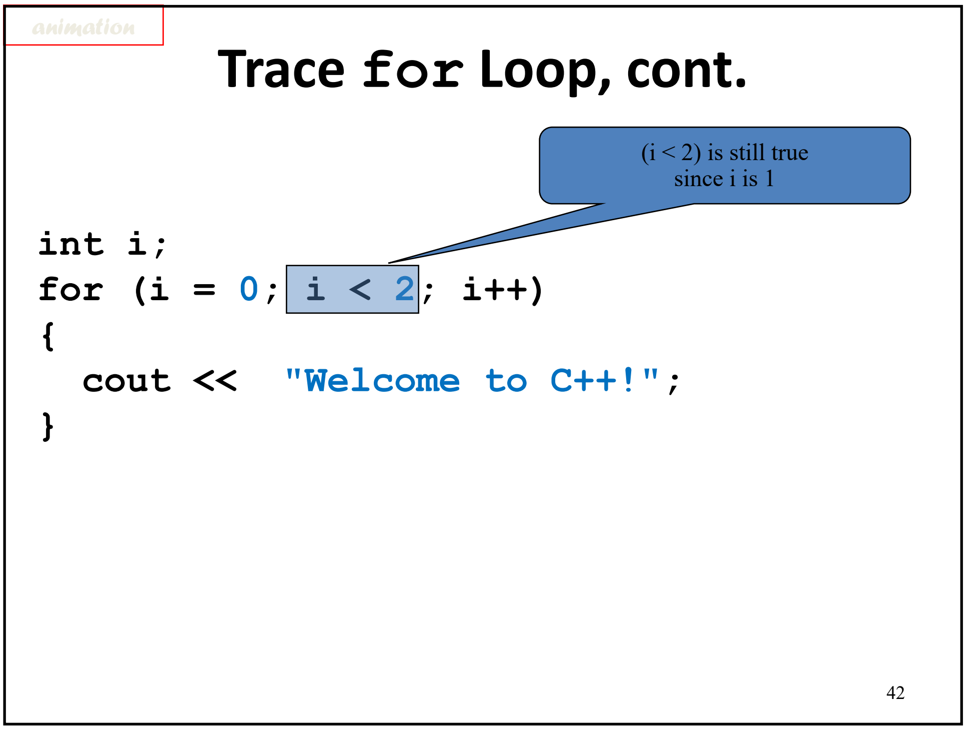


animation

Trace for Loop, cont.

Execute adjustment statement
i now is 1

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```



animation

Trace for Loop, cont.

(i < 2) is still true
since i is 1

```
int i;
for (i = 0; i < 2; i++)
{
   cout <<  "Welcome to C++!";
}
```

animation

Trace for Loop, cont.

Print Welcome to C++

```
int i;
for (i = 0; i < 2; i++)
{
  cout <<  "Welcome to C++!";
}
```

animation

Trace for Loop, cont.

Execute adjustment statement
i now is 2

```
int i;
for (i = 0; i < 2; i++)
{
  cout <<  "Welcome to C++!";
}
```

animation

Trace `for` Loop, cont.

(i < 2) is false
since i is 2

```
int i;
for (i = 0; i < 2; i++)
{
  cout <<  "Welcome to C++!";
}
```

animation

Trace `for` Loop, cont.

Exit the loop. Execute the next
statement after the loop

```
int i;
for (i = 0; i < 2; i++)
{
  cout <<  "Welcome to C++!";
}
```

Note

- The `initial-action` in a `for` loop can be a list of zero or more comma-separated expressions.

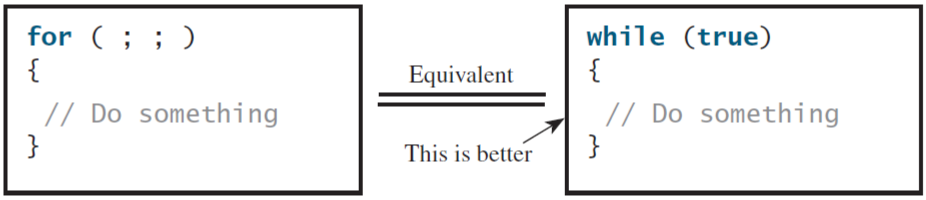
```
for (int i = 0, j = 0; i + j < 10; i++, j++)
{
  // Do something
}
```

- The `action-after-each-iteration` in a `for` loop can be a list of zero or more comma-separated statements.

```
for (int i = 1; i < 100; cout << i << endl, i++);
```

Note

- If the `loop-continuation-condition` in a `for` loop is omitted, it is implicitly true. Thus the `for` statement given below, which is an infinite loop, is correct.
- It is better to use the equivalent `while` loop to avoid confusion:



```
for ( ; ; )
{
  // Do something
}
```

Equivalent

This is better

```
while (true)
{
  // Do something
}
```

Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

Which Loop to Use?

- The loop statements, **`while`**, **`do-while`**, and **`for`**, are expressively equivalent; that is, you can write a loop in any of these three forms.
- The **`while`** loop can always be converted into the **`for`** loop.

```
while (loop-continuation-condition)
{
  // Loop body
}
```

(a)

Equivalent

```
for ( ; loop-continuation-condition; )
{
  // Loop body
}
```

(b)

- The **`for`** loop can generally be converted into the **`while`** loop.

```
for (initial-action;
     loop-continuation-condition;
     action-after-each-iteration)
{
  // Loop body;
}
```

(a)

Equivalent

```
initial-action;
while (loop-continuation-condition)
{
  // Loop body;
  action-after-each-iteration;
}
```

(b)

Which Loop to Use?

- Use the one that is most intuitive and comfortable for you.
- In general, a **`for`** loop may be used if the number of repetitions is counter-controlled, as, for example, when you need to print a message 100 times.
- A **`while`** loop may be used if the number of repetitions is sentinel-controlled, as in the case of reading the numbers until the input is 0.
- A **`do-while`** loop can be used to replace a while loop if the loop body has to be executed before testing the continuation condition.

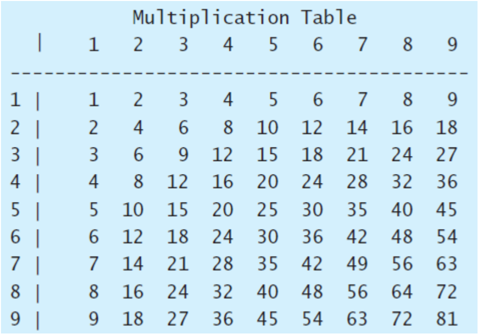
Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

Nested Loops

A loop can be nested inside another loop.

Example: A program that uses nested for loops to print a multiplication table.



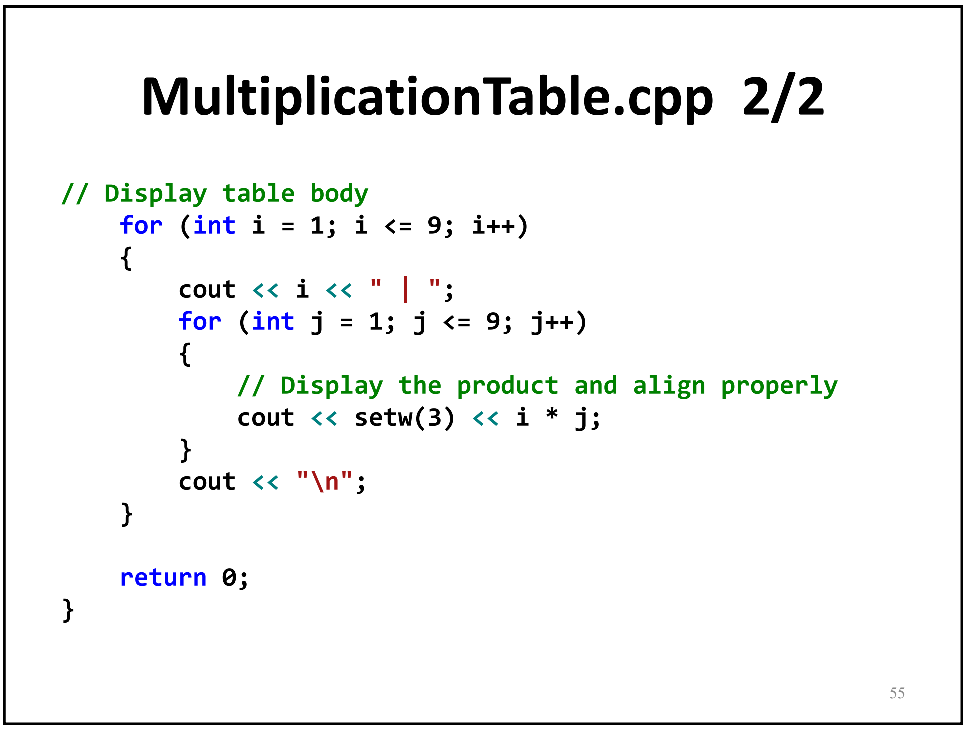
```
          Multiplication Table
   |   1   2   3   4   5   6   7   8   9
-----------------------------------------
1  |   1   2   3   4   5   6   7   8   9
2  |   2   4   6   8  10  12  14  16  18
3  |   3   6   9  12  15  18  21  24  27
4  |   4   8  12  16  20  24  28  32  36
5  |   5  10  15  20  25  30  35  40  45
6  |   6  12  18  24  30  36  42  48  54
7  |   7  14  21  28  35  42  49  56  63
8  |   8  16  24  32  40  48  56  64  72
9  |   9  18  27  36  45  54  63  72  81
```

MultiplicationTable

Run

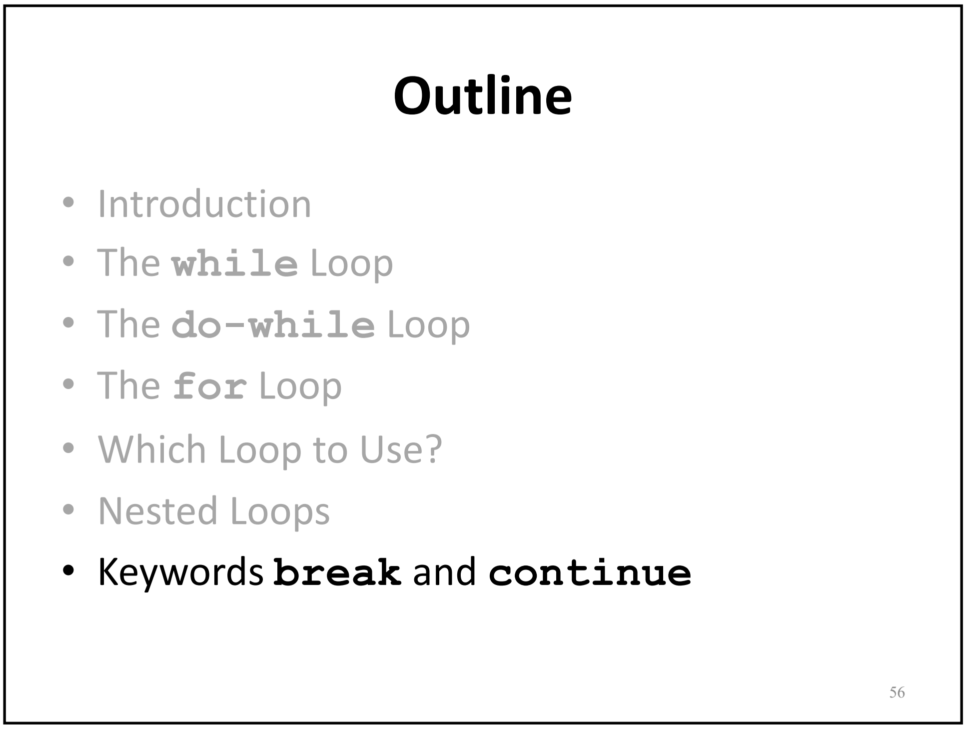
MultiplicationTable.cpp 1/2

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

int main()
{
    cout << "       Multiplication Table\n";

    // Display the number title
    cout << "  | ";
    for (int j = 1; j <= 9; j++)
        cout << setw(3) << j;
    cout << "\n";

    cout << "--------------------------------\n";
```



MultiplicationTable.cpp 2/2

```
// Display table body
    for (int i = 1; i <= 9; i++)
    {
        cout << i << " | ";
        for (int j = 1; j <= 9; j++)
        {
            // Display the product and align properly
            cout << setw(3) << i * j;
        }
        cout << "\n";
    }

    return 0;
}
```



Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

Using break and continue

Use **`break`** in a loop to immediately terminate the loop.

Example: adding integers from 1 to 20 until sum is greater than or equal to 100.

```
while (number < 20)
{
     number++;
     sum += number;
     if (sum >= 100)
         break;
}
```

TestBreak Run

Using break and continue

Use **`continue`** in a loop to proceed to the next iteration.

Example: adding integers from 1 to 20 except 10 and 11.

```
while (number < 20)
   {
       number++;
       if (number == 10 || number == 11)
           continue;
       sum += number;
   }
```

TestContinue Run

Outline

- Introduction
- The **`while`** Loop
- The **`do-while`** Loop
- The **`for`** Loop
- Which Loop to Use?
- Nested Loops
- Keywords **`break`** and **`continue`**

Chapter 6: Functions

Sections 6.1–6.13

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)

Outline

- Introduction
- Defining a Function
- Calling a Function
- `void` Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Introduction

Find the sum of integers from **1** to **10**, from **20** to **37**, and from **35** to **49**, respectively.

Introduction

Write 3 loops

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
cout << "Sum from 1 to 10 is " << sum << endl;

sum = 0;
for (int i = 20; i <= 37; i++)
  sum += i;
cout << "Sum from 20 to 37 is " << sum << endl;

sum = 0;
for (int i = 35; i <= 49; i++)
  sum += i;
cout << "Sum from 35 to 49 is " << sum << endl;
```

Introduction

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
cout << "Sum from 1 to 10 is " << sum << endl;

sum = 0;
for (int i = 20; i <= 37; i++)
  sum += i;
cout << "Sum from 20 to 37 is " << sum << endl;

sum = 0;
for (int i = 35; i <= 49; i++)
  sum += i;
cout << "Sum from 35 to 49 is " << sum << endl;
```

Very similar 3 loops

Introduction

Functions can be used to define reusable code and organize and simplify code.

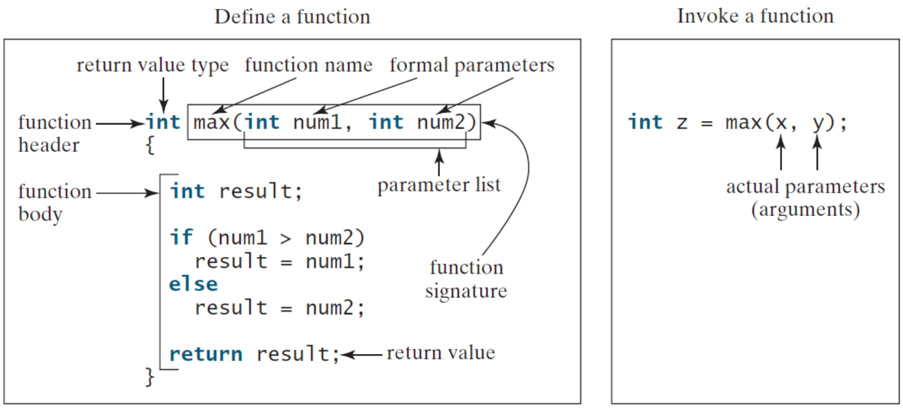
```
int sum(int i1, int i2)
{
    int sum = 0;
    for (int i = i1; i <= i2; i++)
        sum += i;
    return sum;
}

int main()
{
    cout << "Sum from 1 to 10 is " << sum(1, 10) << endl;
    cout << "Sum from 20 to 37 is " << sum(20, 37) << endl;
    cout << "Sum from 35 to 49 is " << sum(35, 49) << endl;
    return 0;
}
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Defining a Function

- A function is a collection of statements that are grouped together to perform an operation.
- A function definition consists of its function name, parameters, return value type, and body.



```
int max(int num1, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

```
int z = max(x, y);
```

Defining Functions, cont.

- *Function signature* is the combination of the function name and the parameter list.
- The variables defined in the function header are known as *formal parameters*.
- When a function is invoked, you pass a value to the parameter. This value is referred to as *actual parameter* or *argument*.

Defining Functions, cont.

- A Function may return a value.
- The *return value type* is the data type of the value the function returns.
- If the function does not return a value, the return value type is the keyword *void*.

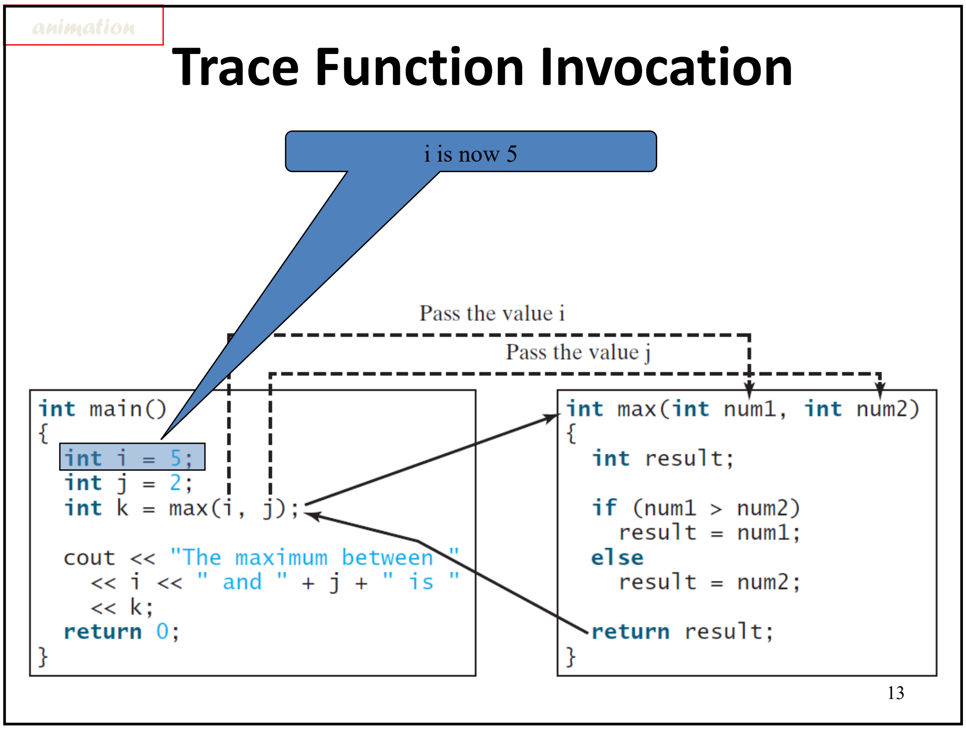
Outline

- Introduction
- Defining a Function
- Calling a Function
- `void` Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Calling a Function

This program demonstrates calling a Function max to return the largest of the **`int`** values

TestMax Run

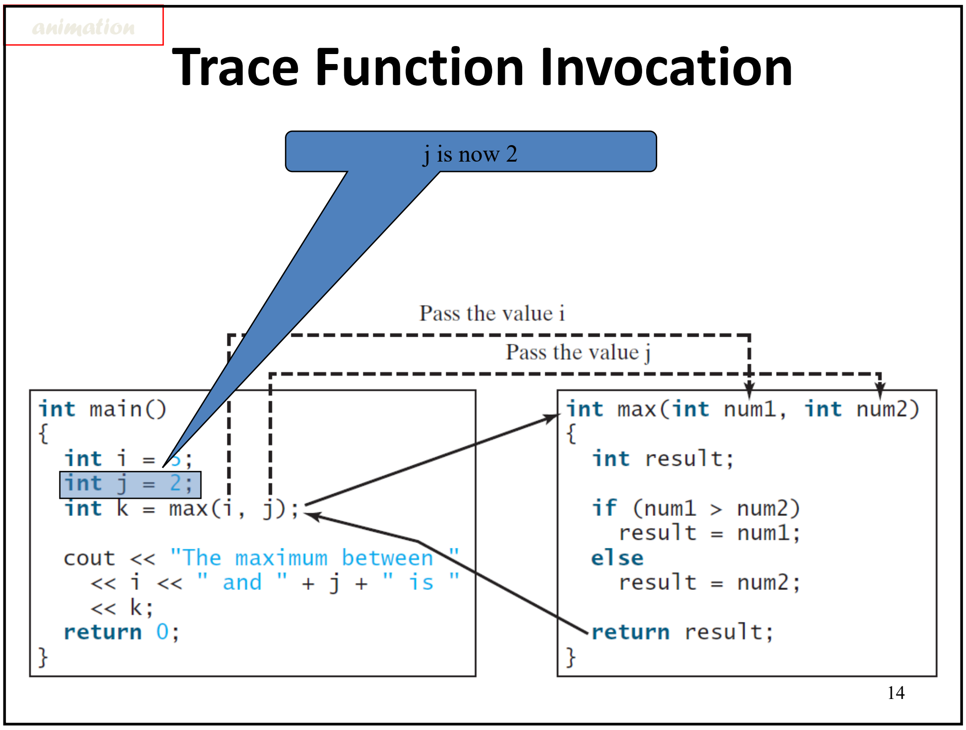


animation

Trace Function Invocation

```
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
int max(int num1, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

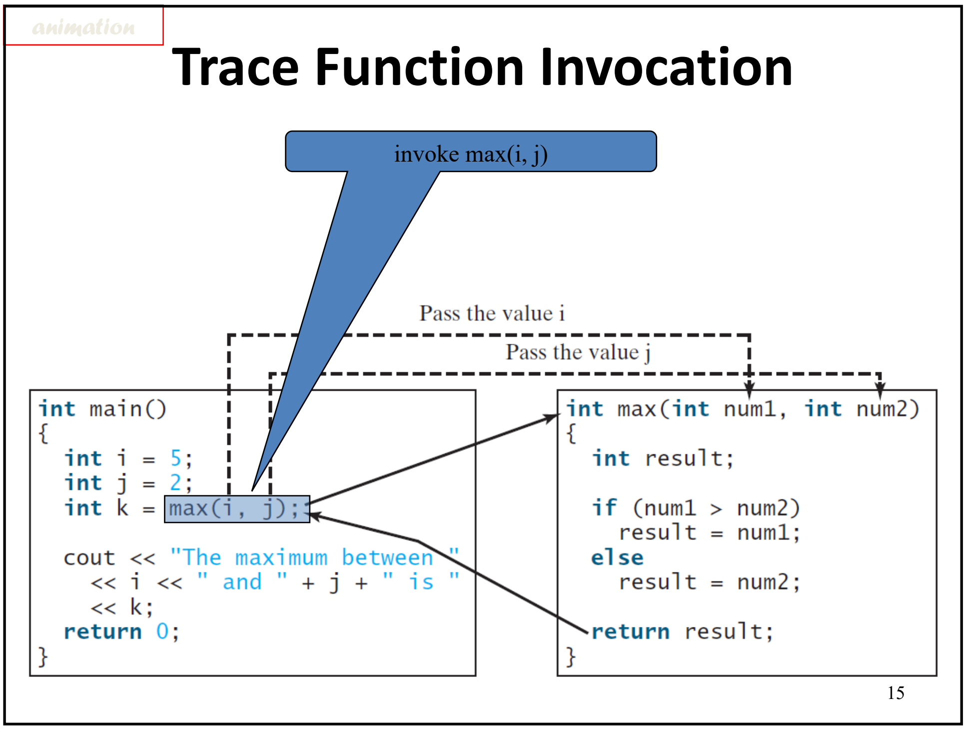


animation

Trace Function Invocation

```
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
int max(int num1, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



animation

Trace Function Invocation

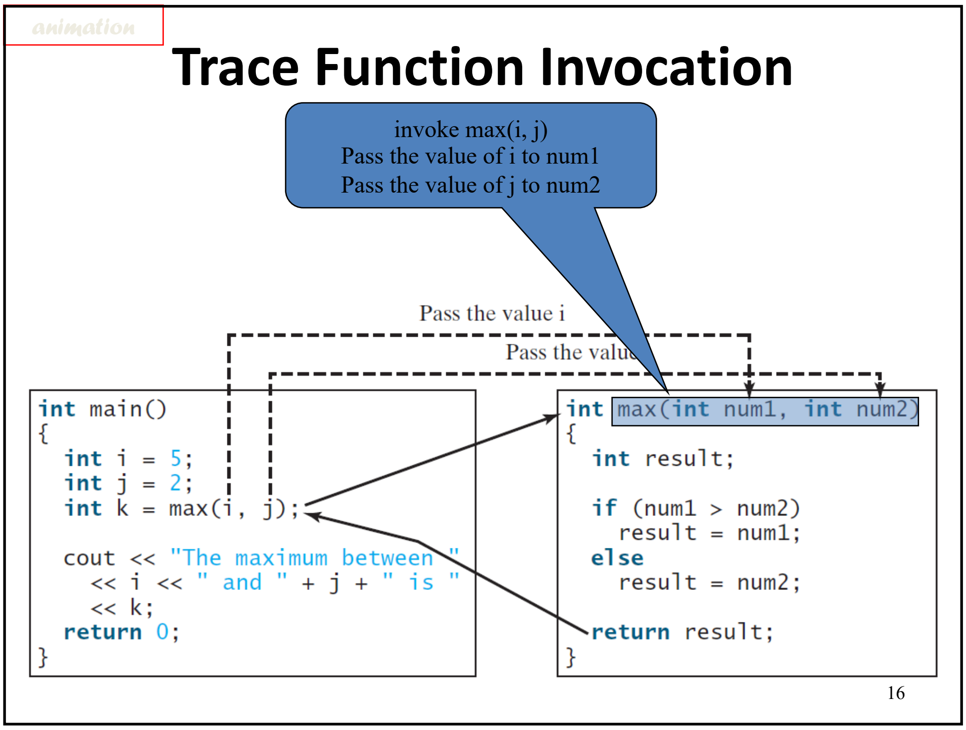
invoke max(i, j)

Pass the value i

Pass the value j

```
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
int max(int num1, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



animation

Trace Function Invocation

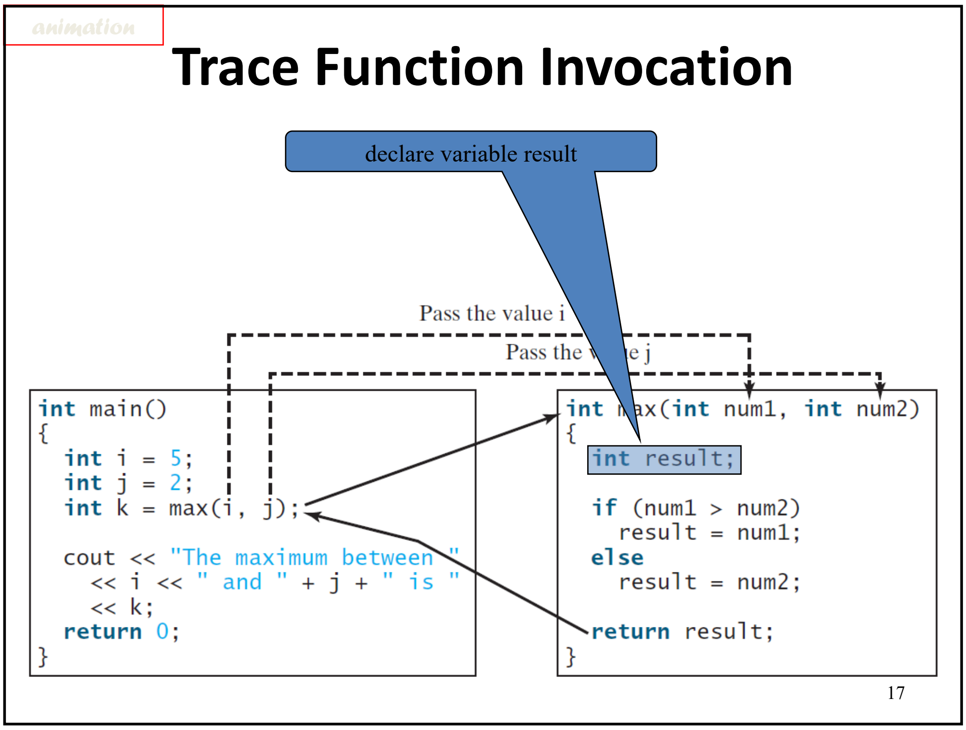
invoke max(i, j)
Pass the value of i to num1
Pass the value of j to num2

Pass the value i

Pass the valu

```
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
int max(int num1, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

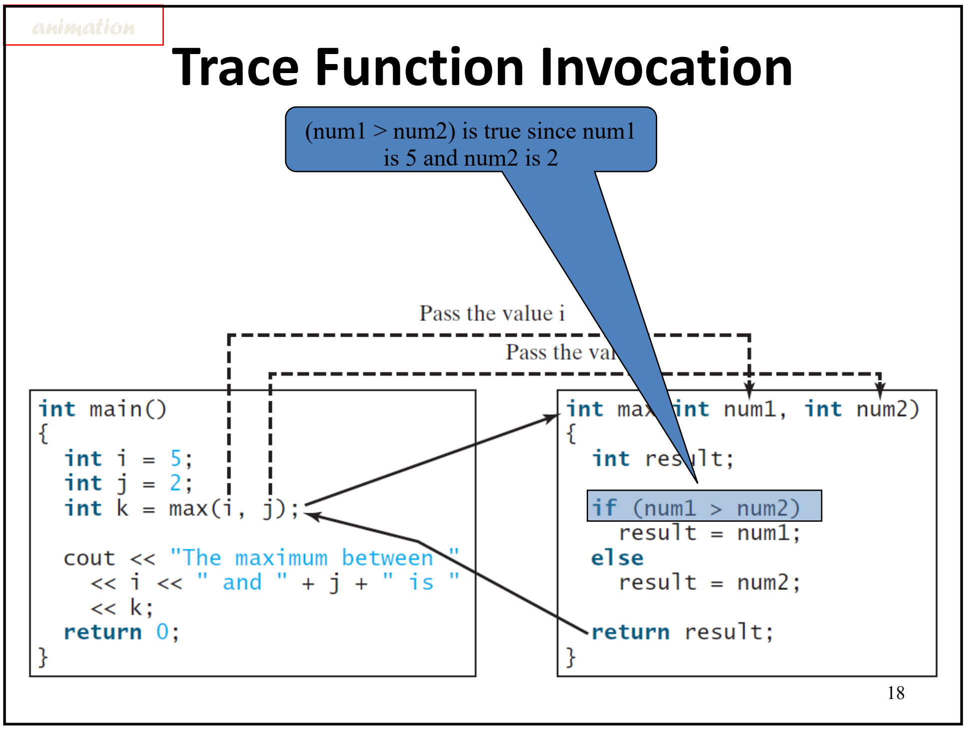


Trace Function Invocation

declare variable result

```
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
int max(int num1, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

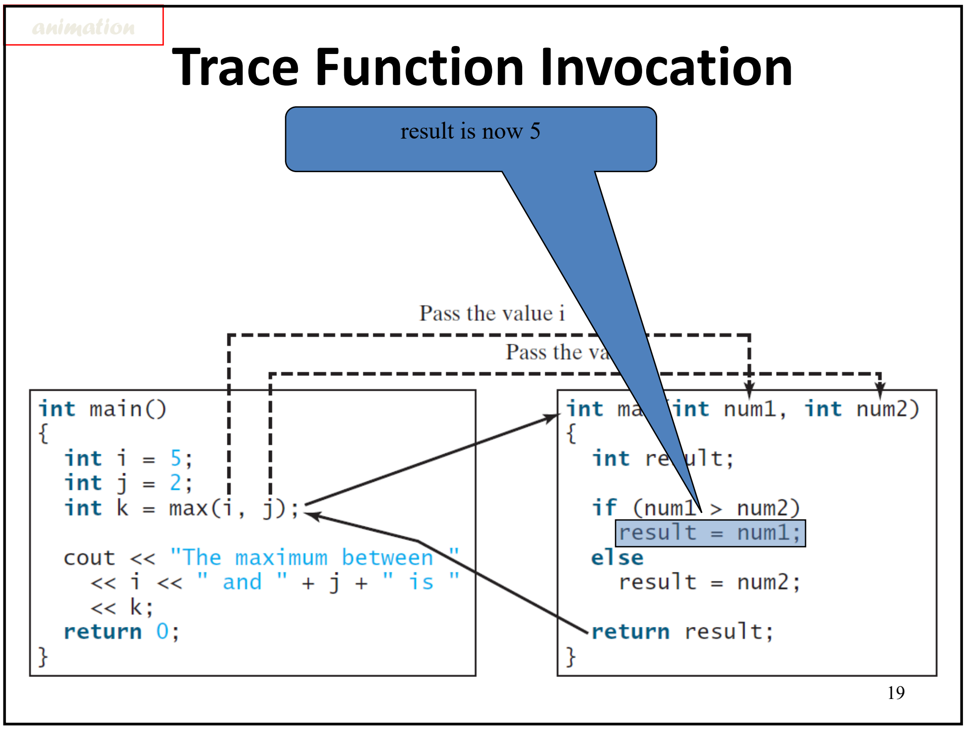


Trace Function Invocation

(num1 > num2) is true since num1 is 5 and num2 is 2

```
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
int max(int num1, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

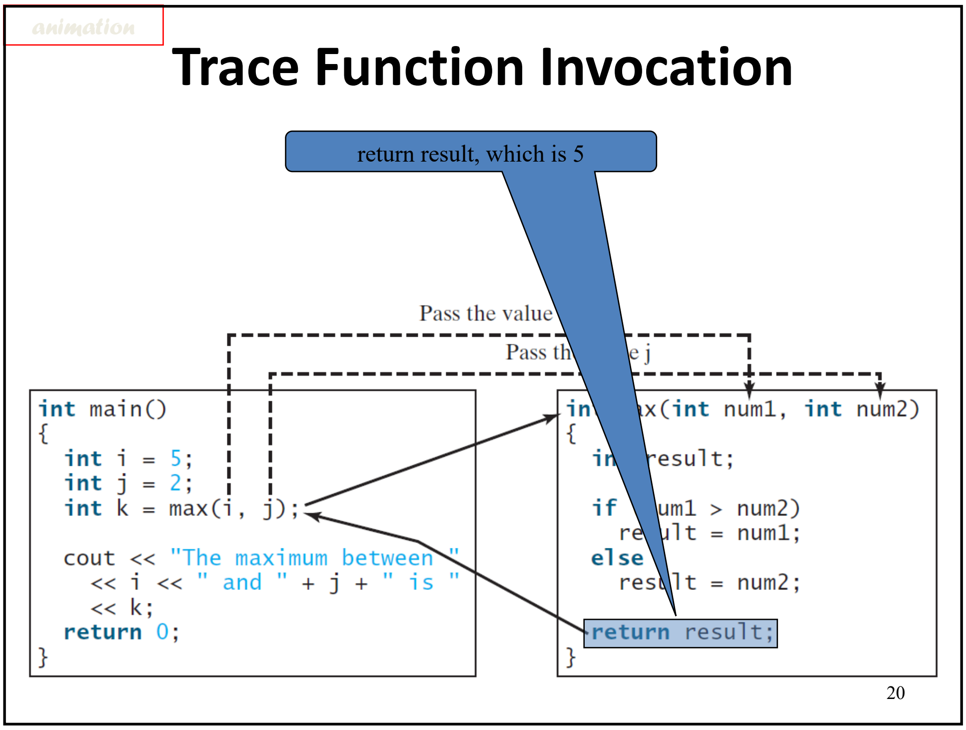


Trace Function Invocation

result is now 5

```
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
int ma  int num1, int num2)
{
  int re ult;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

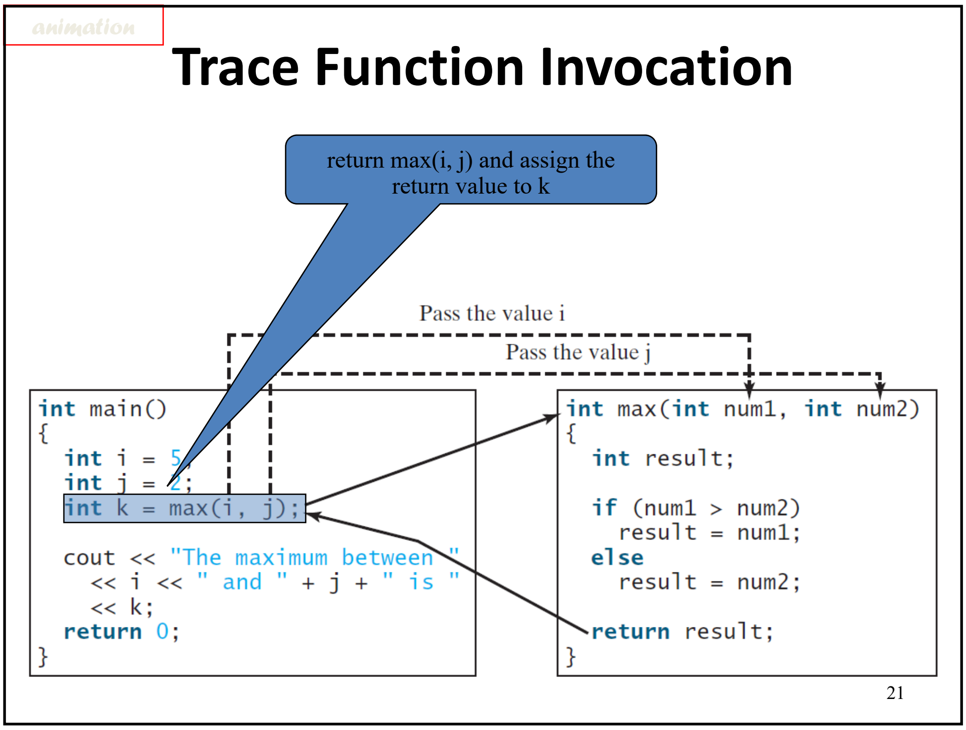


Trace Function Invocation

return result, which is 5

```
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
in    x(int num1, int num2)
{
  in  result;

  if   um1 > num2)
    re ult = num1;
  else
    res lt = num2;

  return result;
}
```

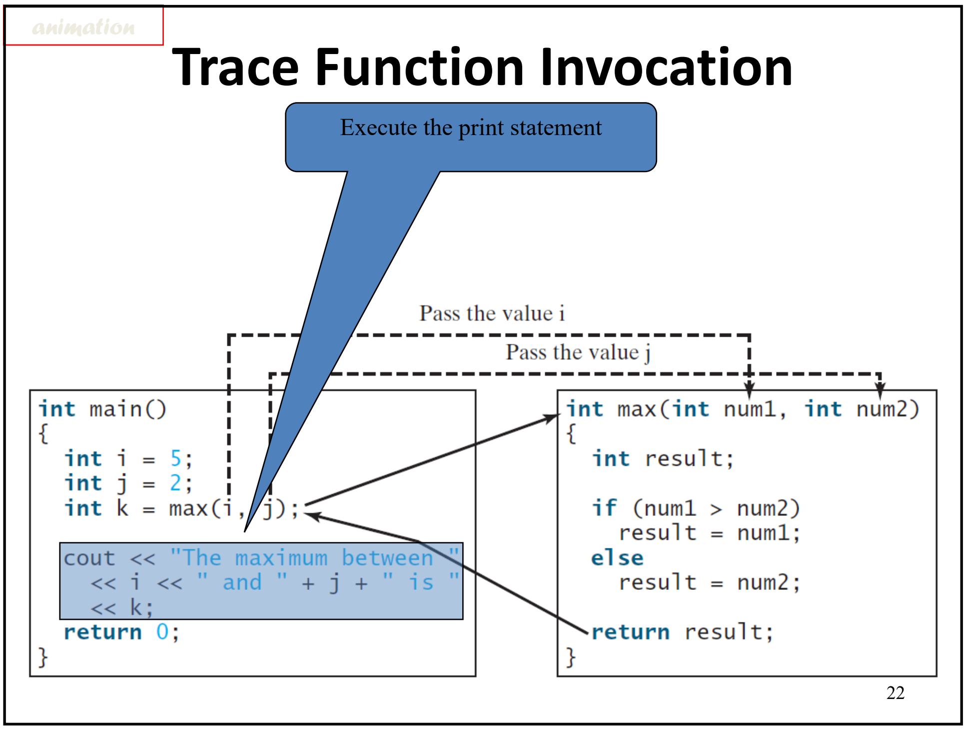


Trace Function Invocation

return max(i, j) and assign the return value to k

```
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
int max(int num1, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```



Trace Function Invocation

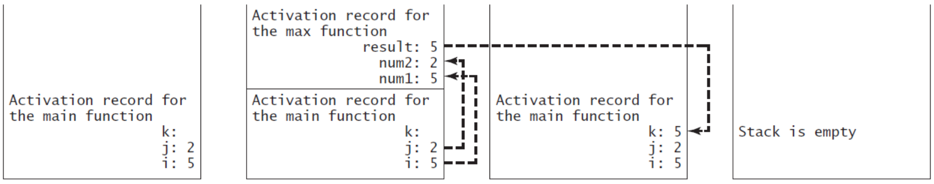
Execute the print statement

```
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
int max(int num1, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Call Stacks

- Each time a function is invoked, the system creates an *activation record*.
- The activation record is placed in an area of memory known as a *call stack*.



(a) The main function is invoked.

(b) The max function is invoked.

(c) The max function is finished and the return value is sent to k.

(d) The main function is finished.

Outline

- Introduction
- Defining a Function
- Calling a Function
- `void` Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

void Functions

A void function does not return a value.

Want to print the grade for a given score.

Two solutions:

1. **`printGrade`** prints the grade
2. **`getGrade`** prints the grade

TestVoidFunction Run

TestReturnGradeFunction Run

void Functions

```
void printGrade(double score)
{
    if (score >= 90.0)
        cout << 'A' << endl;
    else if (score >= 80.0)
        cout << 'B' << endl;
    else if (score >= 70.0)
        cout << 'C' << endl;
    else if (score >= 60.0)
        cout << 'D' << endl;
    else
        cout << 'F' << endl;
}

int main()
{
    cout << "Enter a score: ";
    double score;
    cin >> score;

    cout << "The grade is ";
    printGrade(score);
    return 0;
}
```

```
char getGrade(double score)
{
    if (score >= 90.0)
        return 'A';
    else if (score >= 80.0)
        return 'B';
    else if (score >= 70.0)
        return 'C';
    else if (score >= 60.0)
        return 'D';
    else
        return 'F';
}

int main()
{
 cout << "Enter a score: ";
    double score;
    cin >> score;

    cout << "The grade is ";
    cout << getGrade(score) << endl;
    return 0;
}
```

Terminating a Program

- You can terminate a program at abnormal conditions by calling **`exit(n)`**.
- Select the integer **`n`** to specify the error type.

```
void printGrade(double score)
{
    if (score < 0 || score > 100)
    {
        cout << "Invalid score" << endl;
        exit(1);
    }
    if (score >= 90.0)
        cout << 'A';
    else if (score >= 80.0)
        cout << 'B';
    else if (score >= 70.0)
        cout << 'C';
    else if (score >= 60.0)
        cout << 'D';
    else
        cout << 'F';
}
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- `void` Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Passing Arguments by Value

- By default, the arguments are passed by value to parameters when invoking a function.
- When calling a function, you need to provide arguments, which must be given in the same order as their respective parameters in the function signature.
- The shown code prints **a** character **3** times.

```
void nPrint(char ch, int n)
{
    for (int i = 0; i < n; i++)
        cout << ch;
}

nPrint('a', 3);

aaa
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- `void` Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Modularizing Code

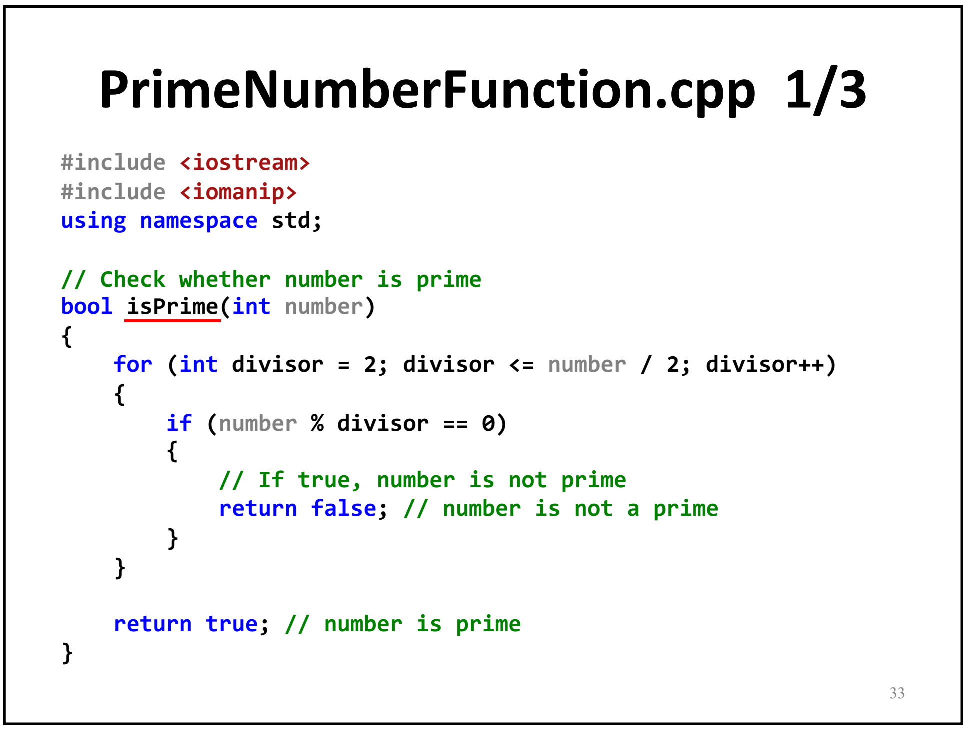
- Modularizing makes the code easy to maintain and debug and enables the code to be reused.
- These two examples use functions to reduce complexity.

GreatestCommonDivisorFunction Run

PrimeNumberFunction Run

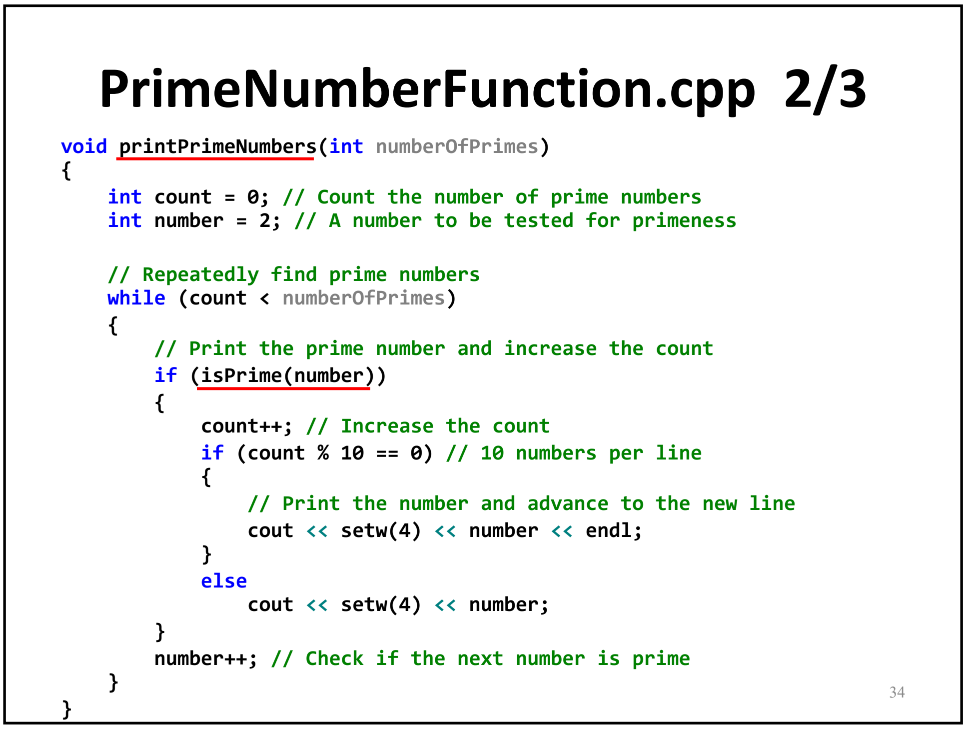
GreatestCommonDivisorFunction.cpp

```
int gcd(int n1, int n2)
{
    int gcd = 1; // Initial gcd is 1
    int k = 2;   // Possible gcd

    while (k <= n1 && k <= n2)
    {
        if (n1 % k == 0 && n2 % k == 0)
            gcd = k; // Update gcd
        k++;
    }
    return gcd; // Return gcd
}
int main()
{
    ...
    cout << "The greatest common divisor for " << n1 <<
        " and " << n2 << " is " << gcd(n1, n2) << endl;

    return 0;
}
```



PrimeNumberFunction.cpp 1/3

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

// Check whether number is prime
bool isPrime(int number)
{
    for (int divisor = 2; divisor <= number / 2; divisor++)
    {
        if (number % divisor == 0)
        {
            // If true, number is not prime
            return false; // number is not a prime
        }
    }

    return true; // number is prime
}
```



PrimeNumberFunction.cpp 2/3

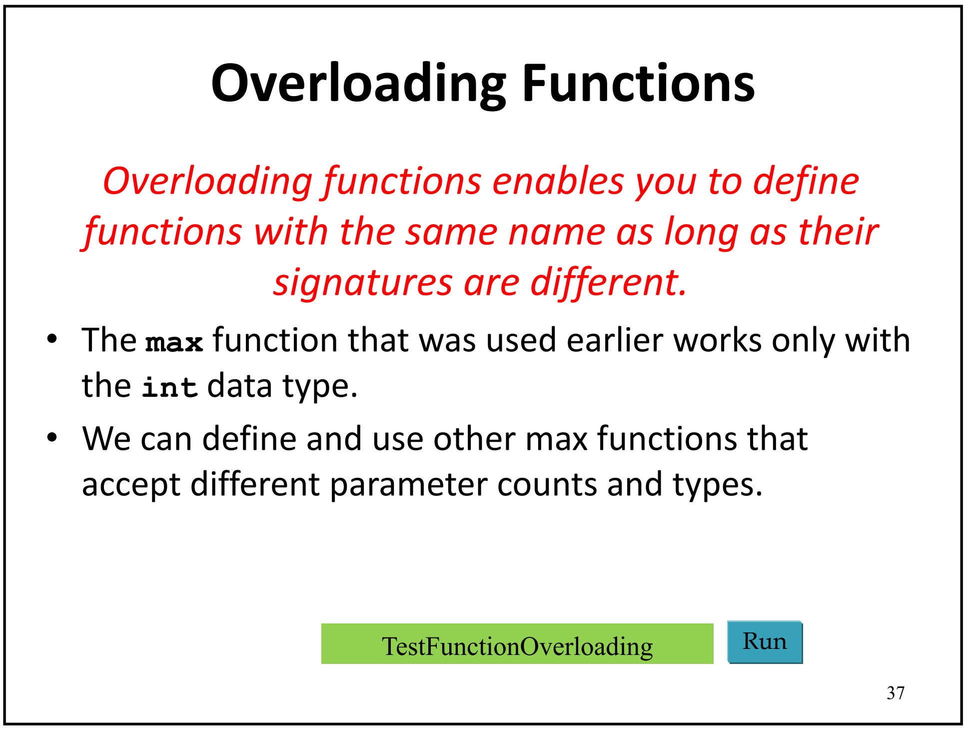
```cpp
void printPrimeNumbers(int numberOfPrimes)
{
    int count = 0; // Count the number of prime numbers
    int number = 2; // A number to be tested for primeness

    // Repeatedly find prime numbers
    while (count < numberOfPrimes)
    {
        // Print the prime number and increase the count
        if (isPrime(number))
        {
            count++; // Increase the count
            if (count % 10 == 0) // 10 numbers per line
            {
                // Print the number and advance to the new line
                cout << setw(4) << number << endl;
            }
            else
                cout << setw(4) << number;
        }
        number++; // Check if the next number is prime
    }
}
```

PrimeNumberFunction.cpp 3/3

```
int main()
{
    cout << "The first 50 prime numbers are \n";
    printPrimeNumbers(50);

    return 0;
}
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- `void` Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

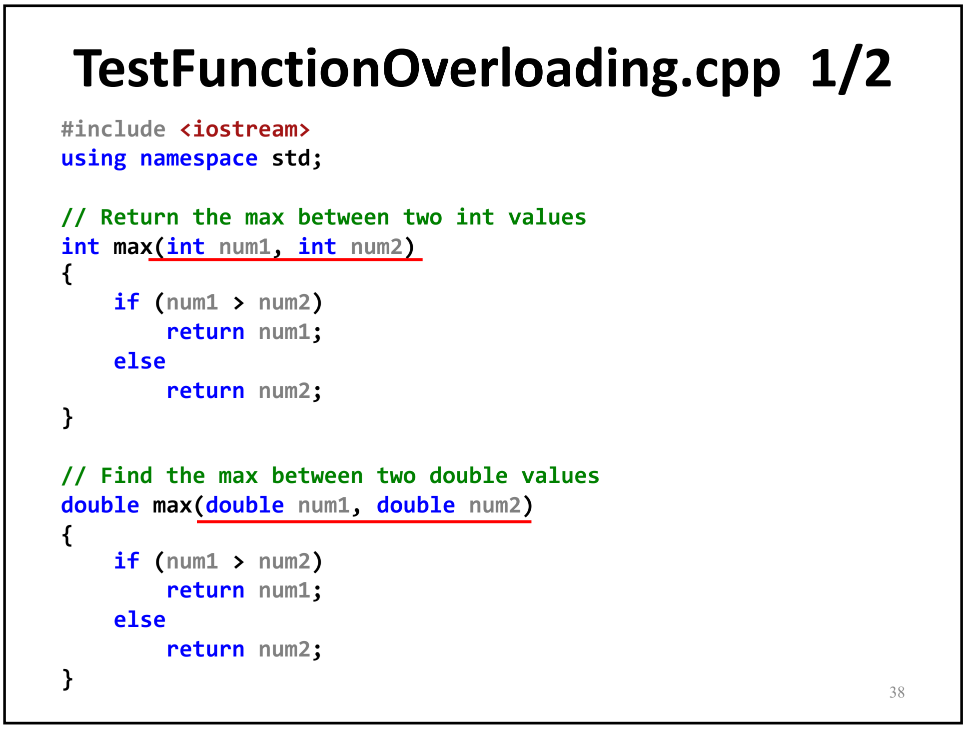


Overloading Functions

Overloading functions enables you to define functions with the same name as long as their signatures are different.

- The **`max`** function that was used earlier works only with the **`int`** data type.
- We can define and use other max functions that accept different parameter counts and types.

TestFunctionOverloading Run



TestFunctionOverloading.cpp 1/2

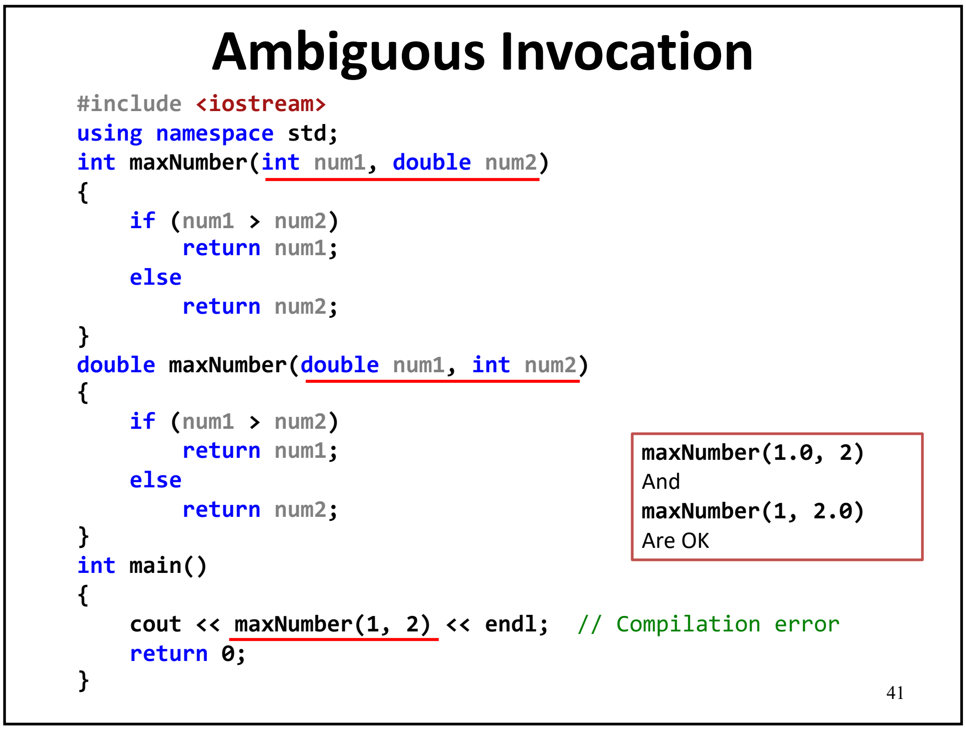
```
#include <iostream>
using namespace std;

// Return the max between two int values
int max(int num1, int num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}

// Find the max between two double values
double max(double num1, double num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
```

TestFunctionOverloading.cpp 2/2

```cpp
// Return the max among three double values
double max(double num1, double num2, double num3)
{
    return max(max(num1, num2), num3);
}

int main()
{
    // Invoke the max function with int parameters
    cout << "The max between 3 and 4 is " << max(3, 4) << endl;

    // Invoke the max function with the double parameters
    cout << "The maximum between 3.0 and 5.4 is "
        << max(3.0, 5.4) << endl;

    // Invoke the max function with three double parameters
    cout << "The maximum between 3.0, 5.4, and 10.14 is "
        << max(3.0, 5.4, 10.14) << endl;

    return 0;
}
```

Ambiguous Invocation

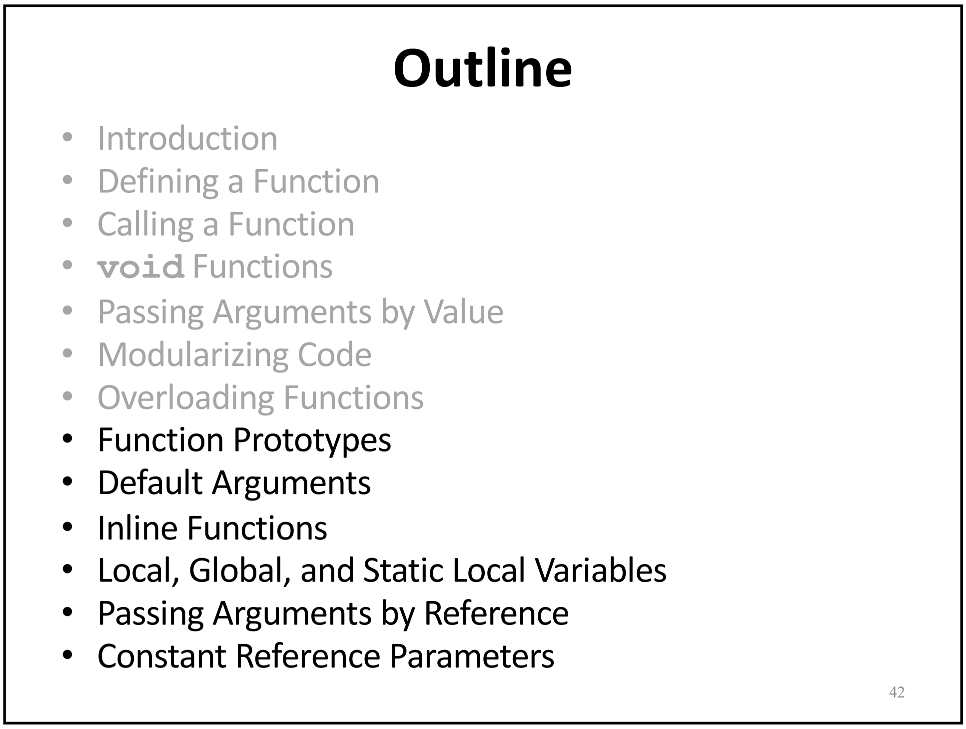
Sometimes there may be two or more possible matches for an invocation of a function, but the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*. Ambiguous invocation is a compilation error.



Ambiguous Invocation

```
#include <iostream>
using namespace std;
int maxNumber(int num1, double num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
double maxNumber(double num1, int num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
int main()
{
    cout << maxNumber(1, 2) << endl;  // Compilation error
    return 0;
}
```

`maxNumber(1.0, 2)`
And
`maxNumber(1, 2.0)`
Are OK



Outline

- Introduction
- Defining a Function
- Calling a Function
- `void` Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Function Prototypes

- Before a function is called, it must be declared first.
- One way to ensure it is to place the declaration before all function calls.
- Another way to approach it is to declare a function prototype before the function is called.
- A *function prototype* is a function declaration without implementation.
- The implementation can be given later in the program.

TestFunctionPrototype Run

TestFunctionPrototype.cpp

```
#include <iostream>
using namespace std;
// Function prototype
int max(int num1, int num2);
double max(double num1, double num2);
double max(double num1, double num2, double num3);

int main()
{
    // Invoke the max function with int parameters
    cout << "The maximum between 3 and 4 is " <<
        max(3, 4) << endl;
    ...
}
// Return the max between two int values
int max(int num1, int num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
...
```

Or simply:

```
int max(int, int);
double max(double, double);
double max(double, double, double);
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- `void` Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Default Arguments

You can define default values for parameters in a function.

The default values are passed to the parameters when a function is invoked without the arguments.

DefaultArgumentDemo Run

DefaultArgumentDemo.cpp

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

// Display area of a circle
void printArea(double radius = 1)
{
    double area = radius * radius * 3.14159;
    cout << "area is " << area << endl;
}

int main()
{
    printArea();
    printArea(4);

    return 0;
}
```

Default Arguments

- When a function contains a mixture of parameters with and without default values, those with default values must be declared last.

```
void t1(int x, int y = 0, int z); // Illegal
void t3(int x, int y = 0, int z = 0); // Legal
```

- When an argument is left out of a function, all arguments that come after it must be left out as well.

```
t3(1,  , 20); // Illegal
t3(1); // Parameters y and z are assigned a default value
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- `void` Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

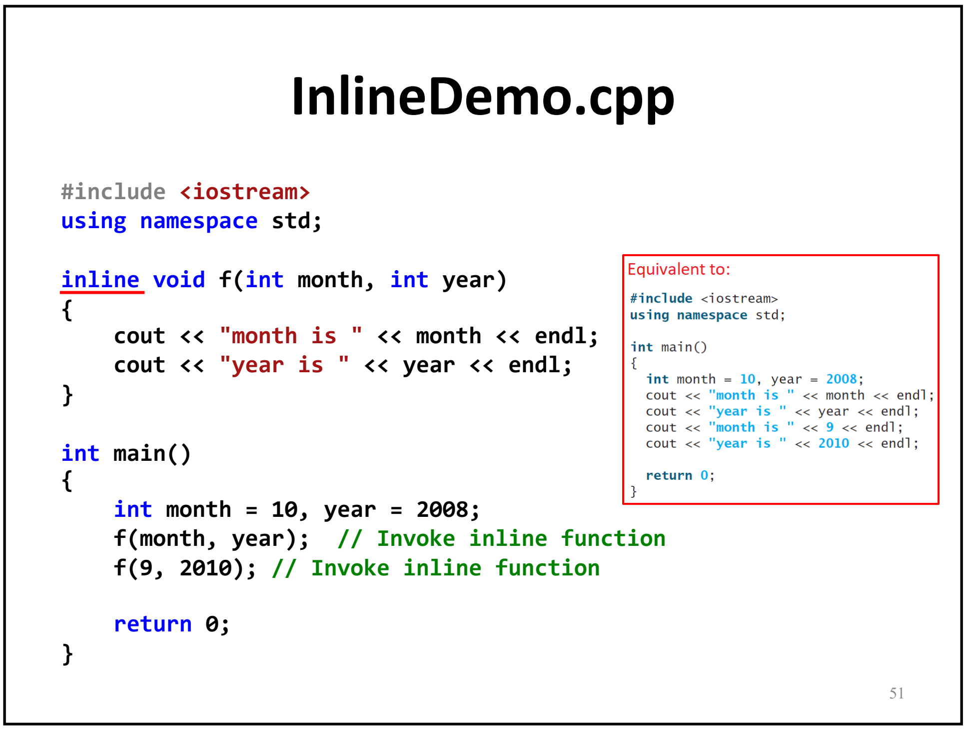
Inline Functions

C++ provides inline functions for improving performance for short functions.

- Inline functions are not called; rather, the compiler copies the function code in line at the point of each invocation.
- To specify an inline function, precede the function declaration with the **`inline`** keyword.
- Inline functions are desirable for short functions but not for long ones.

InlineDemo Run

InlineExpandedDemo

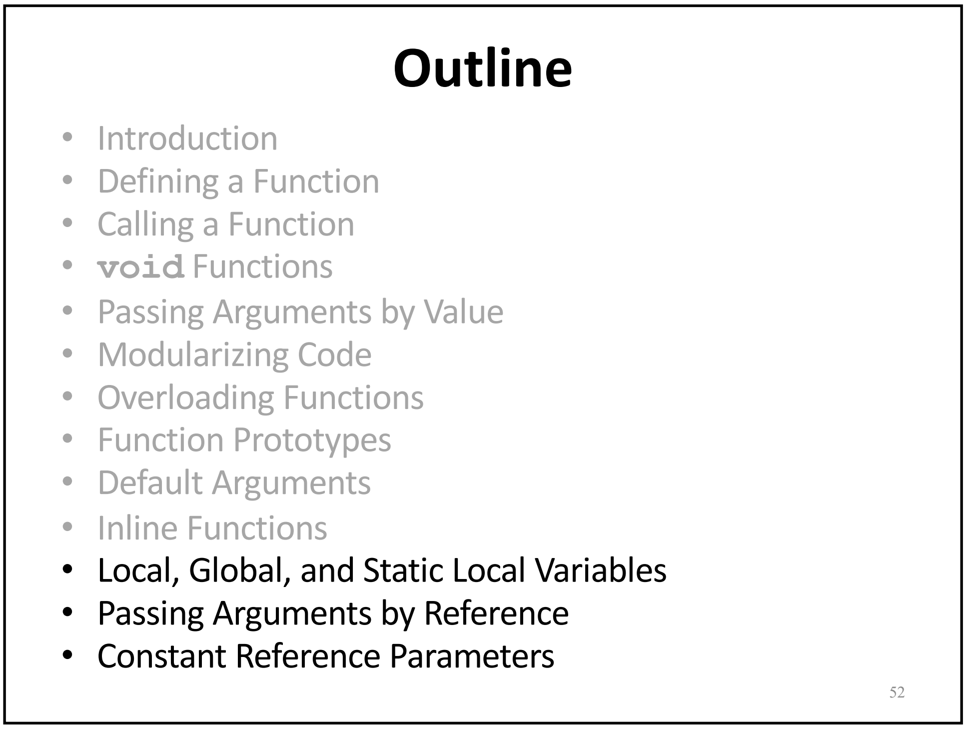


InlineDemo.cpp

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

inline void f(int month, int year)
{
    cout << "month is " << month << endl;
    cout << "year is " << year << endl;
}

int main()
{
    int month = 10, year = 2008;
    f(month, year);  // Invoke inline function
    f(9, 2010); // Invoke inline function

    return 0;
}
```

Equivalent to:

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

int main()
{
  int month = 10, year = 2008;
  cout << "month is " << month << endl;
  cout << "year is " << year << endl;
  cout << "month is " << 9 << endl;
  cout << "year is " << 2010 << endl;

  return 0;
}
```



Outline

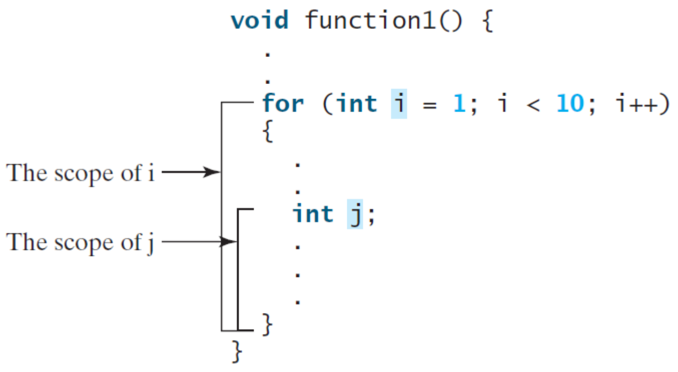
- Introduction
- Defining a Function
- Calling a Function
- `void` Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Scope of Variables

- *Scope*: the part of the program where the variable can be referenced.
- The scope of a variable starts from its declaration and continues to the end of the block that contains the variable.
- A variable can be declared as a *local*, a *global*, or a *static* local.
- A *local variable*: a variable defined inside a function.
- You can declare a local variable with the same name in different blocks.

Scope of Local Variables

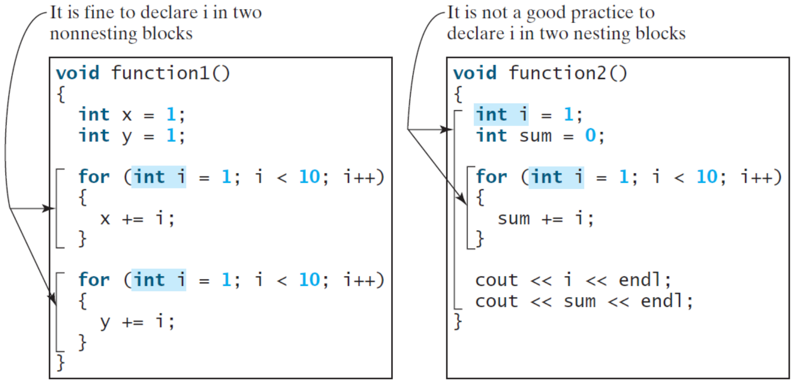
- A variable declared in the initial action part of a `for` loop has its scope in the entire loop.
- A variable declared inside a `for` loop body has its scope limited the rest of the loop body.



```
void function1() {
  .
  .
  for (int i = 1; i < 10; i++)
  {
    .
    .
    int j;
    .
    .
    .
  }
}
```

Scope of Local Variables, cont.

- It is acceptable to declare a local variable with the same name in different non-nesting blocks.
- Avoid using same variable name in nesting blocks to minimize making mistakes.



It is fine to declare i in two nonnesting blocks

```
void function1()
{
  int x = 1;
  int y = 1;

  for (int i = 1; i < 10; i++)
  {
    x += i;
  }

  for (int i = 1; i < 10; i++)
  {
    y += i;
  }
}
```

It is not a good practice to declare i in two nesting blocks

```
void function2()
{
  int i = 1;
  int sum = 0;

  for (int i = 1; i < 10; i++)
  {
    sum += i;
  }

  cout << i << endl;
  cout << sum << endl;
}
```

Global Variables

- *Global variables* are declared outside all functions and are accessible to all functions in their scope.
- Local variables do not have default values, but global variables are defaulted to zero.

VariableScopeDemo Run

VariableScopeDemo.cpp

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

void t1(); // Function prototype
void t2(); // Function prototype

int main()
{
    t1();
    t2();

    return 0;
}

int y; // Global variable
       // default to 0
```

```
void t1()
{
    int x = 1;
    cout << "x is " << x << endl;
    cout << "y is " << y << endl;
    x++;
    y++;
}

void t2()
{
    int x = 1;
    cout << "x is " << x << endl;
    cout << "y is " << y << endl;
}
```

```
x is 1
y is 0
x is 1
y is 1
```

Unary Scope Resolution

If a local variable name is the same as a global variable name, you can access the global variable using `::globalVariable`. The `::` operator is known as the *unary scope resolution*.

```
#include <iostream>
using namespace std;
int v1 = 10;
int main()
{
    int v1 = 5;
    cout << "local variable v1 is " << v1 << endl;
    cout << "global variable v1 is " << ::v1 << endl;
    return 0;
}
```

```
local variable v1 is 5
global variable v1 is 10
```

Static Local Variables

- After a function completes its execution, all its local variables are destroyed.
- To retain the value stored in local variables so that they can be used in the next call, use *static local variables*.
- Static local variables are permanently allocated in the memory for the lifetime of the program.
- To declare a static variable, use the keyword `static`.

StaticVariableDemo Run

StaticVariableDemo.cpp

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

void t1(); // Function prototype

int main()
{
    t1();
    t1();
    return 0;
}

void t1()
{
    static int x = 1; // Static local
    int y = 1;        // Local, not static
    x++;
    y++;
    cout << "x is " << x << endl;
    cout << "y is " << y << endl;
}
```

```
x is 2
y is 2
x is 3
y is 2
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- `void` Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Pass by Value

- When you invoke a function with a parameter, the value of the argument is passed to the parameter. This is referred to as *pass-by-value*.
- The variable is not affected, regardless of the changes made to the parameter inside the function.

Increment Run

Increment.cpp

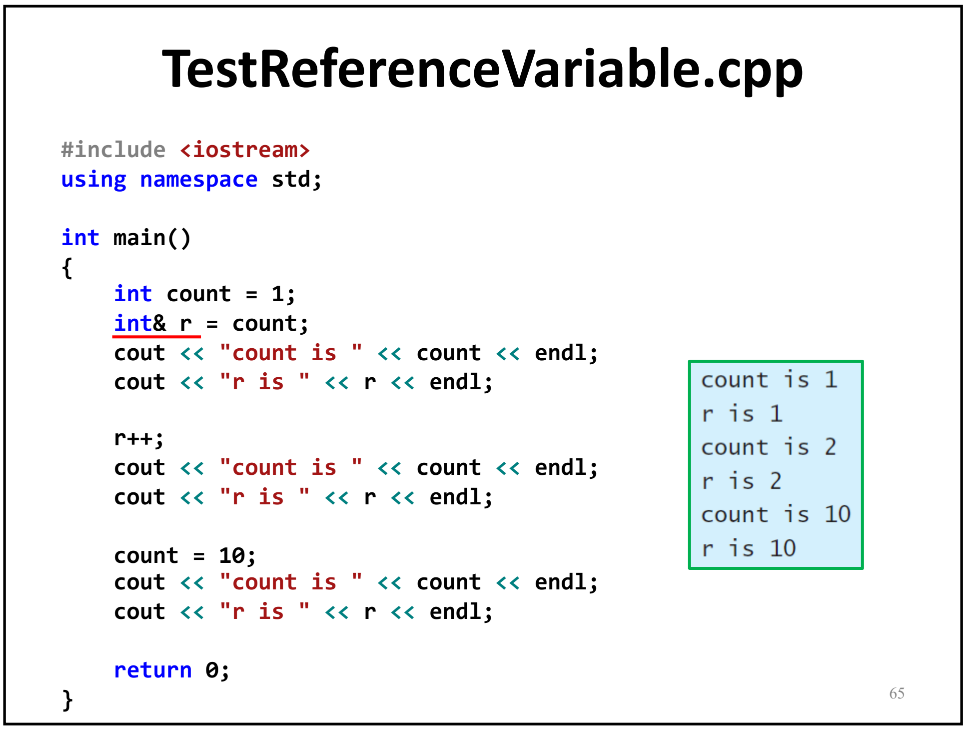
```
#include <iostream>
using namespace std;

void increment(int n)
{
    n++;
    cout << "\tn inside the function is " << n << endl;
}

int main()
{
    int x = 1;
    cout << "Before the call, x is " << x << endl;
    increment(x);
    cout << "after the call, x is " << x << endl;

    return 0;
}
```

```
Before the call, x is 1
  n inside the function is 2
after the call, x is 1
```

Reference Variables

- A *reference variable* can be used as a function parameter to reference the original variable.
- A reference variable is an alias for another variable.
- Any changes made through the reference variable are actually performed on the original variable.
- To declare a reference variable, place the ampersand (**&**) in front of the name.

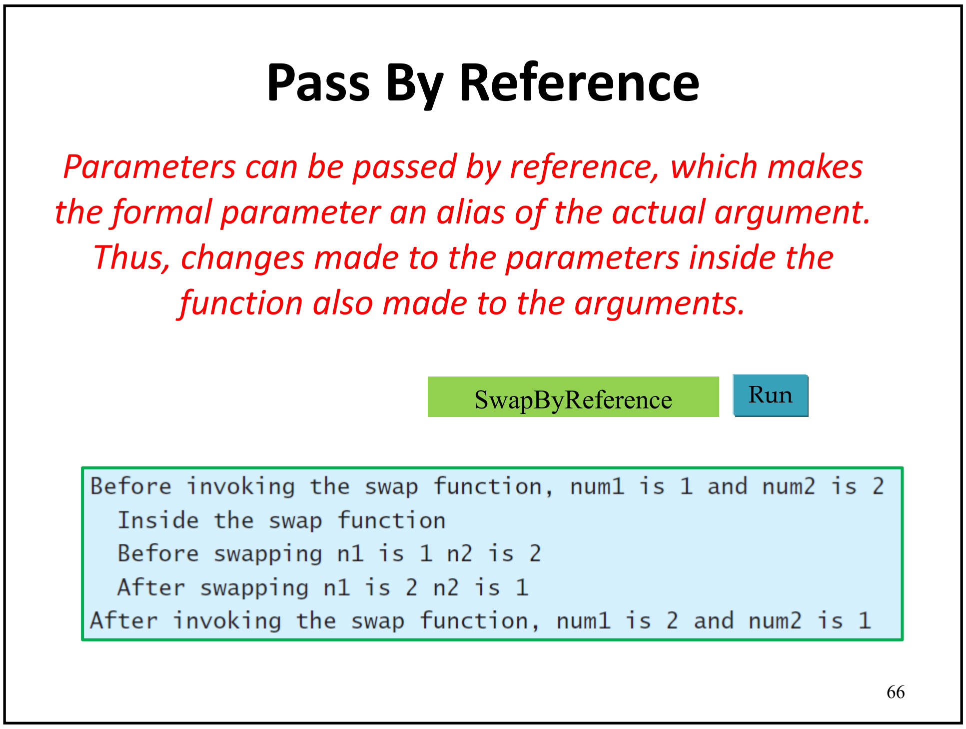
TestReferenceVariable Run



TestReferenceVariable.cpp

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

int main()
{
    int count = 1;
    int& r = count;
    cout << "count is " << count << endl;
    cout << "r is " << r << endl;

    r++;
    cout << "count is " << count << endl;
    cout << "r is " << r << endl;

    count = 10;
    cout << "count is " << count << endl;
    cout << "r is " << r << endl;

    return 0;
}
```

```
count is 1
r is 1
count is 2
r is 2
count is 10
r is 10
```



Pass By Reference

Parameters can be passed by reference, which makes the formal parameter an alias of the actual argument. Thus, changes made to the parameters inside the function also made to the arguments.

```
Before invoking the swap function, num1 is 1 and num2 is 2
  Inside the swap function
  Before swapping n1 is 1 n2 is 2
  After swapping n1 is 2 n2 is 1
After invoking the swap function, num1 is 2 and num2 is 1
```

SwapByReference.cpp 1/2

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

// Swap two variables
void swap(int& n1, int& n2)
{
    cout << "\tInside the swap function" << endl;
    cout << "\tBefore swapping n1 is " << n1 <<
        " n2 is " << n2 << endl;

    // Swap n1 with n2
    int temp = n1;
    n1 = n2;
    n2 = temp;

    cout << "\tAfter swapping n1 is " << n1 <<
        " n2 is " << n2 << endl;
}
```

SwapByReference.cpp 2/2

```
int main()
{
    // Declare and initialize variables
    int num1 = 1;
    int num2 = 2;

    cout << "Before invoking the swap function, num1 is "
        << num1 << " and num2 is " << num2 << endl;

    // Invoke the swap function to attempt to swap two variables
    swap(num1, num2);

    cout << "After invoking the swap function, num1 is " << num1
         << " and num2 is " << num2 << endl;

    return 0;
}
```

Pass-by-Value vs. Pass-by-Reference

- In *pass-by-value*, the actual parameter and its formal parameter are independent variables.
- In *pass-by-reference*, the actual parameter and its formal parameter refer to the same variable.
- *Pass-by-reference* is more efficient than *pass-by-value*. However, the difference is negligible for parameters of primitive types such as `int` and `double`.
- So, if a primitive data type parameter is not changed in the function, you should declare it as *pass-by-value* parameter.

Outline

- Introduction
- Defining a Function
- Calling a Function
- `void` Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Constant Reference Parameters

You can specify a constant reference parameter to prevent its value from being changed by accident.

```
// Return the max between two numbers
int max(const int& num1, const int& num2)
{
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

Outline

- Introduction
- Defining a Function
- Calling a Function
- **`void`** Functions
- Passing Arguments by Value
- Modularizing Code
- Overloading Functions
- Function Prototypes
- Default Arguments
- Inline Functions
- Local, Global, and Static Local Variables
- Passing Arguments by Reference
- Constant Reference Parameters

Chapter 7: Single-Dimensional Arrays and C-Strings

Sections 7.1–7.7, 7.11

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)

Outline

- Introduction
- Array Basics
- Problem: Lotto Numbers
- Problem: Deck of Cards
- Passing Arrays to Functions
- Preventing Changes of Array Arguments in Functions
- Returning Arrays from Functions
- C-Strings

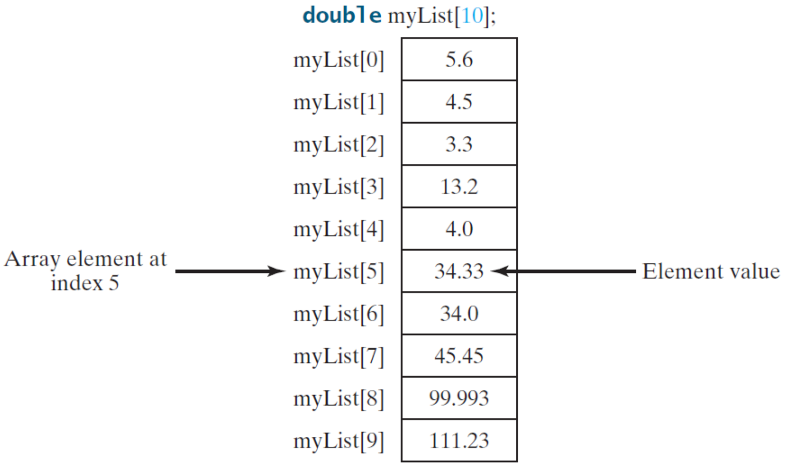
Introduction

- How to read one hundred numbers and compute their average?
- Use **A1**, **A2**, …, **A100**?
- Or use a single array that stores all the numbers?

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

int main()
{
    double numbers[100];
    double sum = 0;

    for (int i = 0; i < 100; i++)
    {
        cout << "Enter a number: ";
        cin >> numbers[i];
        sum += numbers[i];
    }
    double average = sum / 100;
    cout << "Average is " << average
         << endl;
    return 0;
}
```

Introduction

Array is a data structure that represents a collection of the same types of data.



Outline

- Introduction
- Array Basics
- Problem: Lotto Numbers
- Problem: Deck of Cards
- Passing Arrays to Functions
- Preventing Changes of Array Arguments in Functions
- Returning Arrays from Functions
- C-Strings

Declaring Array Variables

```
datatype arrayRefVar[arraySize];
```

Example:

```
double myList[10];
```

C++ requires that the array size used to declare an array must be a constant expression. For example, the following code is illegal:

```
int size = 10;
double myList[size]; // Wrong
```

But it would be OK, if **size** is a constant as follow:

```
const int size = 10;
double myList[size], list2[5]; // Correct
```

Arbitrary Initial Values

When an array is created, its elements are assigned with arbitrary values.

They are not initialized.

Accessing Array Elements

- The array elements are accessed through the index. Array indices are 0-based; that is, they start from 0 to arraySize-1.
- Each element in the array is represented using the following syntax, known as an *indexed variable*:

  `arrayName[index];`

- For example, `myList[9]` represents the last element in the array `myList`.

Using Indexed Variables

- After an array is created, an indexed variable can be used in the same way as a regular variable.
- Examples:

```
myList[2] = myList[0] + myList[1];
myList[3]++;
cout << max(myList[0], myList[1]) << endl;
```

- C++ does not check array's boundary. So, accessing array elements using subscripts beyond the boundary (e.g., `myList[-1]` and `myList[11]`) does not cause syntax errors, but the operating system might report a memory access violation.

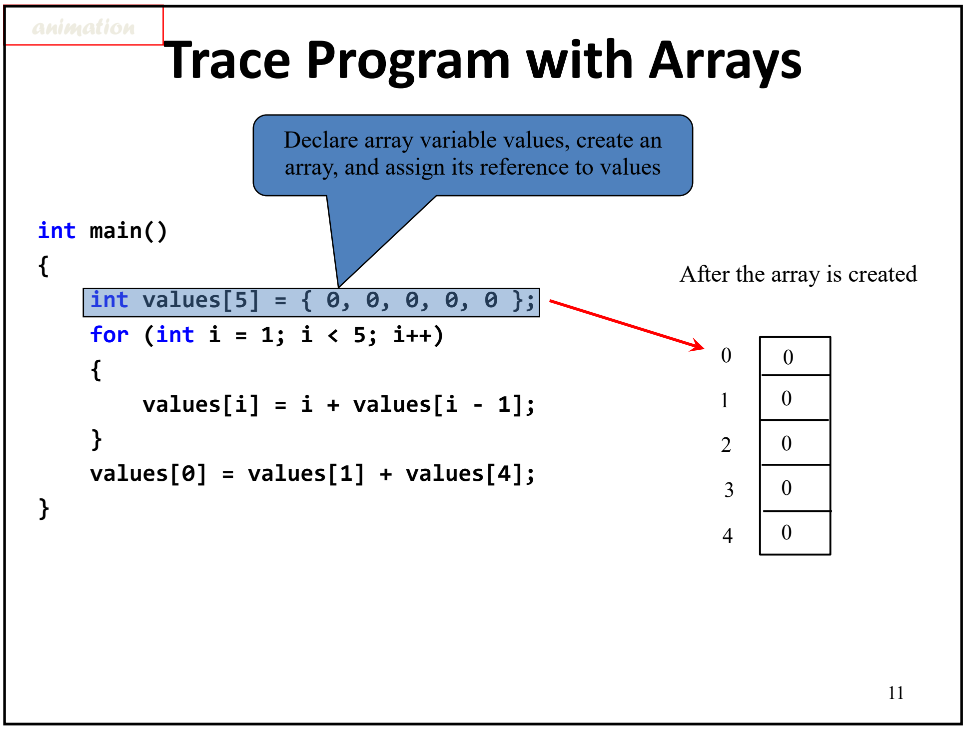
Array Initializers

Declaring, creating, initializing in one step:

```
dataType arrayName[arraySize] = {value0, value1,
     ..., valuek};
```

Examples:

```
double myList[4] = {1.9, 2.9, 3.4, 3.5};
double myList[] = {1.9, 2.9, 3.4, 3.5};
double myList[4] = {1.9, 2.9};
```



animation

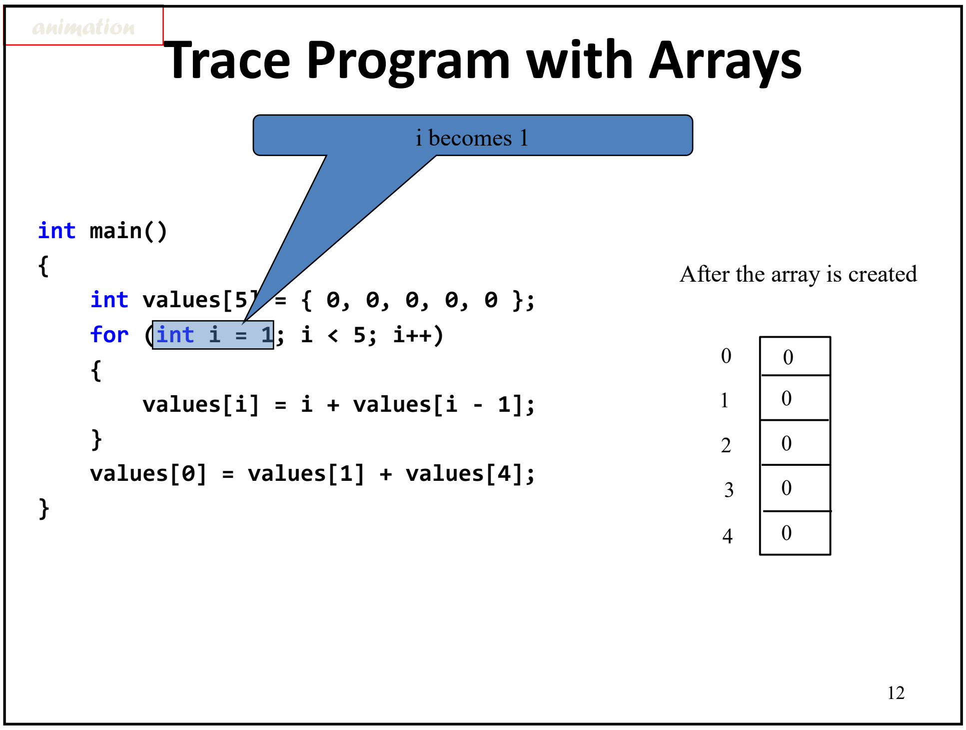
Trace Program with Arrays

Declare array variable values, create an array, and assign its reference to values

```
int main()
{
    int values[5] = { 0, 0, 0, 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created

0	0
1	0
2	0
3	0
4	0



animation

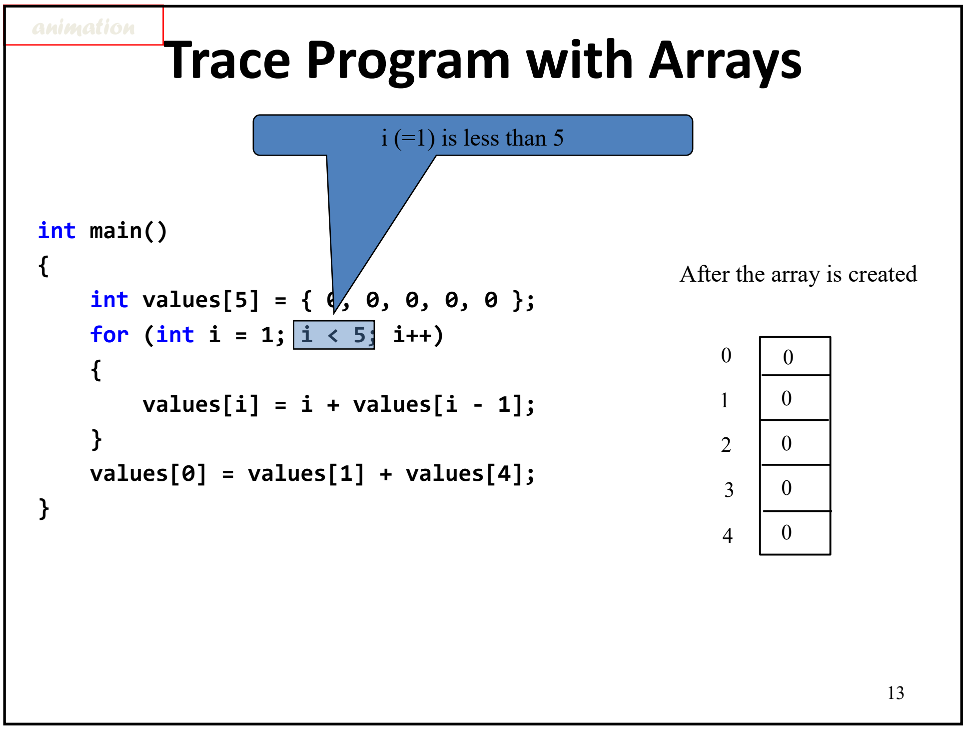
Trace Program with Arrays

i becomes 1

```
int main()
{
    int values[5] = { 0, 0, 0, 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created

0	0
1	0
2	0
3	0
4	0



animation

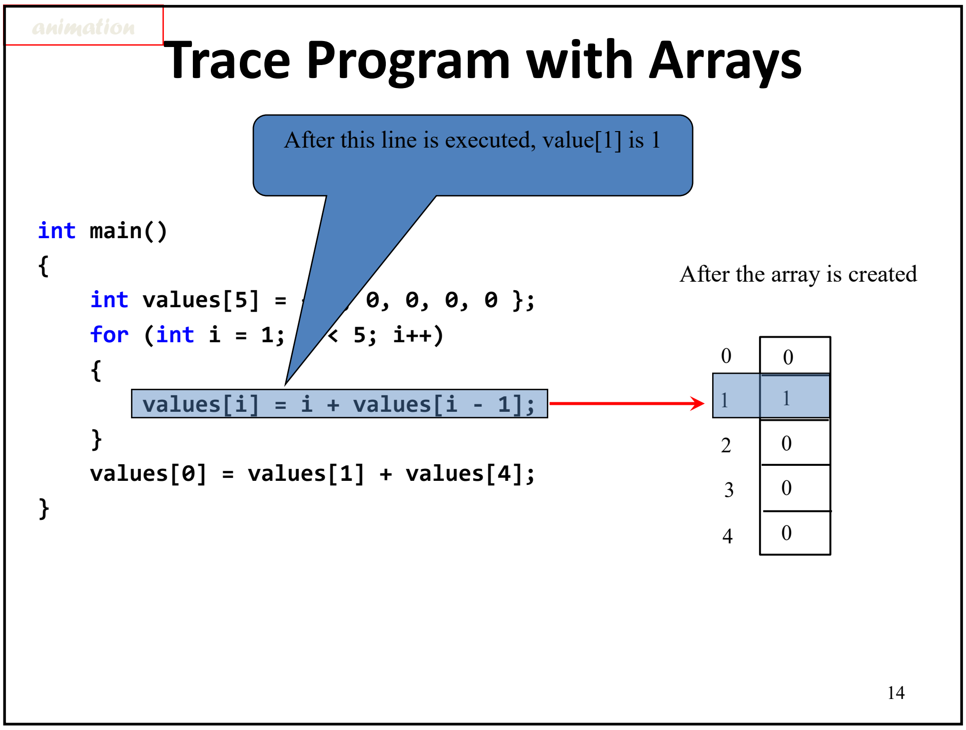
Trace Program with Arrays

i (=1) is less than 5

```
int main()
{
    int values[5] = { 0, 0, 0, 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created

0	0
1	0
2	0
3	0
4	0



animation

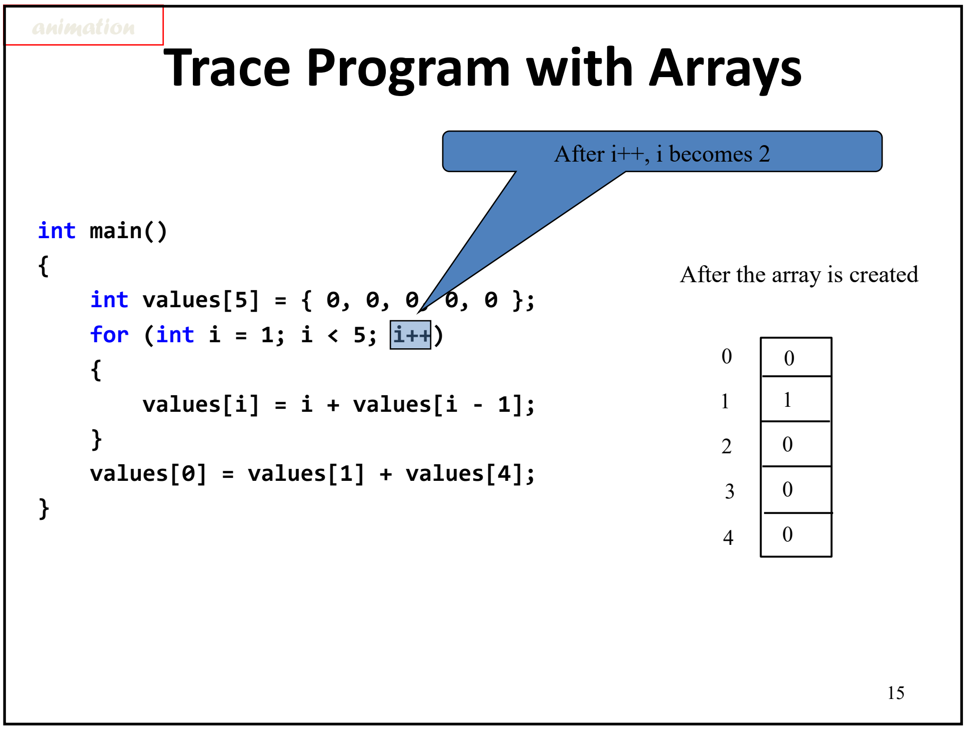
Trace Program with Arrays

After this line is executed, value[1] is 1

```
int main()
{
    int values[5] =      0, 0, 0, 0 };
    for (int i = 1;   < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created

0	0
1	1
2	0
3	0
4	0



animation

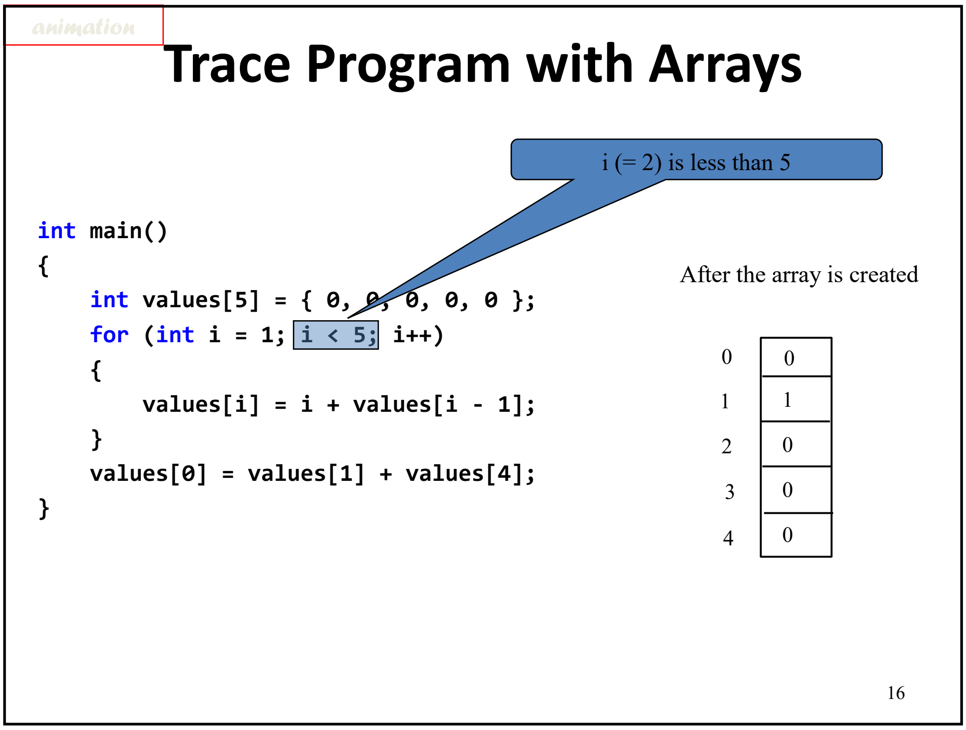
Trace Program with Arrays

After i++, i becomes 2

```
int main()
{
    int values[5] = { 0, 0, 0, 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created

0	0
1	1
2	0
3	0
4	0



animation

Trace Program with Arrays

i (= 2) is less than 5

```
int main()
{
    int values[5] = { 0, 0, 0, 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created

0	0
1	1
2	0
3	0
4	0

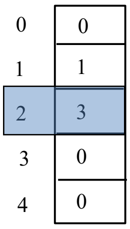
animation

Trace Program with Arrays

After this line is executed, values[2] is 3 (2 + 1)

```
int main()
{
    int values[5] =   , 0, 0, 0 };
    for (int i = 1;   < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created



---

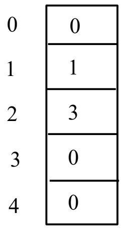
animation

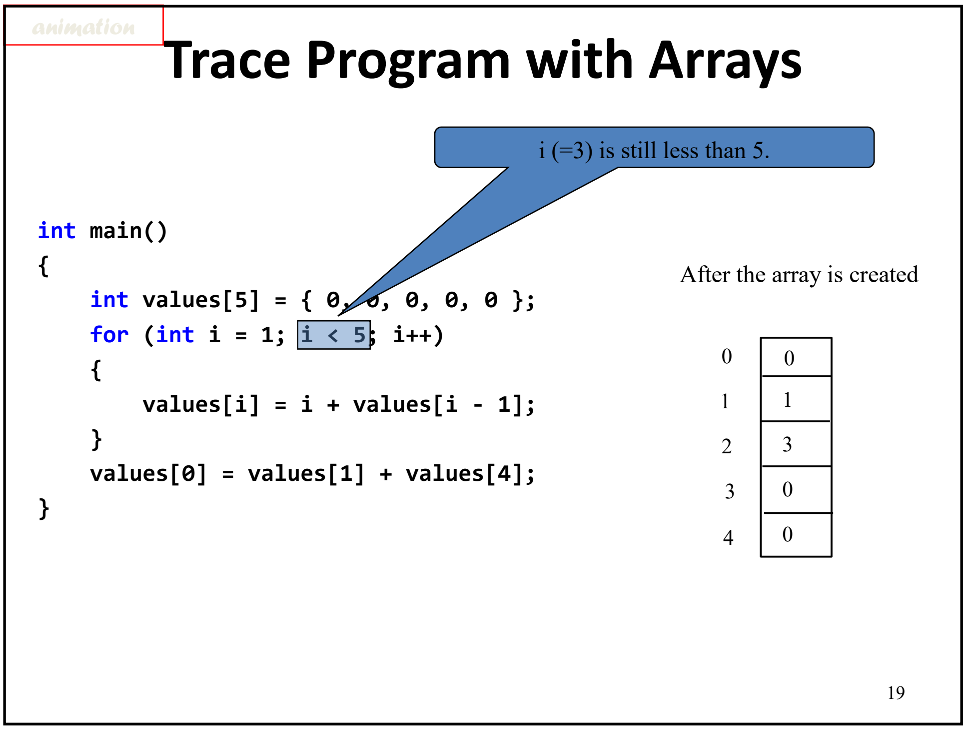
Trace Program with Arrays

After this, i becomes 3.

```
int main()
{
    int values[5] = { 0, 0,  , 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created





animation

Trace Program with Arrays

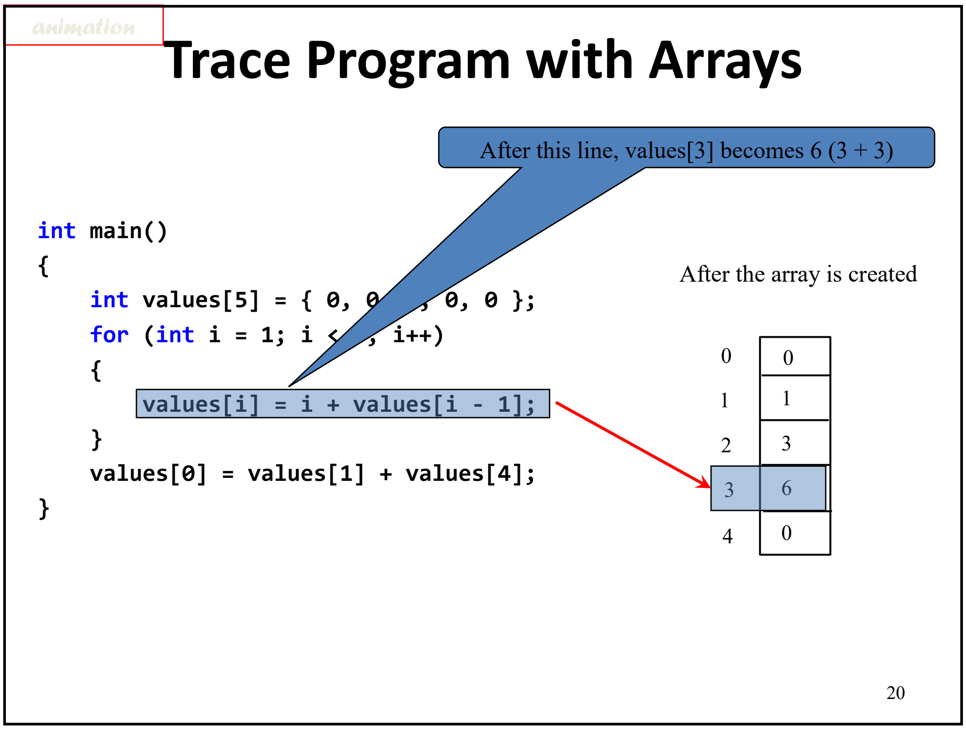
i (=3) is still less than 5.

```
int main()
{
    int values[5] = { 0, 0, 0, 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created

0	0
1	1
2	3
3	0
4	0

19



animation

Trace Program with Arrays

After this line, values[3] becomes 6 (3 + 3)

```
int main()
{
    int values[5] = { 0, 0, 0, 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created

0	0
1	1
2	3
3	6
4	0

20

animation

Trace Program with Arrays

After this, i becomes 4

```
int main()
{
    int values[5] = { 0, 0, 0, 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created

0	0
1	1
2	3
3	6
4	0

---

animation

Trace Program with Arrays

i (=4) is still less than 5

```
int main()
{
    int values[5] = { 0, 0, 0, 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created

0	0
1	1
2	3
3	6
4	0

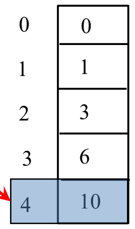
animation

Trace Program with Arrays

After this, values[4] becomes 10 (4 + 6)

```
int main()
{
    int values[5] =    , 0, 0, 0, 0 };
    for (int i = 1;    < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created



0	0
1	1
2	3
3	6
4	10

---

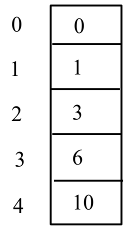
animation

Trace Program with Arrays

After i++, i becomes 5

```
int main()
{
    int values[5] = { 0, 0, 0, 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created



0	0
1	1
2	3
3	6
4	10

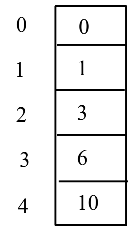
animation

Trace Program with Arrays

i (=5) < 5 is false. Exit the loop

```
int main()
{
    int values[5] = { 0, 0, 0, 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created



0	0
1	1
2	3
3	6
4	10

---

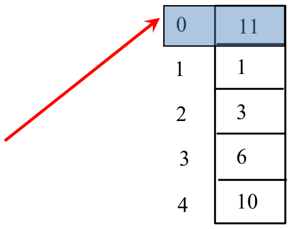
animation

Trace Program with Arrays

After this line, values[0] is 11 (1 + 10)

```
int main()
{
    int values[5] = { 0, 0, 0, 0, 0 };
    for (int i = 1; i < 5; i++)
    {
        values[i] = i + values[i - 1];
    }
    values[0] = values[1] + values[4];
}
```

After the array is created



0	11
1	1
2	3
3	6
4	10

Processing Arrays

- The following loop initializes the array `myList` with random values between `0` and `99`:

```
const int ARRAY_SIZE = 10;
double myList[ARRAY_SIZE];
for (int i = 0; i < ARRAY_SIZE; i++)
{
    myList[i] = rand() % 100;
}
```

- Summing all elements:

```
double total = 0;
for (int i = 0; i < ARRAY_SIZE; i++)
{
    total += myList[i];
}
```

Printing Arrays

To print an array, you have to print each element in the array using a loop like the following:

```
for (int i = 0; i < ARRAY_SIZE; i++)
{
    cout << myList[i] << " ";
}
```

Copying Arrays

Can you copy array using a syntax like this?

```
list = myList;  // Does not work
```

This is not allowed in C++. You have to copy individual elements from one array to the other as follows:

```
for (int i = 0; i < ARRAY_SIZE; i++)
{
    list[i] = myList[i];
}
```

Finding the Largest Element

- Use a variable named **max** to store the largest element. Initially **max** is **myList[0]**.
- To find the largest element in the array **myList**, compare each element in **myList** with **max**, update **max** if the element is greater than **max**.

```
double max = myList[0];
for (int i = 1; i < ARRAY_SIZE; i++)
{
    if (myList[i] > max)
        max = myList[i];
}
```

Finding the Smallest Index of the Largest Element

```
double max = myList[0];
int indexOfMax = 0;
for (int i = 1; i < ARRAY_SIZE; i++)
{
    if (myList[i] > max)
    {
        max = myList[i];
        indexOfMax = i;
    }
}
```

Shifting/Rotating Elements

```
double temp = myList[0]; // Save the first
// Shift elements up
for (int i = 1; i < ARRAY_SIZE; i++)
{
    myList[i - 1] = myList[i];
}
// First element to last position
myList[ARRAY_SIZE - 1] = temp;
```

C++11: Foreach loops are defined in C++11

Foreach Loops

```
double myList[] = { 0, 1.5, 2.1 };
for (double e : myList) {
    cout << e << endl;
}
```

```
0
1.5
2.1
```

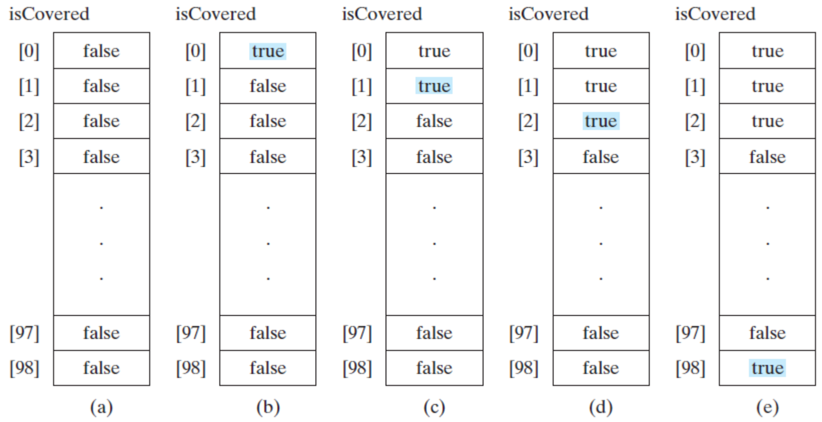
Outline

- Introduction
- Array Basics
- Problem: Lotto Numbers
- Problem: Deck of Cards
- Passing Arrays to Functions
- Preventing Changes of Array Arguments in Functions
- Returning Arrays from Functions
- C-Strings

supplement

Problem: Lotto Numbers

The problem is to write a program that checks if all the input numbers cover 1 to 99



isCovered	(a)	(b)	(c)	(d)	(e)
[0]	false	true	true	true	true
[1]	false	false	true	true	true
[2]	false	false	false	true	true
[3]	false	false	false	false	false
.	.	.	.	.	.
[97]	false	false	false	false	false
[98]	false	false	false	false	true

FIGURE 7.2 If number `i` appears in a lotto ticket, `isCovered[i-1]` is set to `true`.

LottoNumbers Run

LottoNumbers.cpp 1/2

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

int main()
{
    bool isCovered[99];
    int number; // number read from a file

    // Initialize the array
    for (int i = 0; i < 99; i++)
        isCovered[i] = false;

    // Read each number and mark its corresponding element
    cin >> number;
    while (number != 0)
    {
        isCovered[number - 1] = true;
        cin >> number;
    }
```

LottoNumbers.cpp 2/2

```cpp
    // Check if all covered
    bool allCovered = true; // Assume all covered initially
    for (int i = 0; i < 99; i++)
        if (!isCovered[i])
        {
            allCovered = false; // Find one number not covered
            break;
        }

    // Display result
    if (allCovered)
        cout << "The tickets cover all numbers" << endl;
    else
        cout << "The tickets don't cover all numbers" << endl;

    return 0;
}
```

Outline

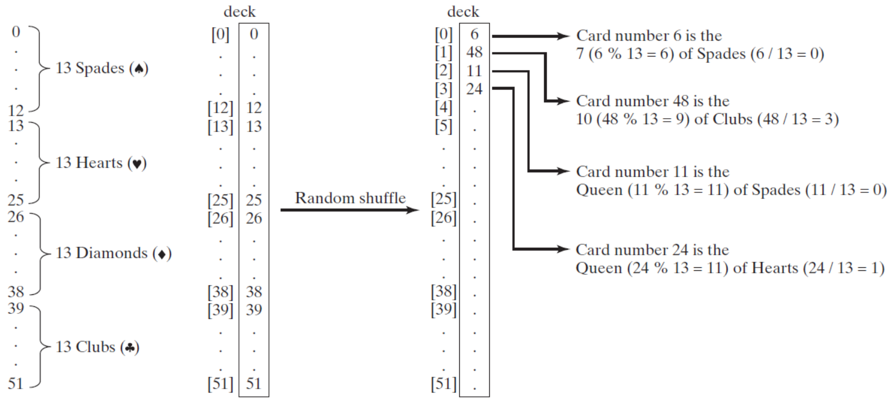
- Introduction
- Array Basics
- Problem: Lotto Numbers
- Problem: Deck of Cards
- Passing Arrays to Functions
- Preventing Changes of Array Arguments in Functions
- Returning Arrays from Functions
- C-Strings

Problem: Deck of Cards

- The problem is to write a program that picks four cards randomly from a deck of **52** cards.
- All the cards can be represented using an array named **deck**, filled with initial values **0** to **52**, as follows:

```
const int NUMBER_OF_CARDS = 52;
int deck[NUMBER_OF_CARDS];

// Initialize cards
for (int i = 0; i < NUMBER_OF_CARDS; i++)
    deck[i] = i;
```

Problem: Deck of Cards, cont.



Card number 6 is the 7 (6 % 13 = 6) of Spades (6 / 13 = 0)

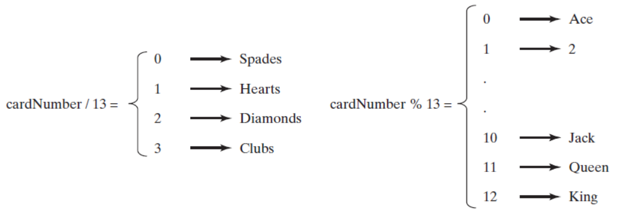
Card number 48 is the 10 (48 % 13 = 9) of Clubs (48 / 13 = 3)

Card number 11 is the Queen (11 % 13 = 11) of Spades (11 / 13 = 0)

Card number 24 is the Queen (24 % 13 = 11) of Hearts (24 / 13 = 1)

DeckOfCards Run

DeckOfCards.cpp 1/2



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

int main()
{
    const int NUMBER_OF_CARDS = 52;
    int deck[NUMBER_OF_CARDS];
    string suits[] = { "Spades", "Hearts", "Diamonds", "Clubs" };
    string ranks[] = { "Ace", "2", "3", "4", "5", "6", "7", "8", "9",
      "10", "Jack", "Queen", "King" };

    // Initialize cards
    for (int i = 0; i < NUMBER_OF_CARDS; i++)
        deck[i] = i;
```

DeckOfCards.cpp 2/2

```
    // Shuffle the cards
    srand(time(0));
    for (int i = 0; i < NUMBER_OF_CARDS; i++)
    {
        // Generate an index randomly
        int index = rand() % NUMBER_OF_CARDS;
        int temp = deck[i];
        deck[i] = deck[index];
        deck[index] = temp;
    }

    // Display the first four cards
    for (int i = 0; i < 4; i++)
    {
        string suit = suits[deck[i] / 13];
        string rank = ranks[deck[i] % 13];
        cout << "Card number " << deck[i] << ": "
            << rank << " of " << suit << endl;
    }

    return 0;
}
```

Outline

- Introduction
- Array Basics
- Problem: Lotto Numbers
- Problem: Deck of Cards
- Passing Arrays to Functions
- Preventing Changes of Array Arguments in Functions
- Returning Arrays from Functions
- C-Strings

Passing Arrays to Functions

- You can pass an entire array to a function.
- You need also to pass the size of the array.
- This program gives an example to demonstrate how to declare and invoke this type of functions.

PassArrayDemo Run

PassArrayDemo.cpp

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

void printArray(int list[], int arraySize); // Prototype

int main()
{
    int numbers[6] = { 1, 4, 3, 6, 8, 9 };
    printArray(numbers, 6); // Invoke the function

    return 0;
}

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

```
1 4 3 6 8 9
```

Pass-by-Value

- Passing an array variable means that the starting address of the array is passed to the formal parameter by value.
- The parameter inside the function references to the same array that is passed to the function. No new arrays are created.

EffectOfPassArrayDemo Run

EffectOfPassArrayDemo.cpp

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

void m(int, int[]);

int main()
{
    int x = 1; // x represents an int value
    int y[10] = { 0 }; // y represents an array of int values

    m(x, y); // Invoke m with arguments x and y

    cout << "x is " << x << endl;
    cout << "y[0] is " << y[0] << endl;

    return 0;
}

void m(int number, int numbers[])
{
    number = 1001; // Assign a new value to number
    numbers[0] = 5555; // Assign a new value to numbers[0]
}
```

```
x is 1
y[0] is 5555
```

Outline

- Introduction
- Array Basics
- Problem: Lotto Numbers
- Problem: Deck of Cards
- Passing Arrays to Functions
- Preventing Changes of Array Arguments in Functions
- Returning Arrays from Functions
- C-Strings

Preventing Changes of Array Arguments in Functions

- Passing arrays by reference makes sense for performance reasons. If an array is passed by value, all its elements must be copied into a new array.
- However, passing arrays by its reference value could lead to errors if your function changes the array accidentally.
- To prevent it from happening, you can put the **`const`** to tell the compiler that the array cannot be changed.
- The compiler will report errors if the code in the function attempts to modify the array.

ConstArrayDemo   Compile error

ConstArrayDemo.cpp

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

void p(int const list[], int arraySize)
{
  // Modify array accidentally
  list[0] = 100; // Compile error!
}

int main()
{
  int numbers[5] = {1, 4, 3, 6, 8};
  p(numbers, 5);

  return 0;
}
```

```
1>C:\ConstArrayDemo.cpp(7,18): error C3892: 'list': you cannot assign to a
variable that is const
1>Done building project "Testing.vcxproj" -- FAILED.
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Outline

- Introduction
- Array Basics
- Problem: Lotto Numbers
- Problem: Deck of Cards
- Passing Arrays to Functions
- Preventing Changes of Array Arguments in Functions
- Returning Arrays from Functions
- C-Strings

Returning Arrays from Functions

- How to return an array from a function?
- You may attempt to declare a function that returns a new array that is a reversal of an array as follows:

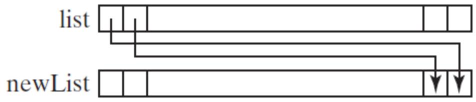
```
// Return the reversal of list
int[] reverse(const int list[], int size);
```

- This is not allowed in C++.

Returning Arrays from Functions, cont.

- However, you can pass two array arguments in the function, as follows:

```
// newList is the reversal of list
void reverse(const int list[], int newList[],
             int size);
```



ReverseArray Run

ReverseArray.cpp 1/2

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

// newList is the reversal of list
void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}

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

ReverseArray.cpp 1/2

```
int main()
{
    const int SIZE = 6;
    int list[] = { 1, 2, 3, 4, 5, 6 };
    int newList[SIZE];

    reverse(list, newList, SIZE);

    cout << "The original array: ";
    printArray(list, SIZE);
    cout << endl;

    cout << "The reversed array: ";
    printArray(newList, SIZE);
    cout << endl;

    return 0;
}
```

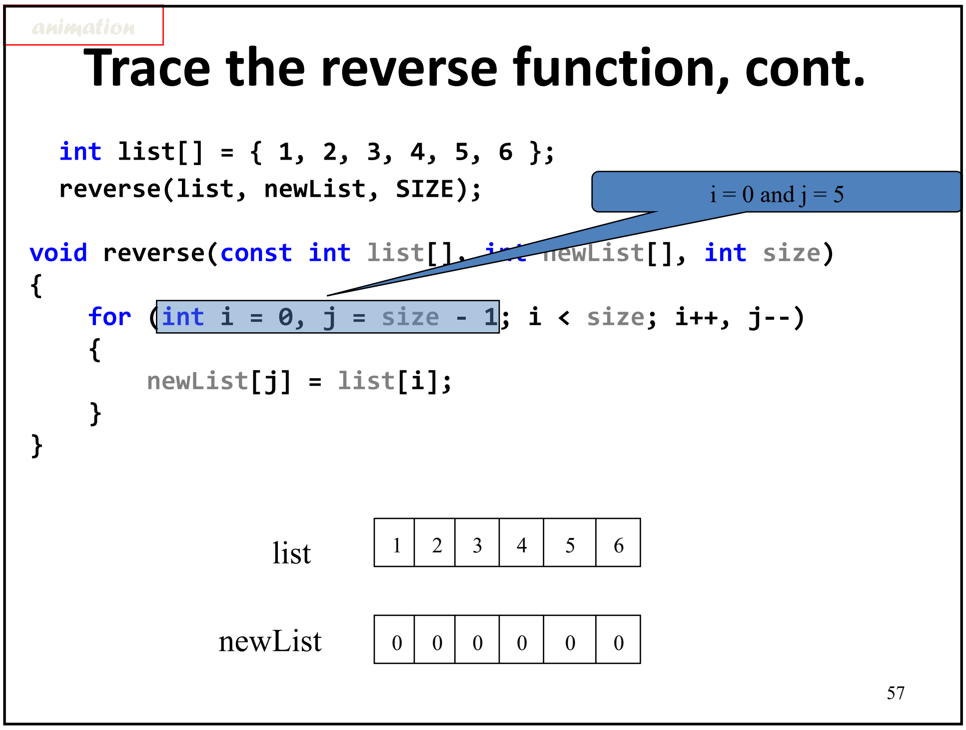
---

animation

Trace the reverse Function

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

list	1	2	3	4	5	6
newList	0	0	0	0	0	0



animation

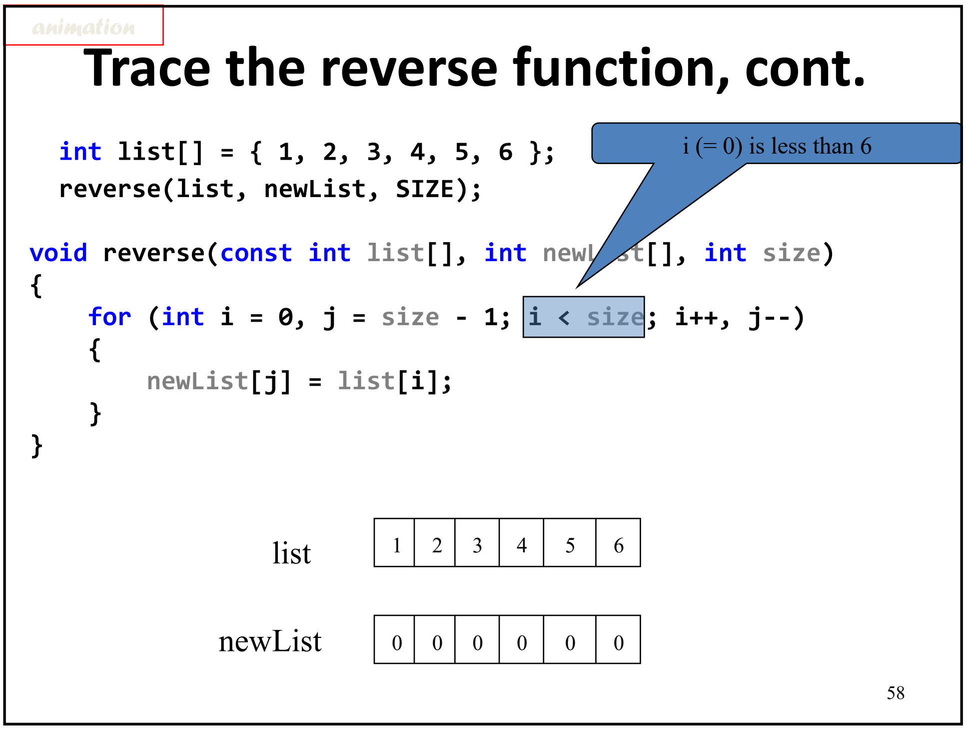
Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i = 0 and j = 5

list					
1	2	3	4	5	6

newList					
0	0	0	0	0	0



animation

Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i (= 0) is less than 6

list					
1	2	3	4	5	6

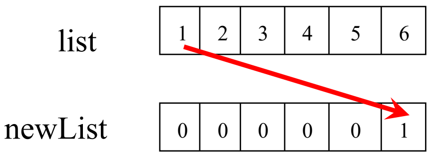
newList					
0	0	0	0	0	0

animation

Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i = 0 and j = 5
Assign list[0] to result[5]



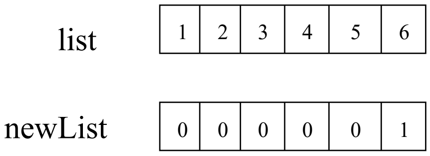
---

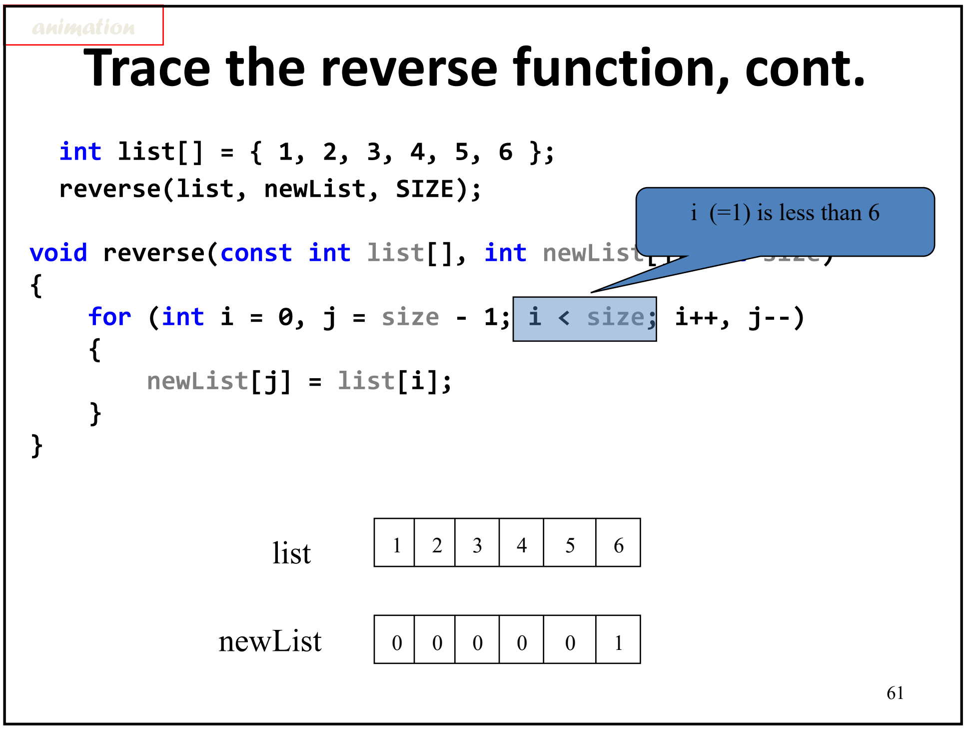
animation

Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

After this, i becomes 1 and j becomes 4





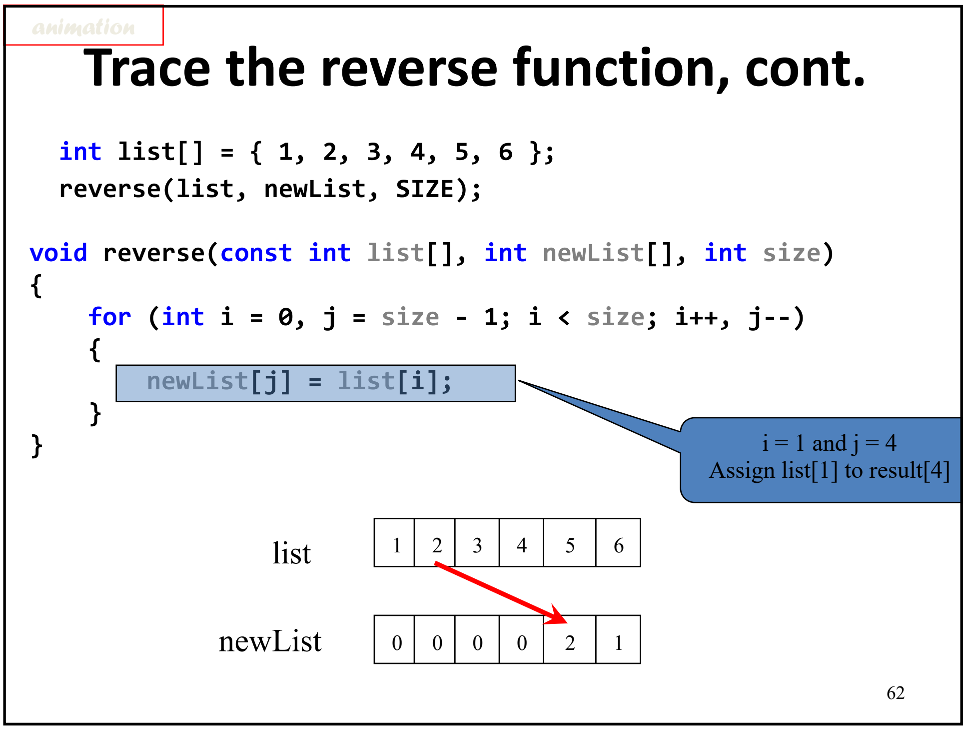
animation

Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i (=1) is less than 6

list	1	2	3	4	5	6
newList	0	0	0	0	0	1



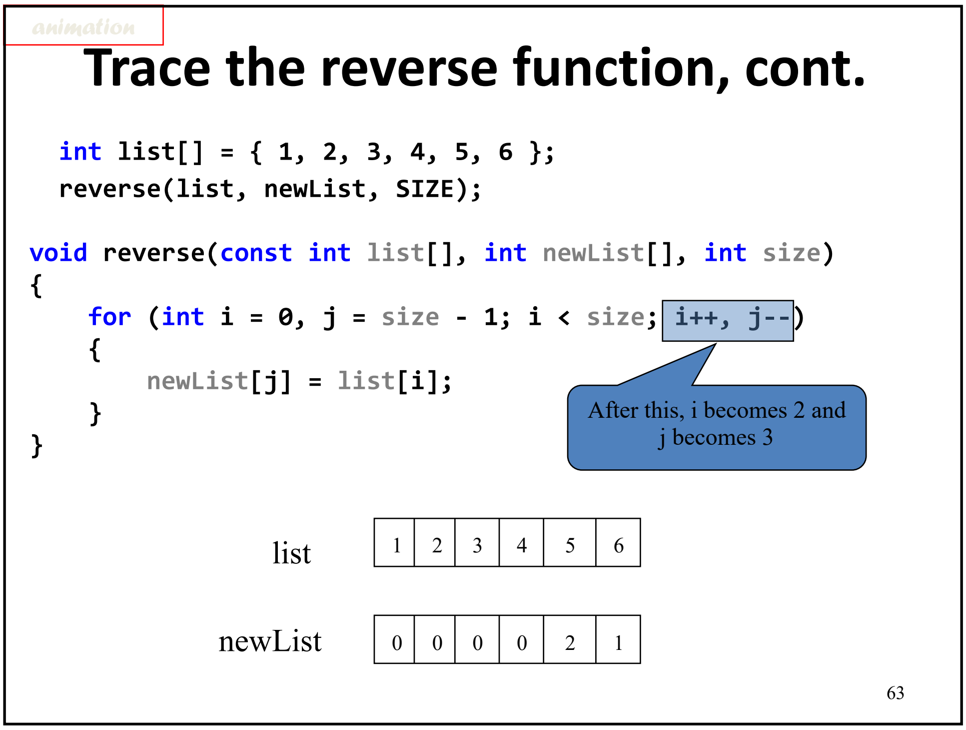
animation

Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i = 1 and j = 4
Assign list[1] to result[4]

list	1	2	3	4	5	6
newList	0	0	0	0	2	1



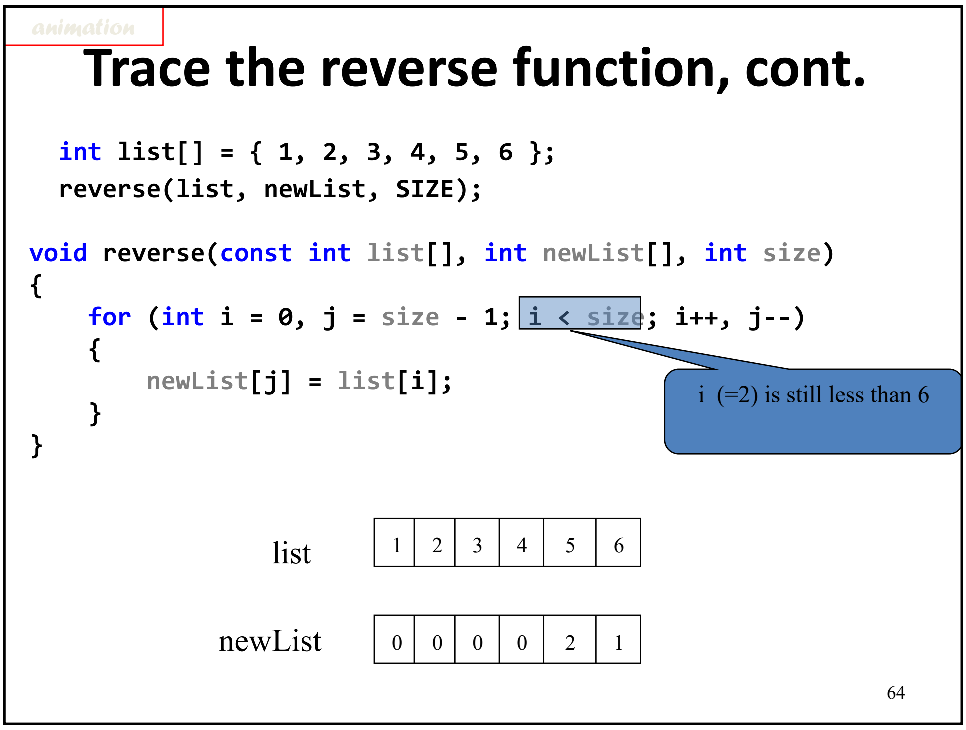
animation

Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

After this, i becomes 2 and j becomes 3

list	1	2	3	4	5	6
newList	0	0	0	0	2	1



animation

Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i (=2) is still less than 6

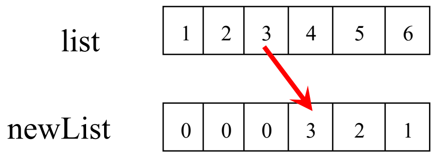
list	1	2	3	4	5	6
newList	0	0	0	0	2	1

animation

Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i = 2 and j = 3
Assign list[i] to result[j]



list: 1 2 3 4 5 6

newList: 0 0 0 3 2 1

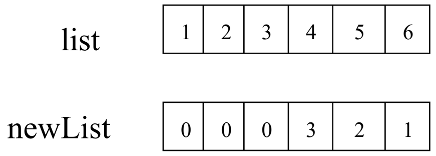
---

animation

Trace the reverse function, cont.

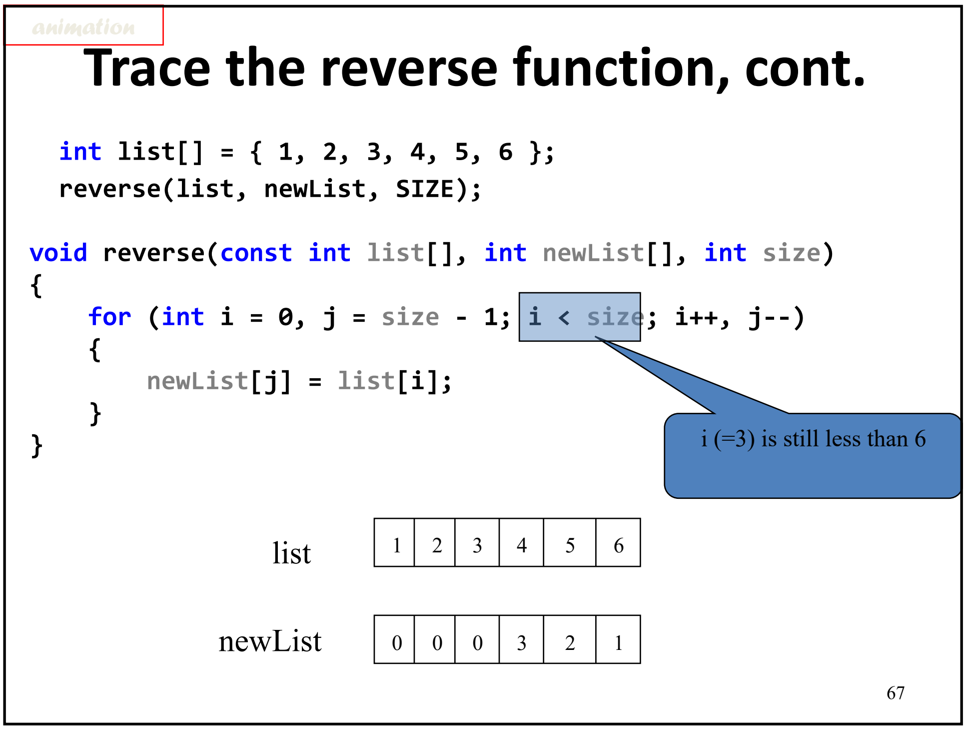
```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

After this, i becomes 3 and j becomes 2



list: 1 2 3 4 5 6

newList: 0 0 0 3 2 1



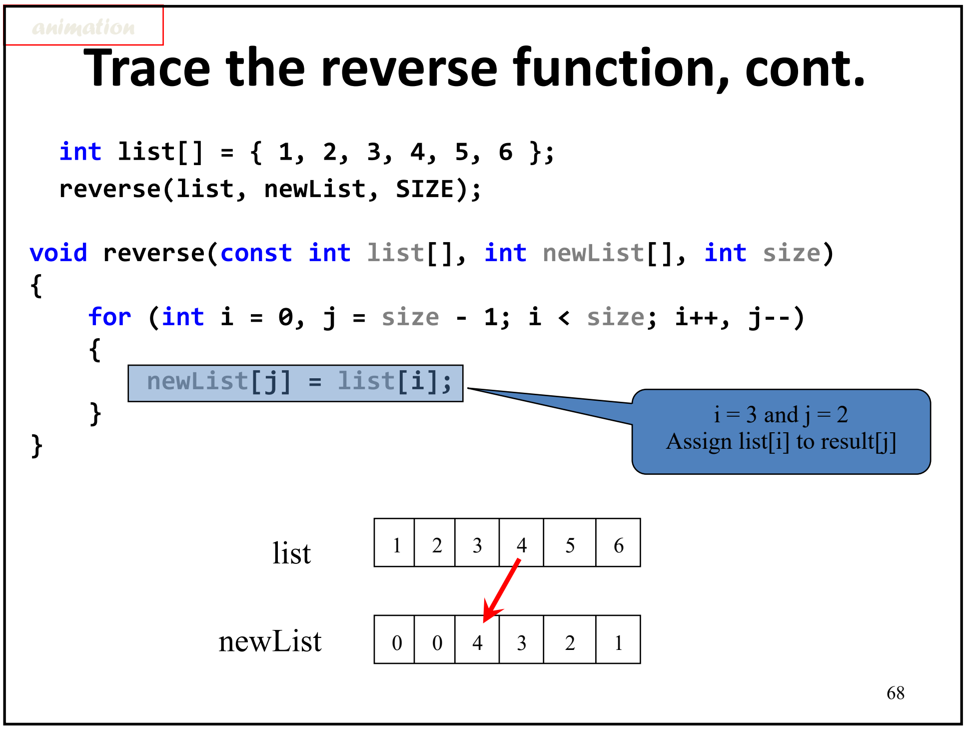
animation

Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i (=3) is still less than 6

list	1	2	3	4	5	6
newList	0	0	0	3	2	1



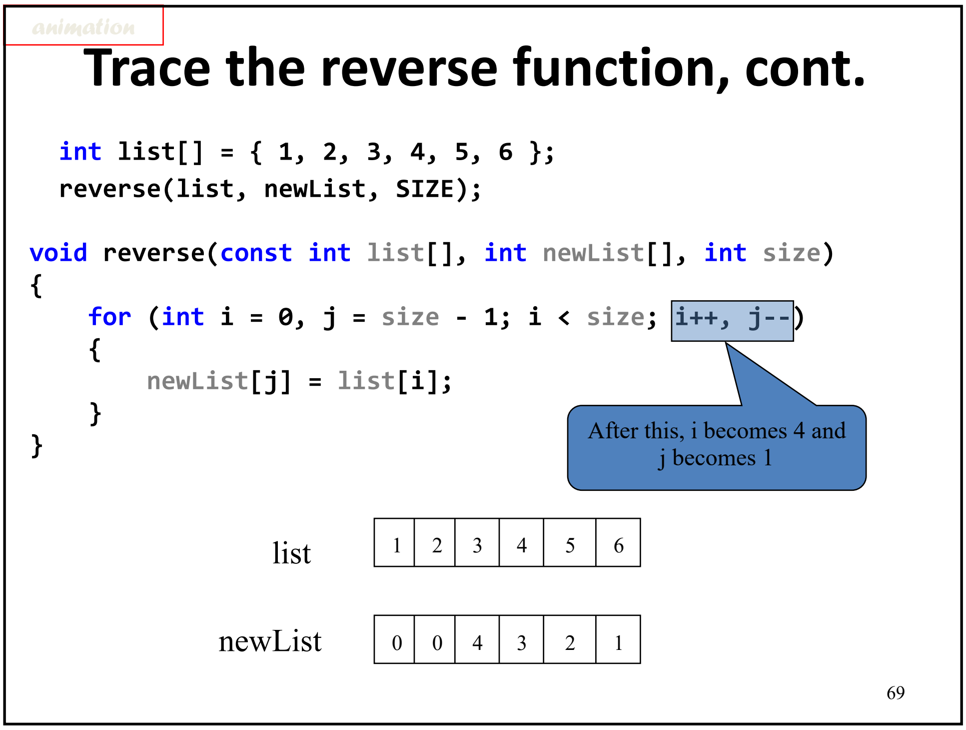
animation

Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i = 3 and j = 2
Assign list[i] to result[j]

list	1	2	3	4	5	6
newList	0	0	4	3	2	1

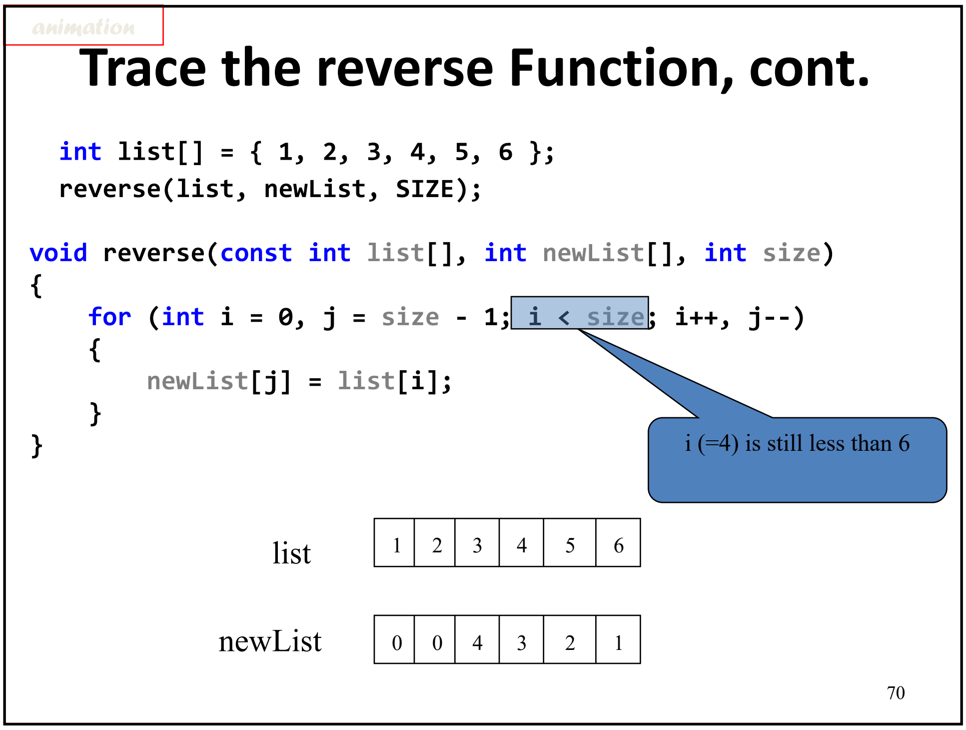


Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

After this, i becomes 4 and j becomes 1

list	1	2	3	4	5	6
newList	0	0	4	3	2	1

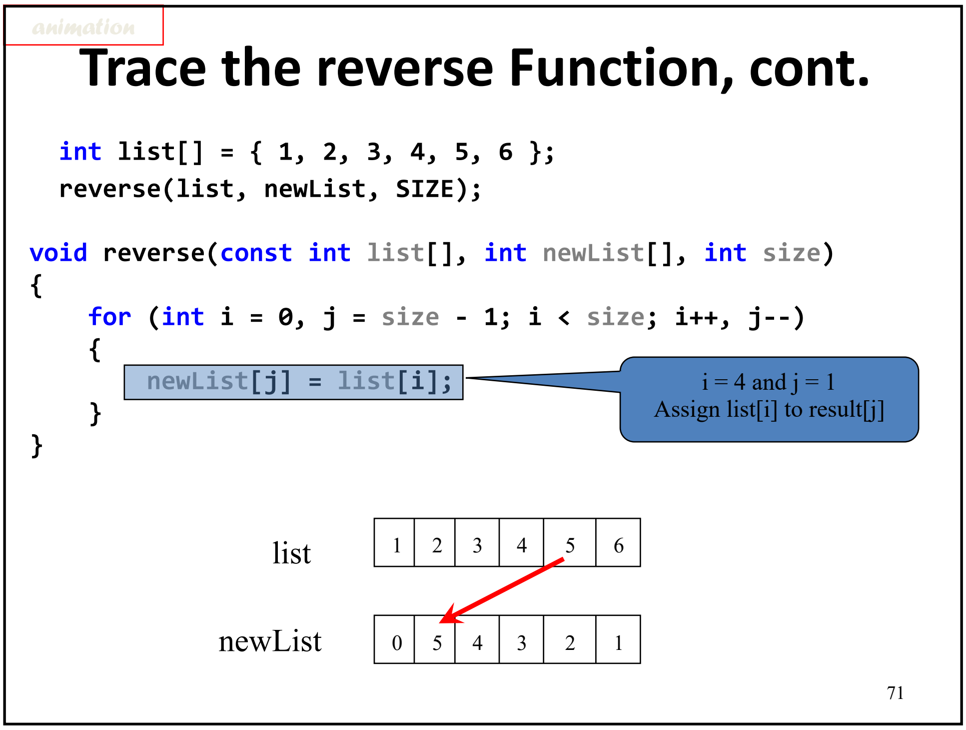


Trace the reverse Function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i (=4) is still less than 6

list	1	2	3	4	5	6
newList	0	0	4	3	2	1



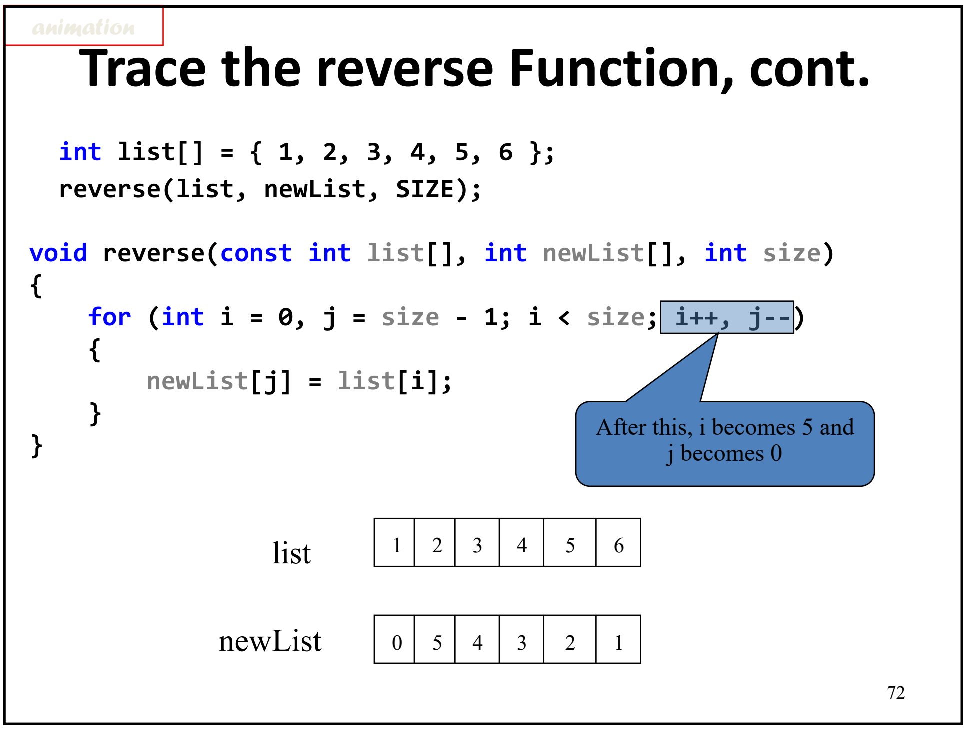
animation

Trace the reverse Function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i = 4 and j = 1
Assign list[i] to result[j]

list	1	2	3	4	5	6
newList	0	5	4	3	2	1



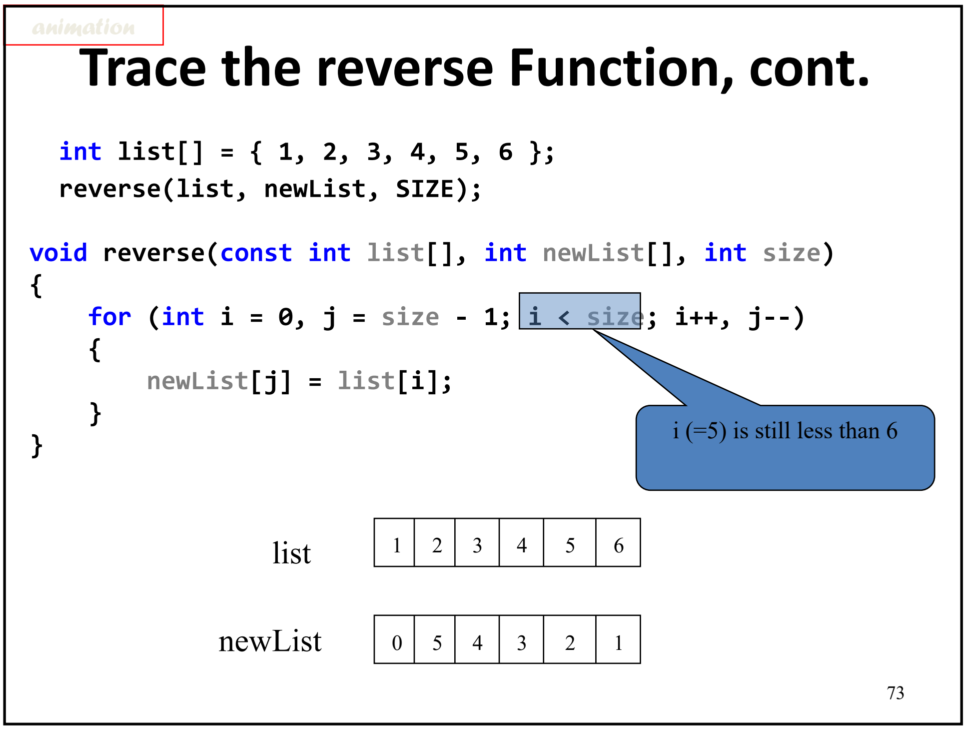
animation

Trace the reverse Function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

After this, i becomes 5 and j becomes 0

list	1	2	3	4	5	6
newList	0	5	4	3	2	1



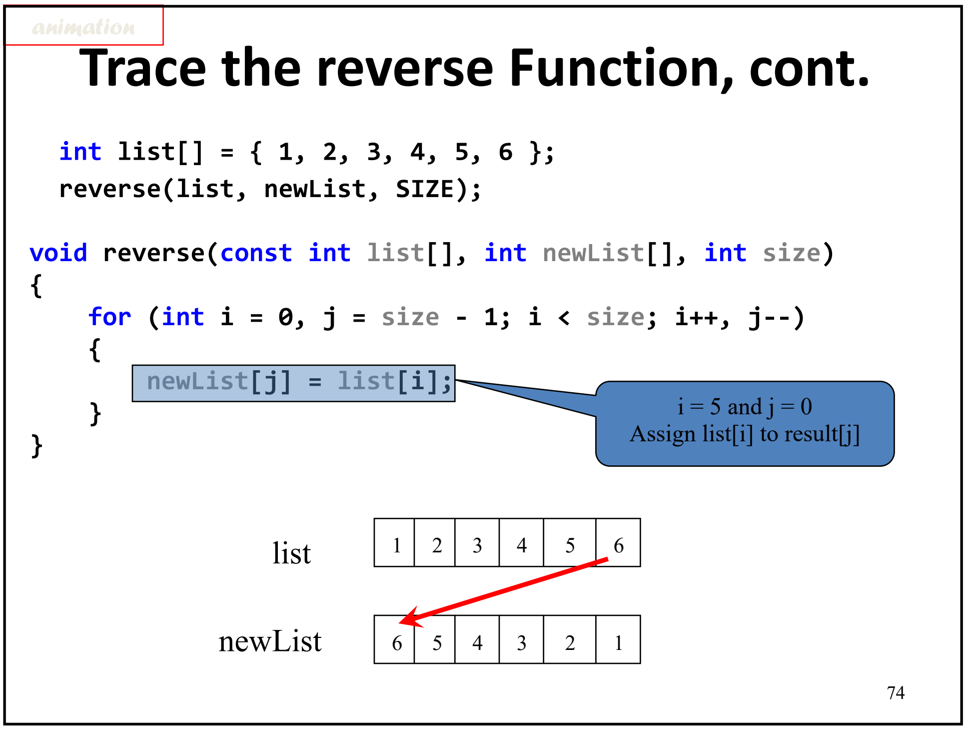
animation

Trace the reverse Function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i (=5) is still less than 6

list	1	2	3	4	5	6
newList	0	5	4	3	2	1



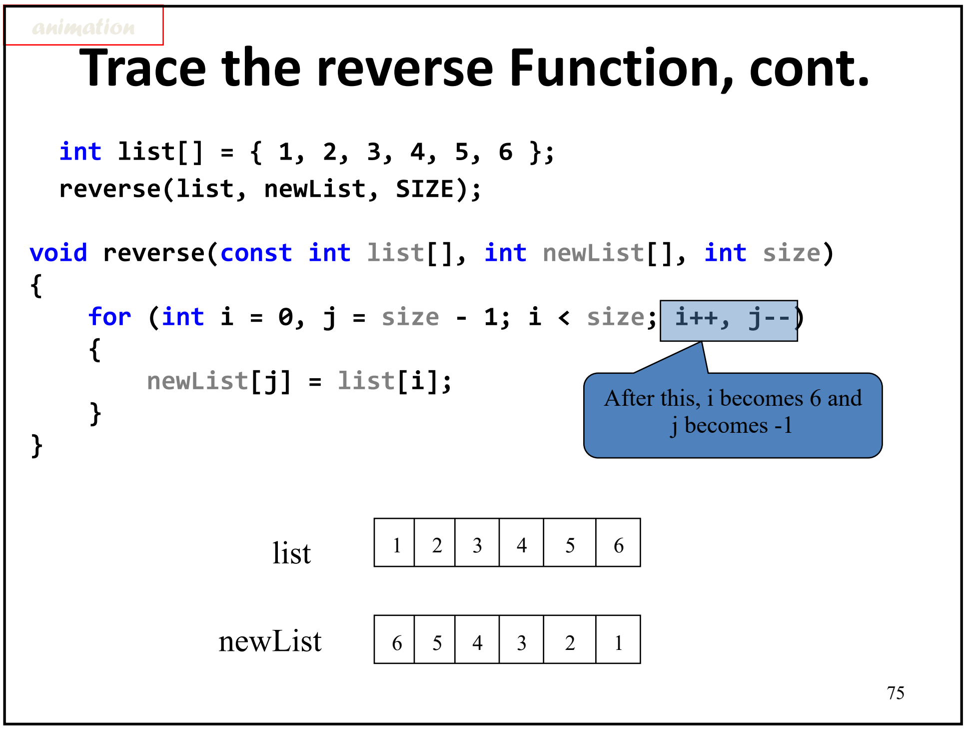
animation

Trace the reverse Function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i = 5 and j = 0
Assign list[i] to result[j]

list	1	2	3	4	5	6
newList	6	5	4	3	2	1



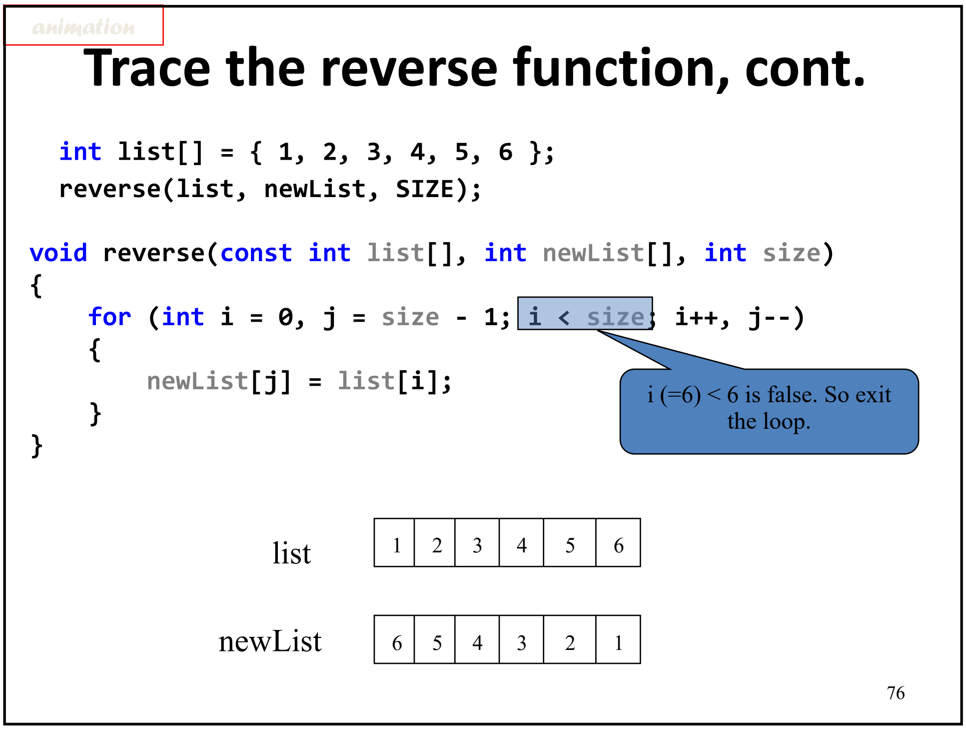
animation

Trace the reverse Function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

After this, i becomes 6 and j becomes -1

list	1	2	3	4	5	6
newList	6	5	4	3	2	1



animation

Trace the reverse function, cont.

```
  int list[] = { 1, 2, 3, 4, 5, 6 };
  reverse(list, newList, SIZE);

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i];
    }
}
```

i (=6) < 6 is false. So exit the loop.

list	1	2	3	4	5	6
newList	6	5	4	3	2	1

Outline

- Introduction
- Array Basics
- Problem: Lotto Numbers
- Problem: Deck of Cards
- Passing Arrays to Functions
- Preventing Changes of Array Arguments in Functions
- Returning Arrays from Functions
- C-Strings

C-Strings

- You studied the string type in Chapter 4.
- Example:

```
string s = "welcome to C++";
s.at(0) = 'W';
cout << s.length() << s[0] << endl;
```

```
14W
```

- Here we study the older C-strings because of their popularity.

Initializing Character Arrays

- You can define arrays of characters.

```
char city[] = { 'D', 'a', 'l', 'l', 'a', 's' };
```

- C-strings are defined as follows:

```
char city[] = "Dallas";
```

- In this case, C++ adds the character '\0', called the *null terminator*, to indicate the end of the string.

'D'	'a'	'l'	'l'	'a'	's'	'\0'
city[0]	city[1]	city[2]	city[3]	city[4]	city[5]	city[6]

Reading C-Strings

You can read a string from the keyboard using the `cin` object. For example, see the following code:

```
char city[10];
cout << "Enter a city: ";
cin >> city; // read to array city
cout << "You entered " << city << endl;
```

Printing Character Array

For a character array, it can be printed using one print statement. For example, the following code displays Dallas:

```
char city[] = "Dallas";
cout << city;
```

Reading C-Strings Using `getline`

- C++ provides the **`cin.getline`** function in the **`iostream`** header file, which reads a string into an array:

```
cin.getline(char array[], int size, char delimitChar);
```

- The function stops reading characters when the delimiter character is encountered or when the **`size - 1`** number of characters are read.
- The last character in the array is reserved for the null terminator (**`'\0'`**).
- If the delimiter is encountered, it is read, but not stored in the array.
- The third argument **`delimitChar`** has a default value (**`'\n'`**).

Working with C-Strings

- The following function finds the length of a C-string:

```
unsigned int strlen(char s[])
{
    for (int i = 0; s[i] != '\0'; i++)
       ;
    return i;
}
```

- The **cstring** and **cstdlib** headers provide many useful C-strings functions.

C-String Functions

Function	*Description*
`size_t strlen(char s[])`	Returns the length of the string, i.e., the number of the characters before the null terminator.
`strcpy(char s1[], const char s2[])`	Copies string s2 to string s1.
`strncpy(char s1[], const char s2[], size_t n)`	Copies the first n characters from string s2 to string s1.
`strcat(char s1[], const char s2[])`	Appends string s2 to s1.
`strncat(char s1[], const char s2[], size_t n)`	Appends the first n characters from string s2 to s1.
`int strcmp(char s1[], const char s2[])`	Returns a value greater than 0, 0, or less than 0 if s1 is greater than, equal to, or less than s2 based on the numeric code of the characters.
`int strncmp(char s1[], const char s2[], size_t n)`	Same as strcmp, but compares up to n number of characters in s1 with those in s2.
`int atoi(char s[])`	Returns an int value for the string.
`double atof(char s[])`	Returns a double value for the string.
`long atol(char s[])`	Returns a long value for the string.
`void itoa(int value, char s[], int radix)`	Obtains an integer value to a string based on specified radix.

C-String Examples

CopyString Run

CombineString Run

CompareString Run

StringConversion Run

CopyString.CPP

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

int main()
{
    char s1[20];
    char s2[20] = "Dallas, Texas";
    char s3[20] = "AAAAAAAAAA";

    strcpy(s1, s2);
    strncpy(s3, s2, 6);
    s3[6] = '\0'; // Insert null terminator

    cout << "The string in s1 is " << s1 << endl;
    cout << "The string in s2 is " << s2 << endl;
    cout << "The string in s3 is " << s3 << endl;
    cout << "The length of string s3 is " << strlen(s3) << endl;

    return 0;
}
```

```
The string in s1 is Dallas, Texas
The string in s2 is Dallas, Texas
The string in s3 is Dallas
The length of string s3 is 6
```

CombineString.cpp

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

int main()
{
    char s1[20] = "Dallas";
    char s2[20] = "Texas, USA";
    char s3[20] = "Dallas";

    strcat(strcat(s1, ", "), s2);
    strncat(strcat(s3, ", "), s2, 5);

    cout << "The string in s1 is " << s1 << endl;
    cout << "The string in s2 is " << s2 << endl;
    cout << "The string in s3 is " << s3 << endl;
    cout << "The length of string s1 is " << strlen(s1) << endl;
    cout << "The length of string s3 is " << strlen(s3) << endl;

    return 0;
}
```

```
The string in s1 is Dallas, Texas, USA
The string in s2 is Texas, USA
The string in s3 is Dallas, Texas
The length of string s1 is 18
The length of string s3 is 13
```

CompareString.cpp

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

int main()
{
    char s1[] = "abcdefg";
    char s2[] = "abcdg";
    char s3[] = "abcdg";

    cout << "strcmp(s1, s2) is " << strcmp(s1, s2) << endl;
    cout << "strcmp(s2, s1) is " << strcmp(s2, s1) << endl;
    cout << "strcmp(s2, s3) is " << strcmp(s2, s3) << endl;
    cout << "strncmp(s1, s2, 3) is " << strncmp(s1, s2, 3)
         << endl;

    return 0;
}
```

```
strcmp(s1, s2) is -1
strcmp(s2, s1) is 1
strcmp(s2, s3) is 0
strncmp(s1, s2, 3) is 0
```

StringConversion.cpp

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

int main()
{
    cout << atoi("4") + atoi("5") << endl;
    cout << atof("4.5") + atof("5.5") << endl;

    char s[10];
    itoa(42, s, 8);
    cout << s << endl;

    itoa(42, s, 10);
    cout << s << endl;

    itoa(42, s, 16);
    cout << s << endl;

    return 0;
}
```

```
9
10
52
42
2a
```

Converting Numbers to Strings

- Note that the **`to_string`** function is useful to convert numbers to string type.

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

int main()
{
    int x = 15;
    double y = 1.32;
    long long int z = 10935;
    string s = "Three numbers: " + to_string(x) + ", " +
        to_string(y) + ", and " + to_string(z);
    cout << s << endl;

    return 0;
}
```

C++11: the to_string function is defined in C++11

```
Three numbers: 15, 1.320000, and 10935
```

Outline

- Introduction
- Array Basics
- Problem: Lotto Numbers
- Problem: Deck of Cards
- Passing Arrays to Functions
- Preventing Changes of Array Arguments in Functions
- Returning Arrays from Functions
- C-Strings

Chapter 8: Multidimensional Arrays

Sections 8.1–8.5, 8.8

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)

Outline

- Introduction
- Declaring Two-Dimensional Arrays
- Processing Two-Dimensional Arrays
- Passing Two-Dimensional Arrays to Functions
- Problem: Grading a Multiple-Choice Test
- Multidimensional Arrays

Introduction

Data in a table or a matrix can be represented using a two-dimensional array.

Distance Table (in miles)

	Chicago	*Boston*	*New York*	*Atlanta*	*Miami*	*Dallas*	*Houston*
Chicago	0	983	787	714	1375	967	1087
Boston	983	0	214	1102	1763	1723	1842
New York	787	214	0	888	1549	1548	1627
Atlanta	714	1102	888	0	661	781	810
Miami	1375	1763	1549	661	0	1426	1187
Dallas	967	1723	1548	781	1426	0	239
Houston	1087	1842	1627	810	1187	239	0

Outline

- Introduction
- Declaring Two-Dimensional Arrays
- Processing Two-Dimensional Arrays
- Passing Two-Dimensional Arrays to Functions
- Problem: Grading a Multiple-Choice Test
- Multidimensional Arrays

Declaring Two-Dimensional Arrays

```
elementType arrayName[ROW_SIZE][COLUMN_SIZE];
```

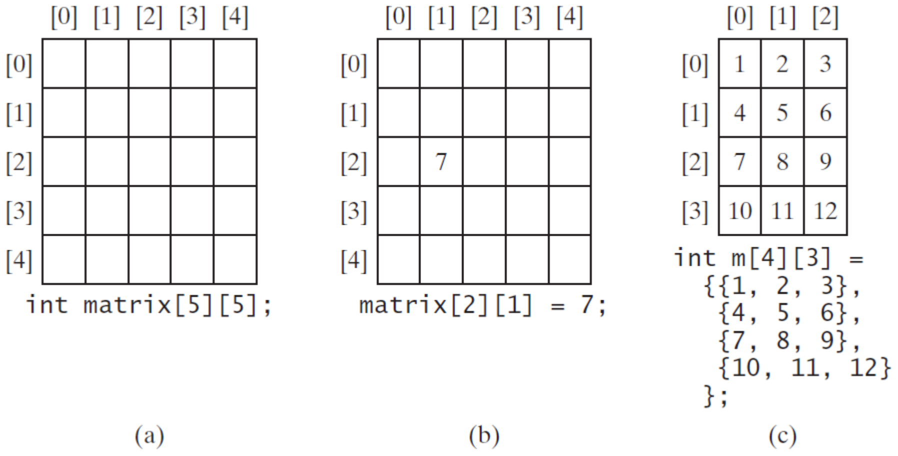
- Example

  `int distances[7][7];`

- An element in a two-dimensional array is accessed through a row and column index.

  `int bostonToDalas = distances[1][5];`

Two-Dimensional Array Illustration



```
int matrix[5][5];
```

(a)

```
matrix[2][1] = 7;
```

(b)

```
int m[4][3] =
  {{1, 2, 3},
   {4, 5, 6},
   {7, 8, 9},
   {10, 11, 12}
  };
```

(c)

Outline

- Introduction
- Declaring Two-Dimensional Arrays
- Processing Two-Dimensional Arrays
- Passing Two-Dimensional Arrays to Functions
- Problem: Grading a Multiple-Choice Test
- Multidimensional Arrays

Initializing Arrays with Random Values

- Nested `for` loops are often used to process a two-dimensional array.
- The following loop initializes the array with random values between `0` and `99`:

```
for (int row = 0; row < rowSize; row++)
  {
    for (int column = 0; column < columnSize; column++)
    {
      matrix[row][column] = rand() % 100;
    }
  }
```

Printing Arrays

- To print a two-dimensional array, you have to print each element in the array using a loop like the following:

```
for (int row = 0; row < rowSize; row++)
{
  for (int column = 0; column < columnSize; column++)
    {
       cout << matrix[row][column] << " ";
    }
    cout << endl;
}
```

Summing All Elements

- To sum all elements of a two-dimensional array:

```
int total = 0;
for (int row = 0; row < ROW_SIZE; row++)
{
  for (int column = 0; column < COLUMN_SIZE; column++)
  {
    total += matrix[row][column];
  }
}
```

Summing Elements by Column

- For each column, use a variable named **`total`** to store its sum. Add each element in the column to **`total`** using a loop like this:

```
for (int column = 0; column < columnSize; column++)
{
  int total = 0;
  for (int row = 0; row < rowSize; row++)
    total += matrix[row][column];
  cout << "Sum for column " << column << " is “
       << total << endl;
}
```

Which row has the largest sum?

- Use variables **`maxRow`** and **`indexOfMaxRow`** to track the largest sum and index of the row. For each row, compute its sum and update **`maxRow`** and **`indexOfMaxRow`** if the new sum is greater.

```
int maxRow = 0;
int indexOfMaxRow = 0;
// Get sum of the first row in maxRow
for (int column = 0; column < COLUMN_SIZE; column++)
    maxRow += matrix[0][column];
for (int row = 1; row < ROW_SIZE; row++)
{
    int totalOfThisRow = 0;
    for (int column = 0; column < COLUMN_SIZE; column++)
        totalOfThisRow += matrix[row][column];
    if (totalOfThisRow > maxRow)
    {
        maxRow = totalOfThisRow;
        indexOfMaxRow = row;
    }
}
cout << "Row " << indexOfMaxRow
    << " has the maximum sum of " << maxRow << endl;
```

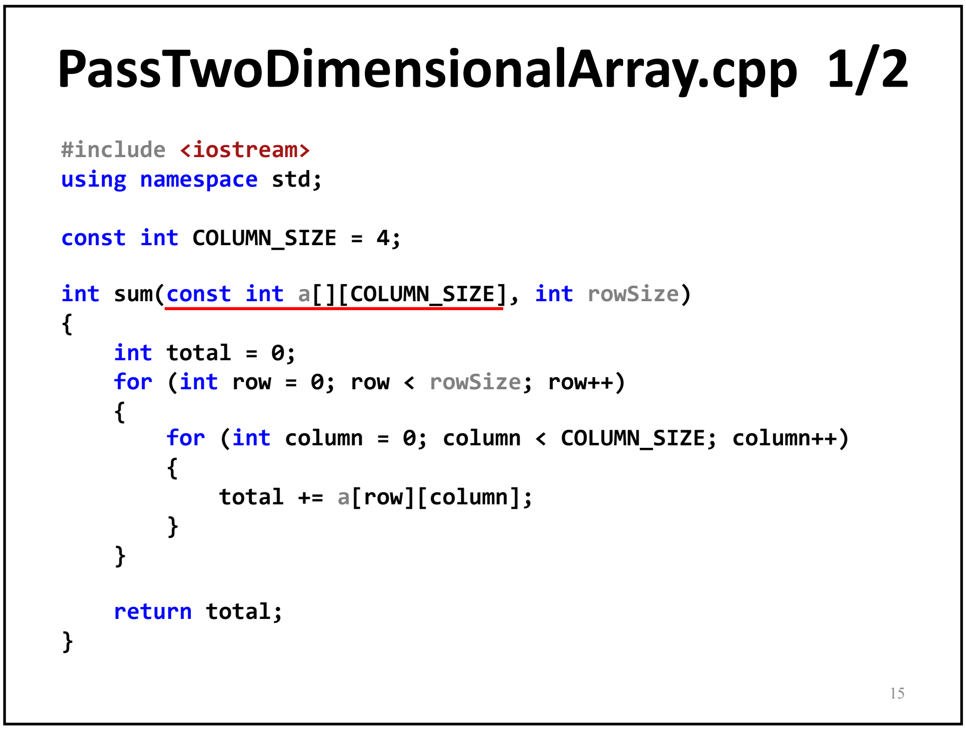
Outline

- Introduction
- Declaring Two-Dimensional Arrays
- Processing Two-Dimensional Arrays
- Passing Two-Dimensional Arrays to Functions
- Problem: Grading a Multiple-Choice Test
- Multidimensional Arrays

Passing Two-Dimensional Arrays to Functions

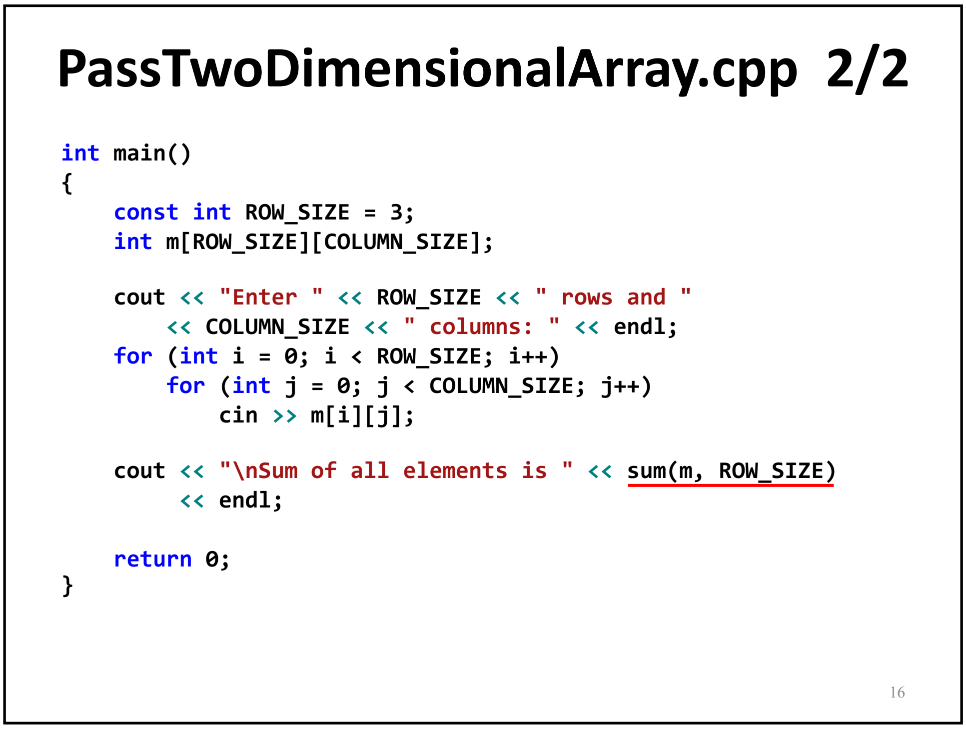
- You can pass a two-dimensional array to a function.
- The column size to be specified in the function declaration.
- A program that for a function that returns the sum of all the elements in a matrix.

PassTwoDimensionalArray Run



PassTwoDimensionalArray.cpp 1/2

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

const int COLUMN_SIZE = 4;

int sum(const int a[][COLUMN_SIZE], int rowSize)
{
    int total = 0;
    for (int row = 0; row < rowSize; row++)
    {
        for (int column = 0; column < COLUMN_SIZE; column++)
        {
            total += a[row][column];
        }
    }

    return total;
}
```



PassTwoDimensionalArray.cpp 2/2

```
int main()
{
    const int ROW_SIZE = 3;
    int m[ROW_SIZE][COLUMN_SIZE];

    cout << "Enter " << ROW_SIZE << " rows and "
        << COLUMN_SIZE << " columns: " << endl;
    for (int i = 0; i < ROW_SIZE; i++)
        for (int j = 0; j < COLUMN_SIZE; j++)
            cin >> m[i][j];

    cout << "\nSum of all elements is " << sum(m, ROW_SIZE)
         << endl;

    return 0;
}
```

Outline

- Introduction
- Declaring Two-Dimensional Arrays
- Processing Two-Dimensional Arrays
- Passing Two-Dimensional Arrays to Functions
- Problem: Grading a Multiple-Choice Test
- Multidimensional Arrays

Problem: Grading Multiple-Choice Test

Key to the Questions:

	0	1	2	3	4	5	6	7	8	9
key	D	B	D	C	C	D	A	E	A	D

Students' Answers to the Questions:

	0	1	2	3	4	5	6	7	8	9
Student 0	A	B	A	C	C	D	E	E	A	D
Student 1	D	B	A	B	C	A	E	E	A	D
Student 2	E	D	D	A	C	B	E	E	A	D
Student 3	C	B	A	E	D	C	E	E	A	D
Student 4	A	B	D	C	C	D	E	E	A	D
Student 5	B	B	E	C	C	D	E	E	A	D
Student 6	B	B	A	C	C	D	E	E	A	D
Student 7	E	B	E	C	C	D	E	E	A	D

GradeExam

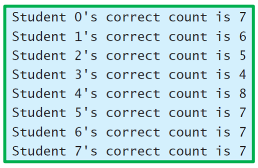
Run

GradeExam.cpp 1/2

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

int main()
{
    const int NUMBER_OF_STUDENTS = 8;
    const int NUMBER_OF_QUESTIONS = 10;

    // Students' answers to the questions
    char answers[NUMBER_OF_STUDENTS][NUMBER_OF_QUESTIONS] =
    {
      {'A', 'B', 'A', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
      {'D', 'B', 'A', 'B', 'C', 'A', 'E', 'E', 'A', 'D'},
      {'E', 'D', 'D', 'A', 'C', 'B', 'E', 'E', 'A', 'D'},
      {'C', 'B', 'A', 'E', 'D', 'C', 'E', 'E', 'A', 'D'},
      {'A', 'B', 'D', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
      {'B', 'B', 'E', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
      {'B', 'B', 'A', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
      {'E', 'B', 'E', 'C', 'C', 'D', 'E', 'E', 'A', 'D'}
    };
```

GradeExam.cpp 2/2

```cpp
    // Key to the questions
    char keys[] = { 'D', 'B', 'D', 'C', 'C', 'D', 'A', 'E', 'A', 'D' };

    // Grade all answers
    for (int i = 0; i < NUMBER_OF_STUDENTS; i++)
    {
        // Grade one student
        int correctCount = 0;
        for (int j = 0; j < NUMBER_OF_QUESTIONS; j++)
        {
            if (answers[i][j] == keys[j])
                correctCount++;
        }

        cout << "Student " << i << "'s correct count is " <<
            correctCount << endl;
    }

    return 0;
}
```



```
Student 0's correct count is 7
Student 1's correct count is 6
Student 2's correct count is 5
Student 3's correct count is 4
Student 4's correct count is 8
Student 5's correct count is 7
Student 6's correct count is 7
Student 7's correct count is 7
```

Outline

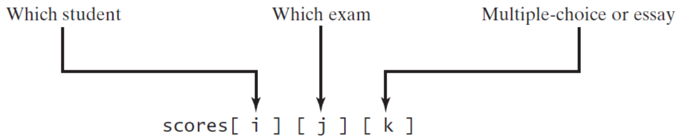
- Introduction
- Declaring Two-Dimensional Arrays
- Processing Two-Dimensional Arrays
- Passing Two-Dimensional Arrays to Functions
- Problem: Grading a Multiple-Choice Test
- Multidimensional Arrays

Multidimensional Arrays

You can create *n*-dimensional arrays for any integer *n*.

For example, you may use a three-dimensional array to store exam scores for a class of **6** students with **5** exams and each exam has **2** parts (multiple-choice and essay).

```
double scores[6][5][2];
```



With initialization:

```
double scores[6][5][2] = {
{{7.5, 20.5}, {9.0, 22.5}, {15, 33.5}, {13, 21.5}, {15, 2.5}},
{{4.5, 21.5}, {9.0, 22.5}, {15, 34.5}, {12, 20.5}, {14, 9.5}},
{{6.5, 30.5}, {9.4, 10.5}, {11, 33.5}, {11, 23.5}, {10, 2.5}},
{{6.5, 23.5}, {9.4, 32.5}, {13, 34.5}, {11, 20.5}, {16, 7.5}},
{{8.5, 26.5}, {9.4, 52.5}, {13, 36.5}, {13, 24.5}, {16, 2.5}},
{{9.5, 20.5}, {9.4, 42.5}, {13, 31.5}, {12, 20.5}, {16, 6.5}} };
```

Problem: Daily Temperature and Humidity

- Suppose a meteorology station records the temperature and humidity at each hour of every day and stores the data for the past ten days in a text file named weather.txt.
- Each line of the file consists of four numbers that indicates the day, hour, temperature, and humidity.

```
1  1  76.4  0.92
1  2  77.7  0.93
...
10  23  97.7  0.71
10  24  98.7  0.74
```

```
Weather.exe < Weather.txt
```

A program that calculates the average daily temperature and humidity for the 10 days.

Weather Run

Weather.cpp 1/2

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

int main()
{
    const int NUMBER_OF_DAYS = 10;
    const int NUMBER_OF_HOURS = 24;
    double data[NUMBER_OF_DAYS][NUMBER_OF_HOURS][2];

    // Read input using input redirection from a file
    int day, hour;
    double temperature, humidity;
    for (int k = 0; k < NUMBER_OF_DAYS * NUMBER_OF_HOURS; k++)
    {
        cin >> day >> hour >> temperature >> humidity;
        data[day - 1][hour - 1][0] = temperature;
        data[day - 1][hour - 1][1] = humidity;
    }
```

Weather.cpp 2/2

```
    // Find the average daily temperature and humidity
    for (int i = 0; i < NUMBER_OF_DAYS; i++)
    {
        double dailyTemperatureTotal = 0, dailyHumidityTotal = 0;
        for (int j = 0; j < NUMBER_OF_HOURS; j++)
        {
            dailyTemperatureTotal += data[i][j][0];
            dailyHumidityTotal += data[i][j][1];
        }

        // Display result
        cout << "Day " << i << "'s average temperature is "
            << dailyTemperatureTotal / NUMBER_OF_HOURS << endl;
        cout << "Day " << i << "'s average humidity is "
            << dailyHumidityTotal / NUMBER_OF_HOURS << endl;
    }

    return 0;
}
```

```
Day 0's average temperature is 77.7708
Day 0's average humidity is 0.929583
Day 1's average temperature is 77.3125
Day 1's average humidity is 0.929583
...
```

Problem: Guessing Birthday

- Listing 4.4, GuessBirthday.cpp, gives a program that guesses a birthday.
- The program can be simplified by storing the numbers in five sets in a three-dimensional array, and it prompts the user for the answers using a loop.

GuessBirthdayUsingArray Run

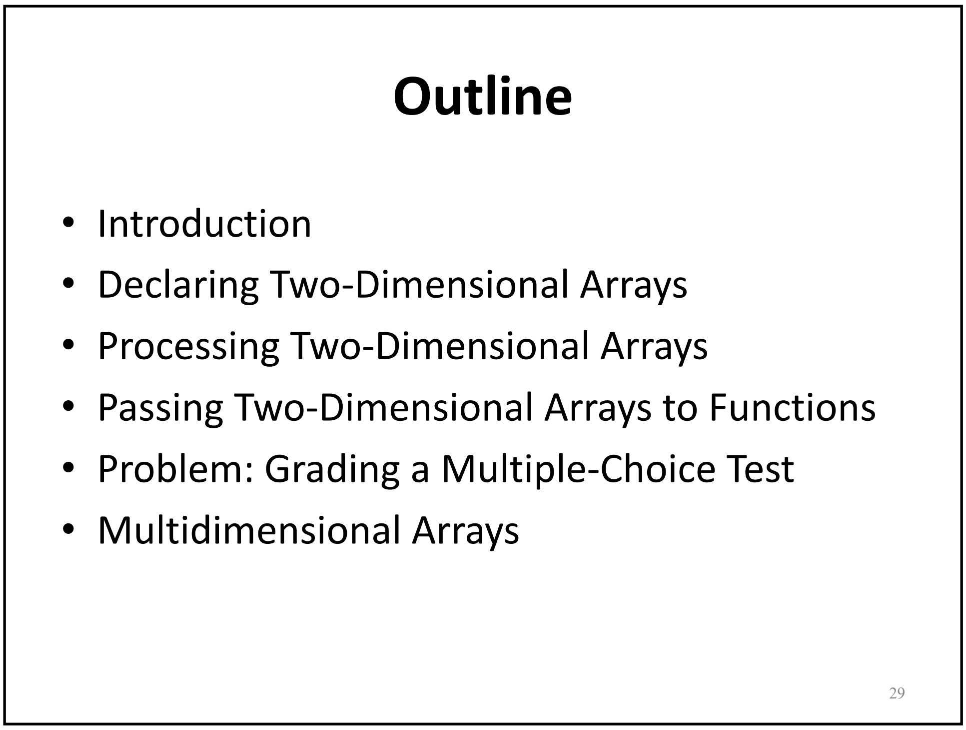
GuessBirthdayUsingArray.cpp 1/2

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

int main()
{
    int day = 0; // Day to be determined
    char answer;
```

```cpp
int dates[5][4][4] = {
  {{ 1,  3,  5,  7},
   { 9, 11, 13, 15},
   {17, 19, 21, 23},
   {25, 27, 29, 31}},
  {{ 2,  3,  6,  7},
   {10, 11, 14, 15},
   {18, 19, 22, 23},
   {26, 27, 30, 31}},
  {{ 4,  5,  6,  7},
   {12, 13, 14, 15},
   {20, 21, 22, 23},
   {28, 29, 30, 31}},
  {{ 8,  9, 10, 11},
   {12, 13, 14, 15},
   {24, 25, 26, 27},
   {28, 29, 30, 31}},
  {{16, 17, 18, 19},
   {20, 21, 22, 23},
   {24, 25, 26, 27},
   {28, 29, 30, 31}} };
```

GuessBirthdayUsingArray.cpp 1/2

```cpp
    for (int i = 0; i < 5; i++)
    {
        cout << "Is your birthday in Set" << (i + 1) << "?" << endl;
        for (int j = 0; j < 4; j++)
        {
            for (int k = 0; k < 4; k++)
                cout << setw(3) << dates[i][j][k] << " ";
            cout << endl;
        }
        cout << "\nEnter N/n for No and Y/y for Yes: ";
        cin >> answer;
        if (answer == 'Y' || answer == 'y')
            day += dates[i][0][0];
    }

    cout << "Your birthday is " << day << endl;

    return 0;
}
```



Outline

- Introduction
- Declaring Two-Dimensional Arrays
- Processing Two-Dimensional Arrays
- Passing Two-Dimensional Arrays to Functions
- Problem: Grading a Multiple-Choice Test
- Multidimensional Arrays

Chapter 17: Recursion

Sections 17.1–17.2

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)

Outline

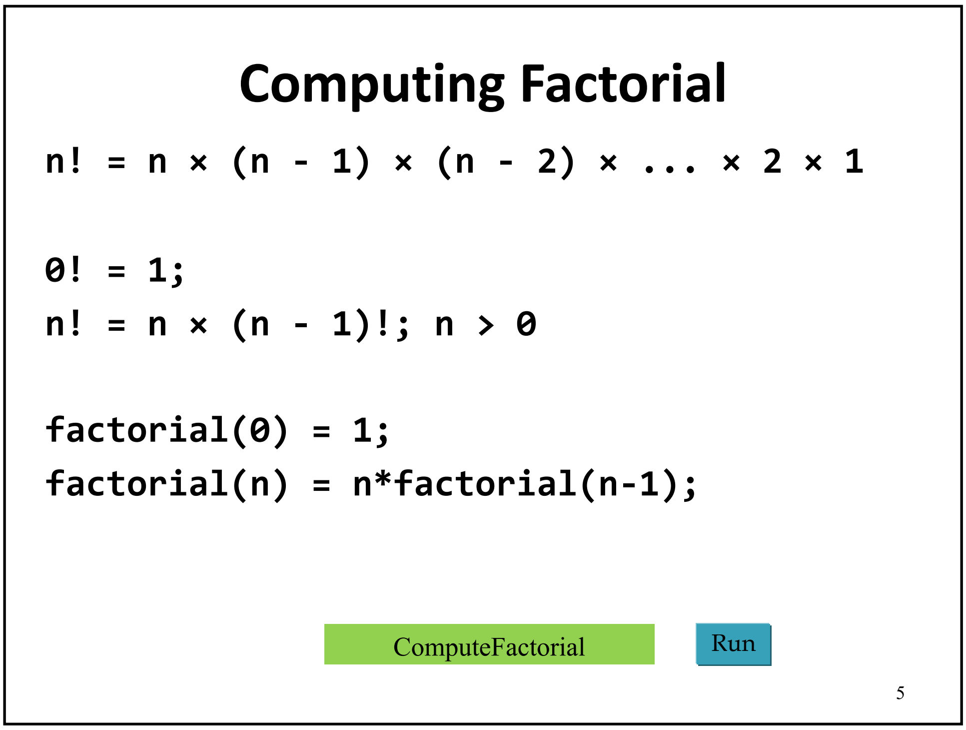
- Introduction
- Example: Factorials

Motivations

- *Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops.*
- *A recursive function is one that invokes itself.*
- Suppose you want to find all the files under a directory that contains a particular word. How do you solve this problem? There are several ways to solve this problem. An intuitive solution is to use recursion by searching the files in the subdirectories recursively.

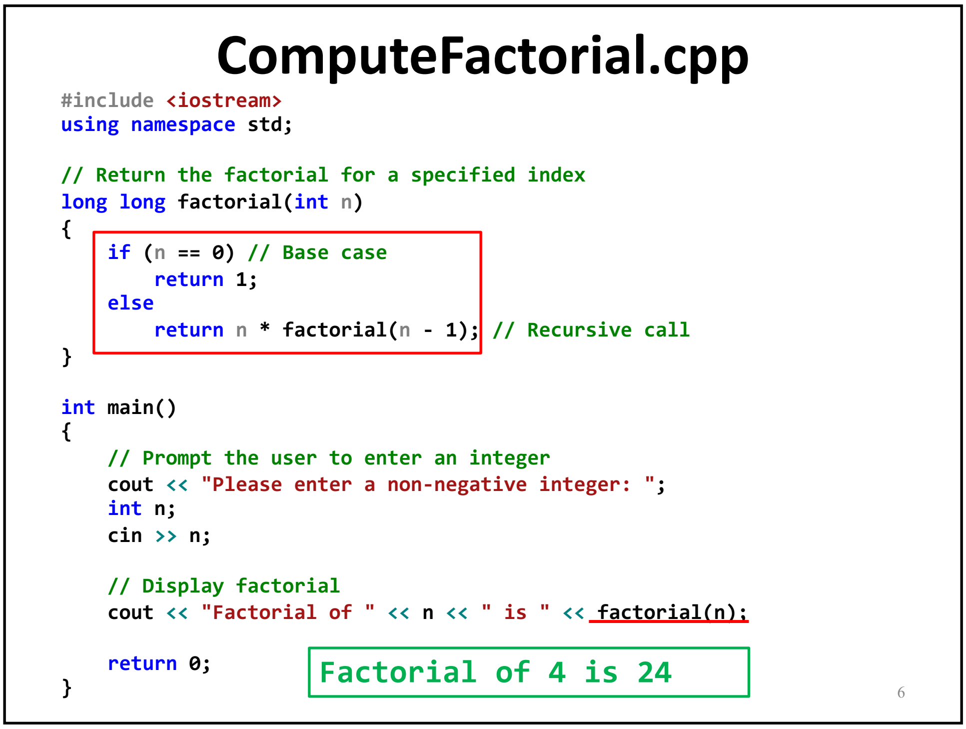
Outline

- Introduction
- Example: Factorials



Computing Factorial

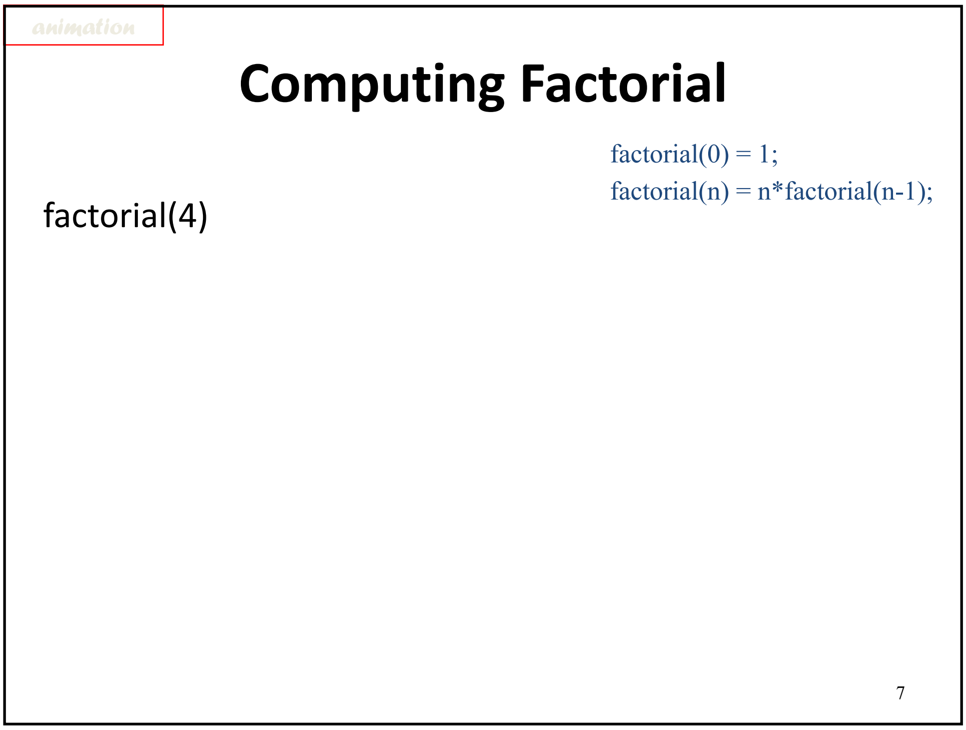
```
n! = n × (n - 1) × (n - 2) × ... × 2 × 1

0! = 1;
n! = n × (n - 1)!; n > 0

factorial(0) = 1;
factorial(n) = n*factorial(n-1);
```



ComputeFactorial.cpp

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

// Return the factorial for a specified index
long long factorial(int n)
{
    if (n == 0) // Base case
        return 1;
    else
        return n * factorial(n - 1); // Recursive call
}

int main()
{
    // Prompt the user to enter an integer
    cout << "Please enter a non-negative integer: ";
    int n;
    cin >> n;

    // Display factorial
    cout << "Factorial of " << n << " is " << factorial(n);

    return 0;
}
```

```
Factorial of 4 is 24
```

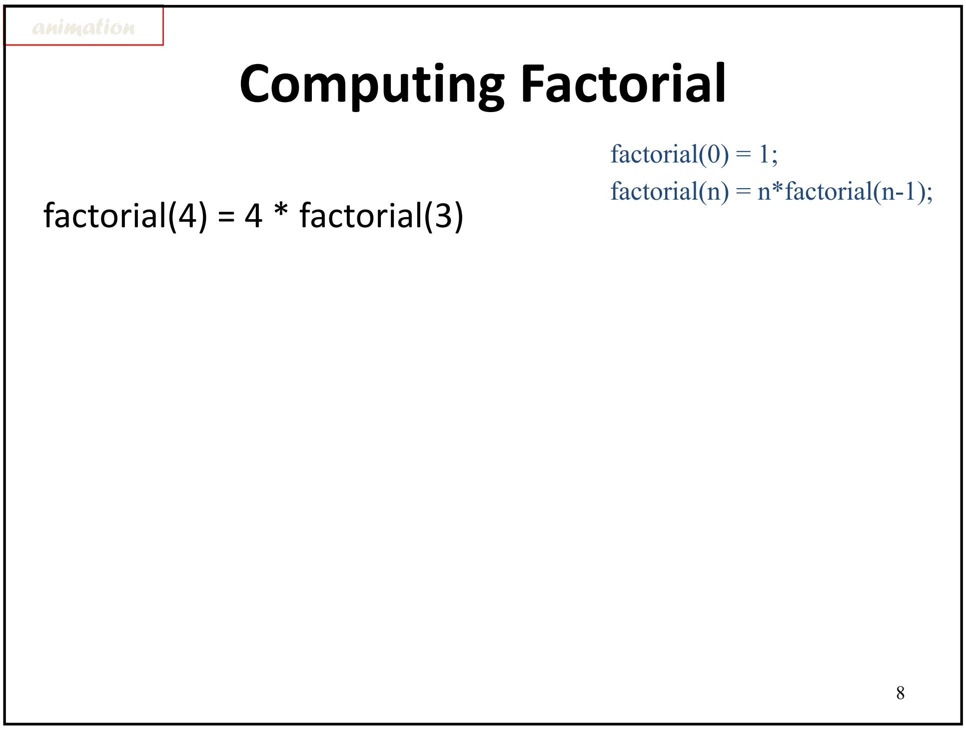


animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4)



animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)

---

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))
= 4 * 3 * (2 * 1)

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))
= 4 * 3 * (2 * 1)
= 4 * 3 * 2

Computing Factorial

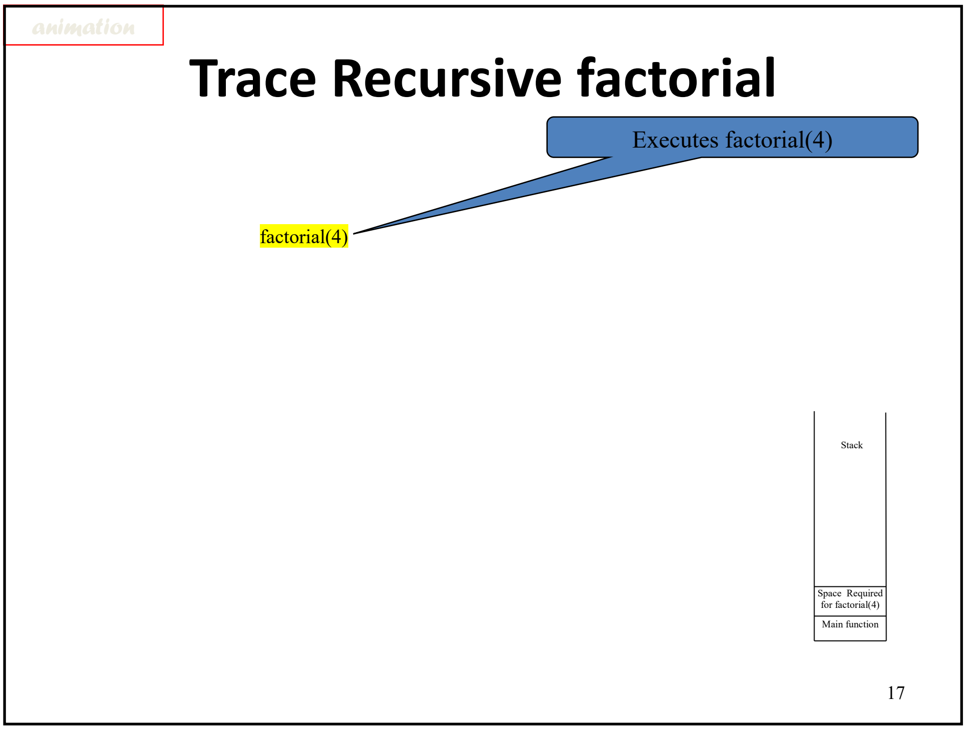
factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))
= 4 * 3 * (2 * 1)
= 4 * 3 * 2
= 4 * 6

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))
= 4 * 3 * (2 * 1)
= 4 * 3 * 2
= 4 * 6
= 24



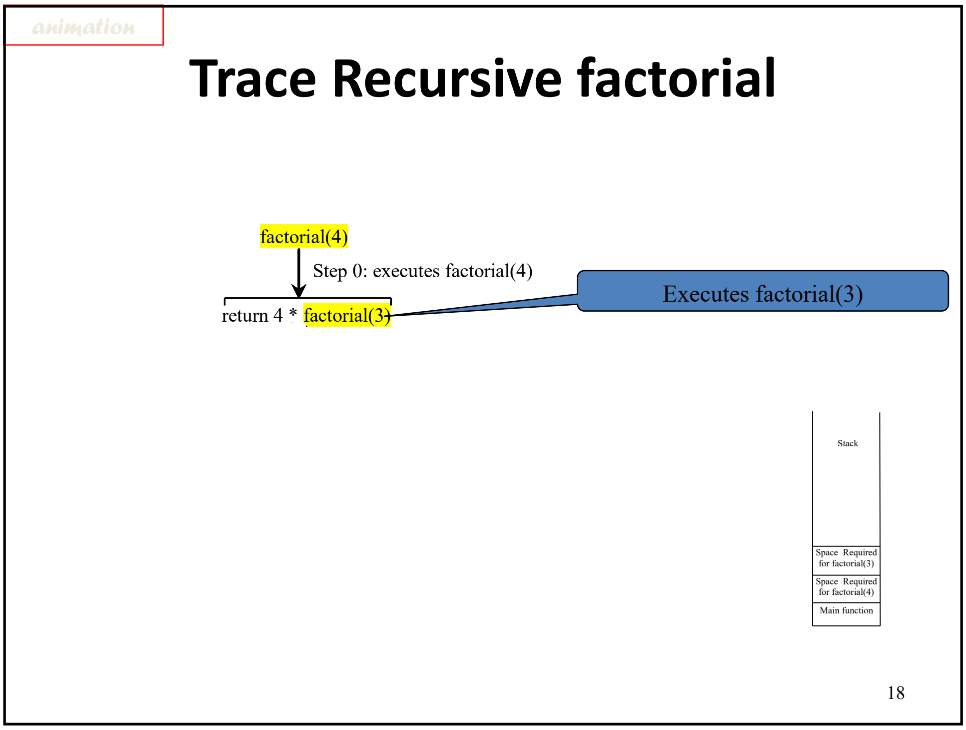
animation

Trace Recursive factorial

Executes factorial(4)

factorial(4)

17



animation

Trace Recursive factorial

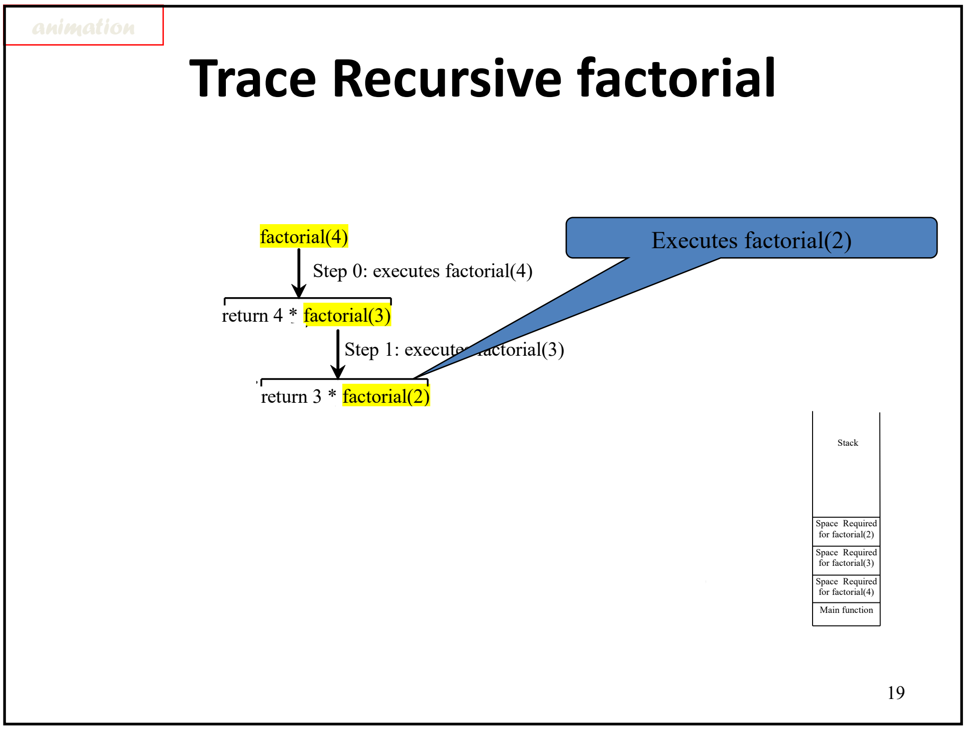
factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Executes factorial(3)

18



animation

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: execute actorial(3)

return 3 * factorial(2)

Executes factorial(2)

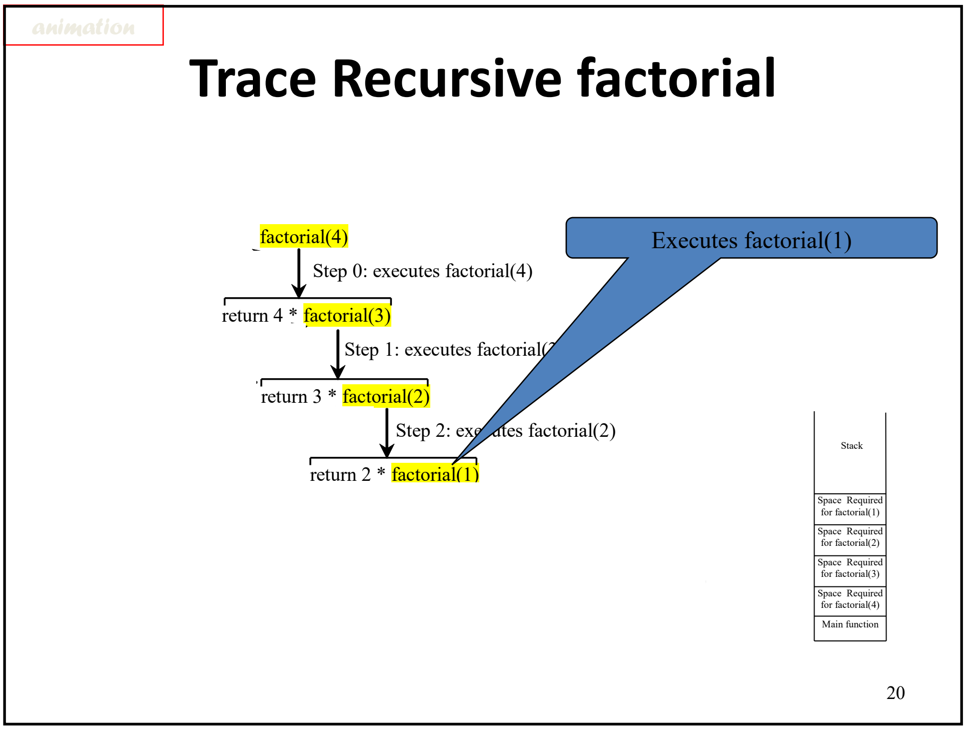
Stack

Space Required for factorial(2)

Space Required for factorial(3)

Space Required for factorial(4)

Main function



animation

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(

return 3 * factorial(2)

Step 2: exe utes factorial(2)

return 2 * factorial(1)

Executes factorial(1)

Stack

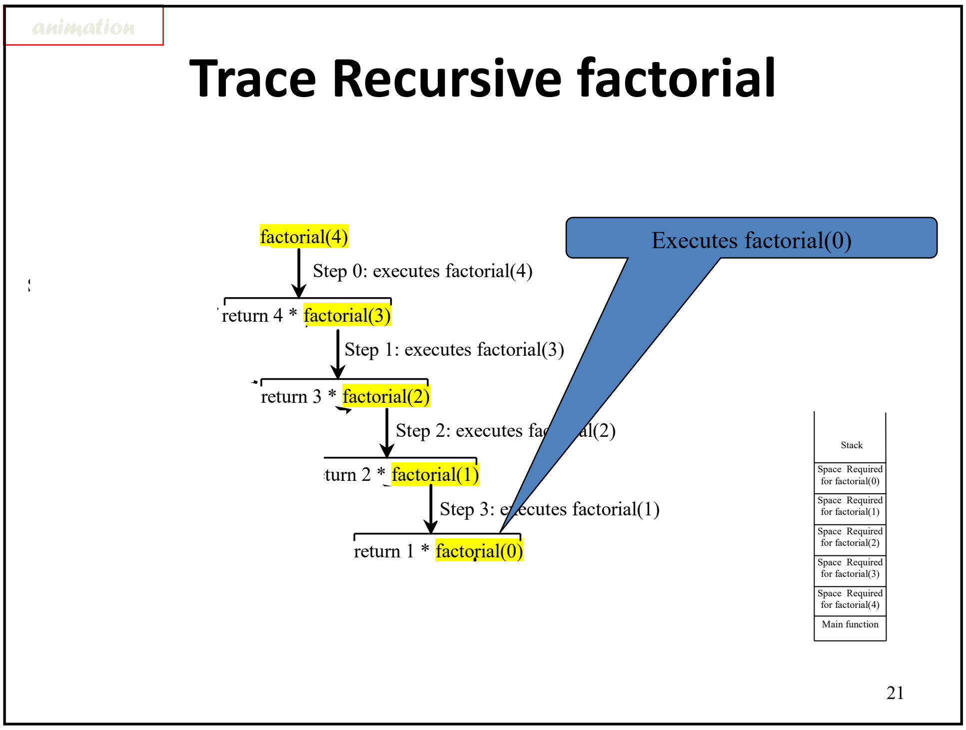
Space Required for factorial(1)

Space Required for factorial(2)

Space Required for factorial(3)

Space Required for factorial(4)

Main function



animation

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

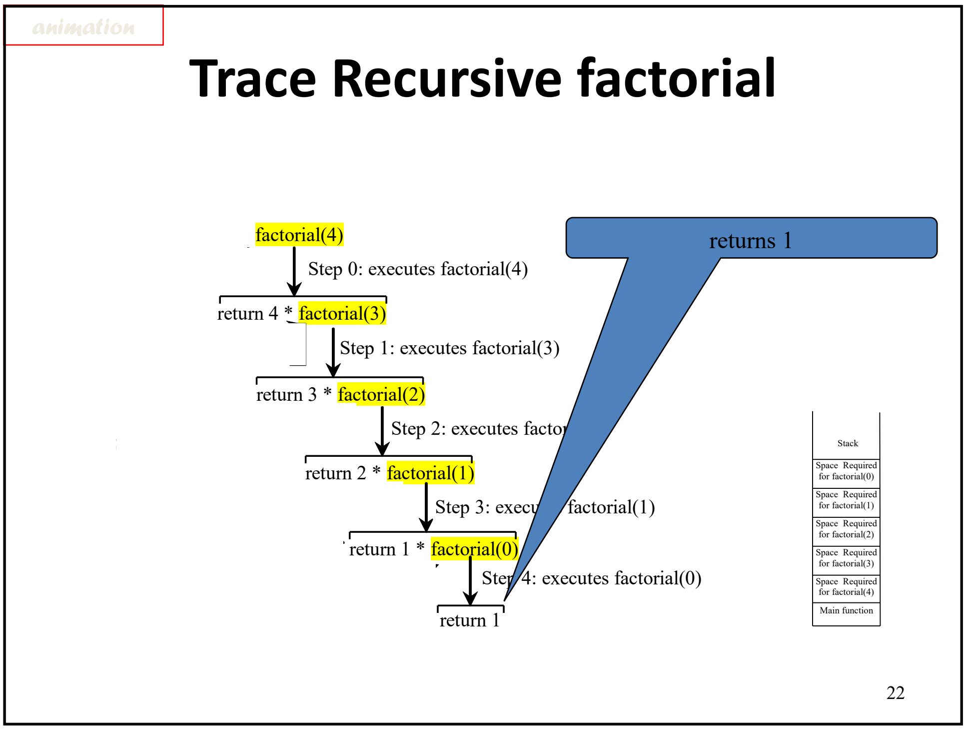
Step 2: executes fa al(2)

turn 2 * factorial(1)

Step 3: e ecutes factorial(1)

return 1 * factorial(0)

Executes factorial(0)



animation

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes facto

return 2 * factorial(1)

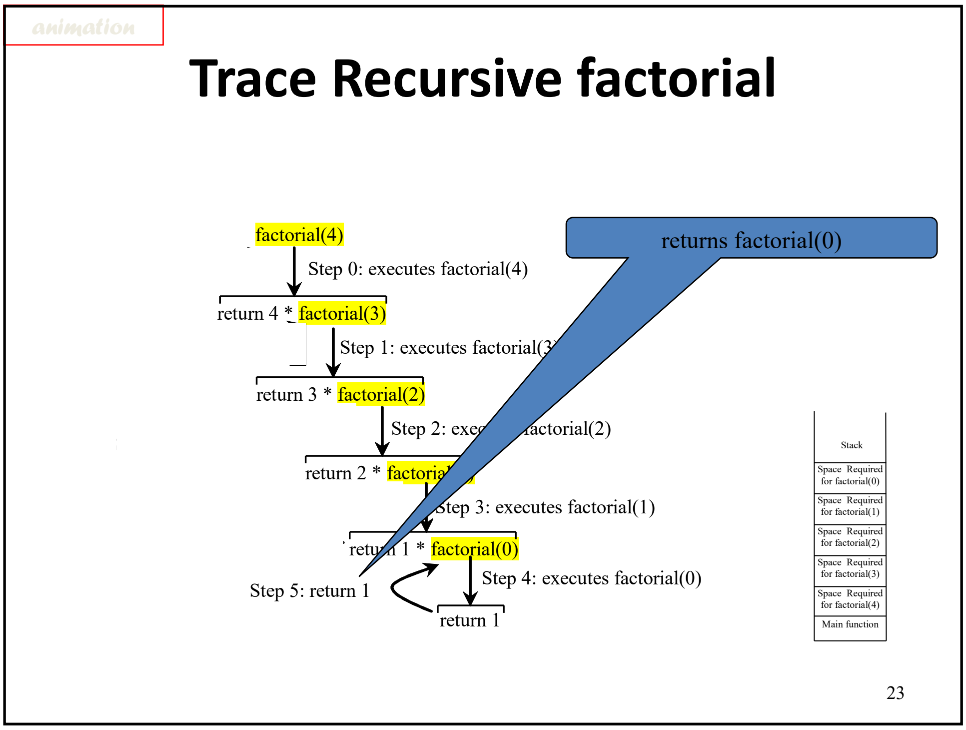
Step 3: execu factorial(1)

return 1 * factorial(0)

Ste 4: executes factorial(0)

return 1

returns 1



animation

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3

return 3 * factorial(2)

Step 2: exec actorial(2)

return 2 * factoria

Step 3: executes factorial(1)

retu 1 * factorial(0)

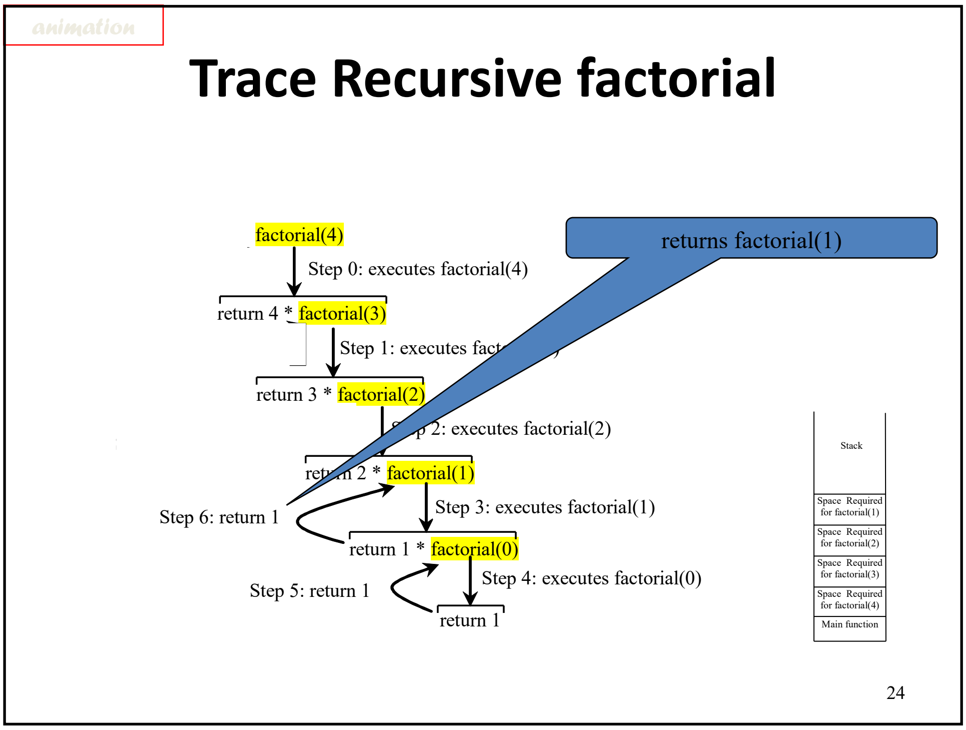
Step 4: executes factorial(0)

Step 5: return 1

return 1

returns factorial(0)

Stack
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main function



animation

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes fact)

return 3 * factorial(2)

p 2: executes factorial(2)

ret n 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

return 1 * factorial(0)

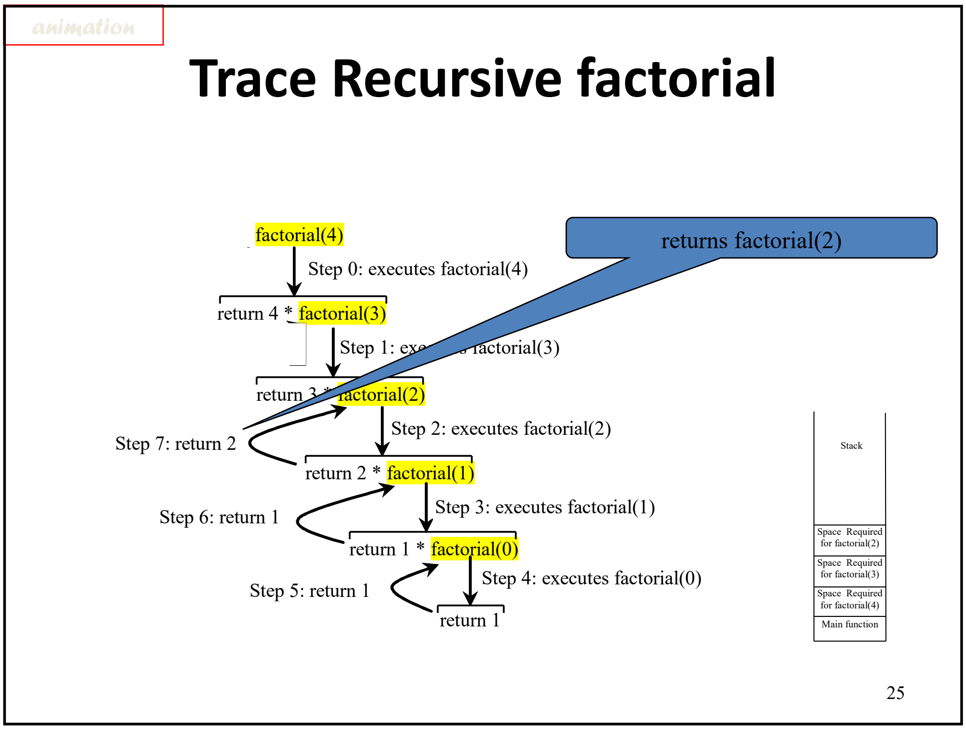
Step 4: executes factorial(0)

Step 5: return 1

return 1

returns factorial(1)

Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main function



Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: exe factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

returns factorial(2)

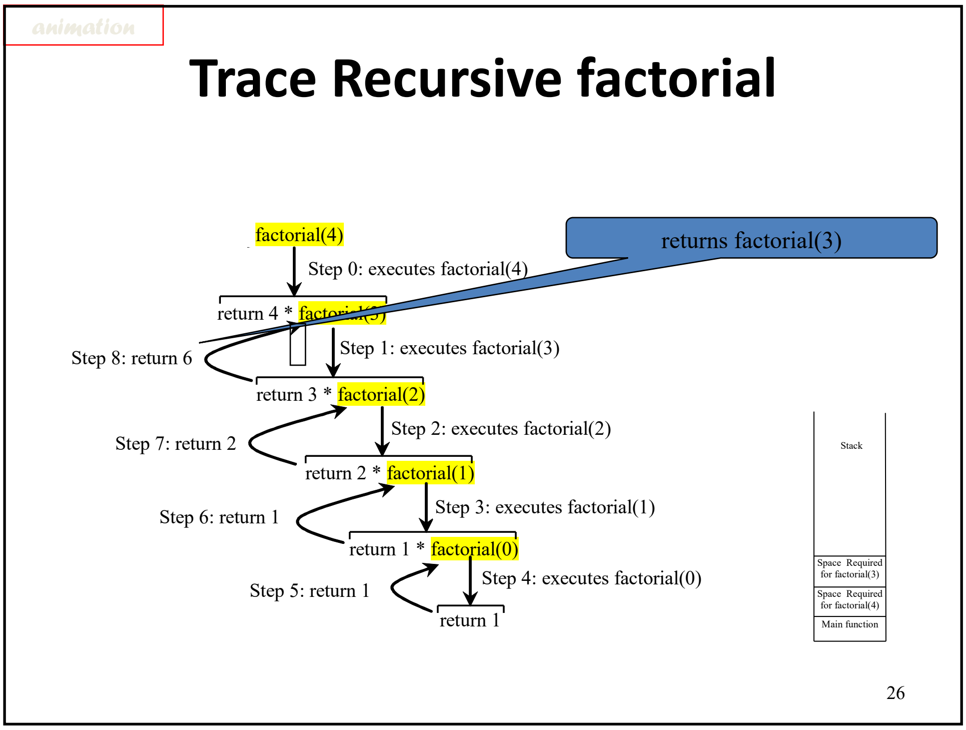
Stack

Space Required for factorial(2)

Space Required for factorial(3)

Space Required for factorial(4)

Main function



Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 8: return 6

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

returns factorial(3)

Stack

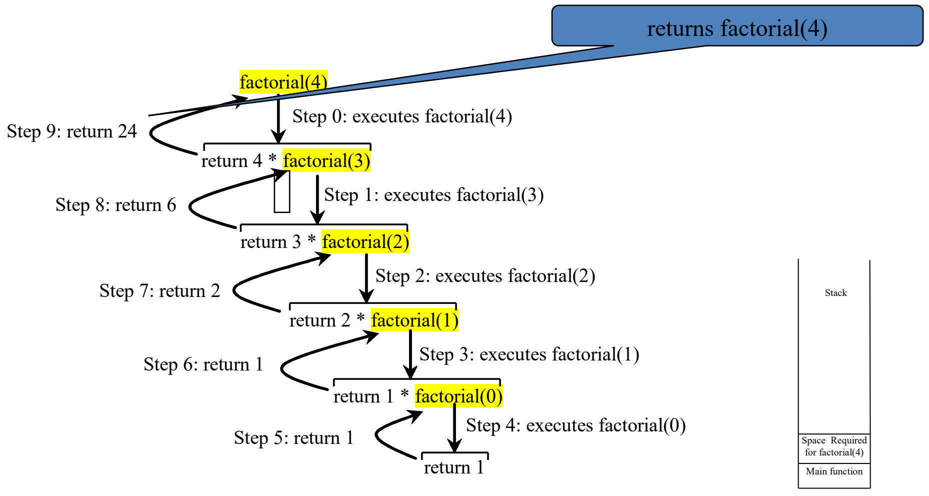
Space Required for factorial(3)

Space Required for factorial(4)

Main function

animation

Trace Recursive factorial



returns factorial(4)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes factorial(3)

Step 8: return 6

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

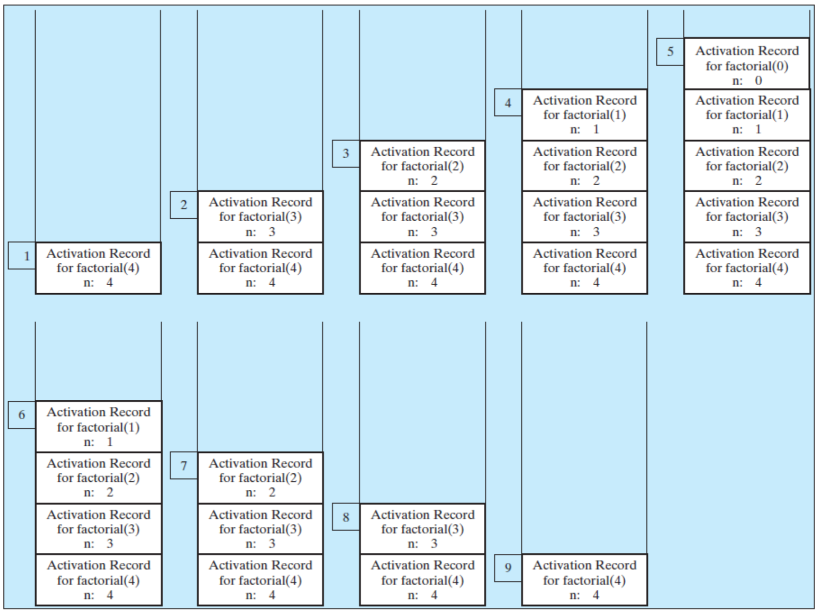
return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

`factorial(4)` Stack Trace



1
Activation Record for factorial(4) n: 4

2
Activation Record for factorial(3) n: 3
Activation Record for factorial(4) n: 4

3
Activation Record for factorial(2) n: 2
Activation Record for factorial(3) n: 3
Activation Record for factorial(4) n: 4

4
Activation Record for factorial(1) n: 1
Activation Record for factorial(2) n: 2
Activation Record for factorial(3) n: 3
Activation Record for factorial(4) n: 4

5
Activation Record for factorial(0) n: 0
Activation Record for factorial(1) n: 1
Activation Record for factorial(2) n: 2
Activation Record for factorial(3) n: 3
Activation Record for factorial(4) n: 4

6
Activation Record for factorial(1) n: 1
Activation Record for factorial(2) n: 2
Activation Record for factorial(3) n: 3
Activation Record for factorial(4) n: 4

7
Activation Record for factorial(2) n: 2
Activation Record for factorial(3) n: 3
Activation Record for factorial(4) n: 4

8
Activation Record for factorial(3) n: 3
Activation Record for factorial(4) n: 4

9
Activation Record for factorial(4) n: 4

Outline

- Introduction
- Example: Factorials

Chapter 9: Objects and Classes

Sections 9.1–9.6, 9.9

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)

Outline

- Introduction
- Defining Classes for Objects
- Example: Defining Classes and Creating Objects
- Constructors
- Constructing and Using Objects
- Separating Class Definition from Implementation
- Data Field Encapsulation

Introduction

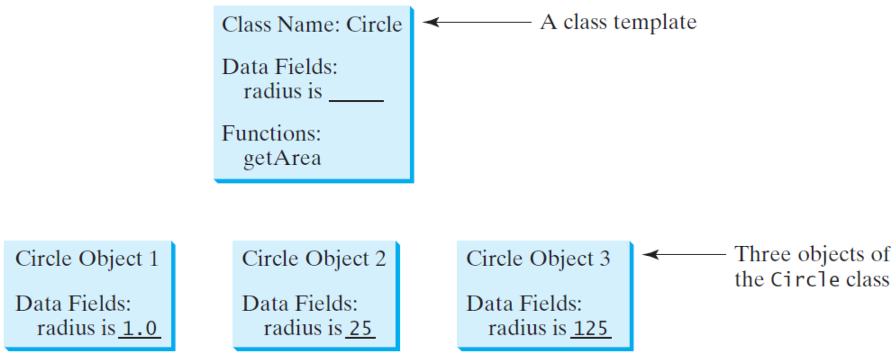
- *Object-oriented programming* (OOP) involves programming using objects.
- An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.
- An object has a unique identity, state, and behaviors.
- The *state* of an object consists of a set of *data fields* (also known as *properties*) with their current values.
- The *behavior* of an object is defined by a set of functions.

Outline

- Introduction
- Defining Classes for Objects
- Example: Defining Classes and Creating Objects
- Constructors
- Constructing and Using Objects
- Separating Class Definition from Implementation
- Data Field Encapsulation

Classes and Objects

A class defines the properties and behaviors for objects..



Class Name: Circle

Data Fields:
radius is ____

Functions:
getArea

A class template

Circle Object 1

Data Fields:
radius is 1.0

Circle Object 2

Data Fields:
radius is 25

Circle Object 3

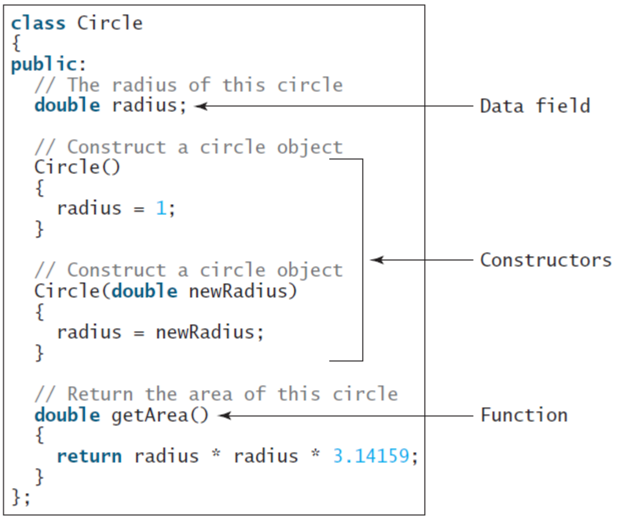
Data Fields:
radius is 125

Three objects of the `Circle` class

Classes

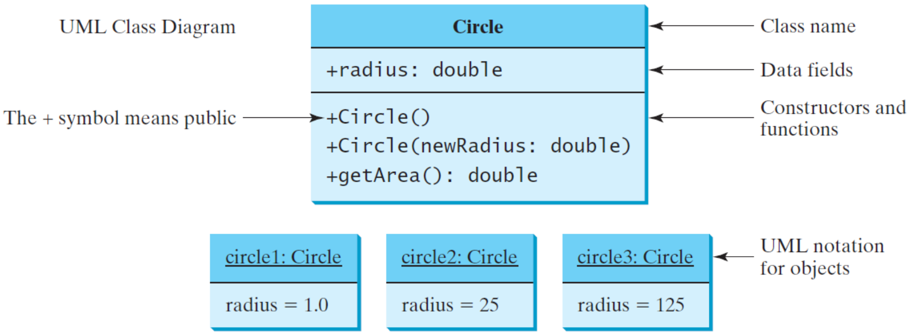
- *Classes* are constructs that define objects of the same type.
- A class uses *variables* to define data fields and *functions* to define behaviors.
- Additionally, a class provides a special type of functions, known as *constructors*, which are invoked to construct objects from the class.

Example of the class for `Circle` objects



```
class Circle
{
public:
  // The radius of this circle
  double radius;

  // Construct a circle object
  Circle()
  {
    radius = 1;
  }

  // Construct a circle object
  Circle(double newRadius)
  {
    radius = newRadius;
  }

  // Return the area of this circle
  double getArea()
  {
    return radius * radius * 3.14159;
  }
};
```

UML Class Diagram



`class` Replaces `struct`

- The C language has the `struct` type for representing records.
- For example, you may define a `struct` type for representing students as shown in (a).
- C++ `class` allows functions in addition to data fields. `class` replaces `struct`, as in (b)

```
struct Student
{
  int id;
  char firstName[30];
  char mi;
  char lastName[30];
};
```

(a)

```
class Student
{
public:
  int id;
  char firstName[30];
  char mi;
  char lastName[30];
};
```

(b)

Outline

- Introduction
- Defining Classes for Objects
- Example: Defining Classes and Creating Objects
- Constructors
- Constructing and Using Objects
- Separating Class Definition from Implementation
- Data Field Encapsulation

A Simple Circle Class

- Objective: Demonstrate creating objects, accessing data, and using functions.

TestCircle Run

TestCircle.cpp 1/2

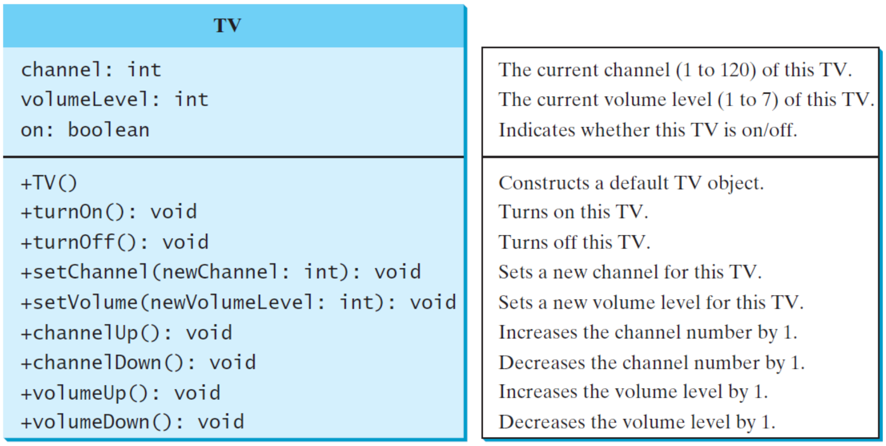
```
#include <iostream>
using namespace std;

class Circle
{
public:
    // The radius of this circle
    double radius;

    // Construct a default object
    Circle()
    {
        radius = 1;
    }

    // Construct a circle object
    Circle(double newRadius)
    {
        radius = newRadius;
    }

    // Return the area of this circle
    double getArea()
    {
        return radius * radius * 3.14159;
    }

    // Return the perimeter of this circle
    double getPermeter()
    {
        return 2 * radius * 3.14159;
    }

    // Set new radius for this circle
    void setRadius(double newRadius)
    {
        radius = newRadius;
    }
}; // Must place a semicolon here
```

TestCircle.cpp 2/2

```
int main()
{
   Circle circle1(1.0);
   Circle circle2(25);
   Circle circle3(125);

   cout << "The area of the circle of radius "
       << circle1.radius << " is " << circle1.getArea() << endl;
   cout << "The area of the circle of radius "
       << circle2.radius << " is " << circle2.getArea() << endl;
   cout << "The area of the circle of radius "
       << circle3.radius << " is " << circle3.getArea() << endl;

   // Modify circle radius
   circle2.radius = 100;
   cout << "The area of the circle of radius "
       << circle2.radius << " is " << circle2.getArea() << endl;

   return 0;
}
```

```
The area of the circle of radius 1 is 3.14159
The area of the circle of radius 25 is 1963.49
The area of the circle of radius 125 is 49087.3
The area of the circle of radius 100 is 31415.9
```

Example: The TV class models TV sets



TV	
channel: int	The current channel (1 to 120) of this TV.
volumeLevel: int	The current volume level (1 to 7) of this TV.
on: boolean	Indicates whether this TV is on/off.
+TV()	Constructs a default TV object.
+turnOn(): void	Turns on this TV.
+turnOff(): void	Turns off this TV.
+setChannel(newChannel: int): void	Sets a new channel for this TV.
+setVolume(newVolumeLevel: int): void	Sets a new volume level for this TV.
+channelUp(): void	Increases the channel number by 1.
+channelDown(): void	Decreases the channel number by 1.
+volumeUp(): void	Increases the volume level by 1.
+volumeDown(): void	Decreases the volume level by 1.

TV Run

TV.cpp 1/4

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

class TV
{
public:
    int channel;
    int volumeLevel; // Default volume level is 1
    bool on; // By default TV is off

    TV()
    {
        channel = 1; // Default channel is 1
        volumeLevel = 1; // Default volume level is 1
        on = false; // By default TV is off
    }

    void turnOn()
    {
        on = true;
    }
```

TV.cpp 2/4

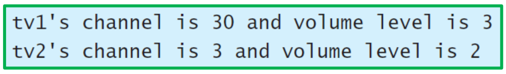
```
    void turnOff()
    {
        on = false;
    }

    void setChannel(int newChannel)
    {
        if (on && newChannel >= 1 && newChannel <= 120)
            channel = newChannel;
    }

    void setVolume(int newVolumeLevel)
    {
        if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
            volumeLevel = newVolumeLevel;
    }

    void channelUp()
    {
        if (on && channel < 120)
            channel++;
    }
```

TV.cpp 3/4

```
    void channelDown()
    {
        if (on && channel > 1)
            channel--;
    }

    void volumeUp()
    {
        if (on && volumeLevel < 7)
            volumeLevel++;
    }

    void volumeDown()
    {
        if (on && volumeLevel > 1)
            volumeLevel--;
    }
};
```

TV.cpp 4/4

```
int main()
{
    TV tv1;
    tv1.turnOn();
    tv1.setChannel(30);
    tv1.setVolume(3);

    TV tv2;
    tv2.turnOn();
    tv2.channelUp();
    tv2.channelUp();
    tv2.volumeUp();

    cout << "tv1's channel is " << tv1.channel
        << " and volume level is " << tv1.volumeLevel << endl;
    cout << "tv2's channel is " << tv2.channel
        << " and volume level is " << tv2.volumeLevel << endl;

    return 0;
}
```



```
tv1's channel is 30 and volume level is 3
tv2's channel is 3 and volume level is 2
```

Outline

- Introduction
- Defining Classes for Objects
- Example: Defining Classes and Creating Objects
- Constructors
- Constructing and Using Objects
- Separating Class Definition from Implementation
- Data Field Encapsulation

Constructors

- The constructor has exactly the same name as the defining class.
- Constructors can be overloaded (i.e., multiple constructors with the same name but different signatures).
- A class normally provides a constructor without arguments (e.g., **`Circle()`**). Such constructor is called a *no-arg* or *no-argument constructor*.
- A class may be declared without constructors. In this case, a no-arg constructor with an empty body is implicitly declared in the class. This constructor is called a *default constructor*.

Constructors Features

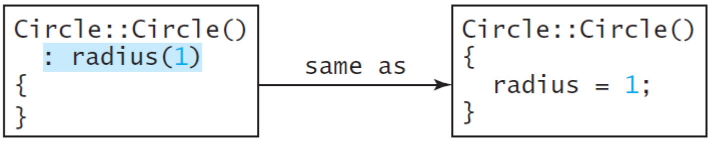
- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even **`void`**.
- Constructors play the role of initializing objects.

Initializer Lists

- Data fields may be initialized in the constructor using an initializer list in the following syntax:

```
ClassName(parameterList)
  : datafield1(value1), datafield2(value2) // Initializer list
{
  // Additional statements if needed
}
```

- Example:



```
Circle::Circle()
  : radius(1)
{
}
```

same as

```
Circle::Circle()
{
  radius = 1;
}
```

Outline

- Introduction
- Defining Classes for Objects
- Example: Defining Classes and Creating Objects
- Constructors
- Constructing and Using Objects
- Separating Class Definition from Implementation
- Data Field Encapsulation

Object Names

- You can assign a name when creating an object.
- A constructor is invoked when an object is created.
- The syntax to create an object using the no-arg constructor is:

```
ClassName objectName;
```

- For example,

```
Circle circle1;
```

- The size of and object depends on its data fields only.

```
cout << sizeof(circle1) << endl;;
8
```

Constructing with Arguments

- The syntax to declare an object using a constructor with arguments is:

```
ClassName objectName(arguments);
```

- For example, the following declaration creates an object named **`circle2`** by invoking the **`Circle`** class's constructor with a specified radius **`5.5`**.

```
Circle circle2(5.5);
```

Access Operator

- After an object is created, its data can be accessed and its functions invoked using the dot operator (**`.`**), also known as the *object member access operator*:

- **`objectName.dataField`** references a data field in the object.

- **`objectName.function(arguments)`** invokes a function on the object.

Naming Objects and Classes

- When you declare a custom class, capitalize the first letter of each word in a class name; for example, the class names **`Circle`**, **`Rectangle`**, and **`Desk`**.
- The class names in the C++ library are named in lowercase.
- The objects are named like variables.

Class is a Type

- You can use primitive data types, like **`int`**, to declare variables.
- You can also use class names to declare object names.
- In this sense, a class is also a data type.

Memberwise Copy

- You can also use the assignment operator **=** to copy the contents from one object to the other.
- By default, each data field of one object is copied to its counterpart in the other object. For example,

```
circle2 = circle1;
```

- Copies the **`radius`** in **`circle1`** to **`circle2`**.
- After the copy, **`circle1`** and **`circle2`** are still two different objects, but with the same radius.

Constant Object Name

- Object names are like array names. Once an object name is declared, it represents an object.
- It cannot be reassigned to represent another object.
- In this sense, an object name is a constant, though the contents of the object may change.

Anonymous Object

- Most of the time, you create a named object and later access the members of the object through its name.
- Occasionally, you may create an object and use it only once. In this case, you don't have to name the object. Such objects are called *anonymous objects*.
- The syntax to create an anonymous object is

  `ClassName()` or `ClassName(arguements)`
- You can create an anonymous object just for finding the area by:

```
cout << "Area:" << Circle(5).getArea() << endl;
```

Outline

- Introduction
- Defining Classes for Objects
- Example: Defining Classes and Creating Objects
- Constructors
- Constructing and Using Objects
- Separating Class Definition from Implementation
- Data Field Encapsulation

Separating Definition from Implementation

- C++ allows you to separate class *definition* from *implementation*.
- The class definition describes the contract of the class and the class implementation implements the contract.
- The class declaration simply lists all the data fields, constructor prototypes, and the function prototypes.
- The class implementation implements the constructors and functions.
- The class declaration and implementation are in two separate files. Both files should have the same name, but with different extension names. The class declaration file has an extension name *.h* and the class implementation file has an extension name *.cpp*.

Circle.h Circle.cpp TestCircleWithHeader.cpp Run

Circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H
class Circle
{
public:
    // The radius of this circle
    double radius;

    // Construct a default circle object
    Circle();

    // Construct a circle object
    Circle(double);

    // Return the area of this circle
    double getArea();
};
#endif
```

Used to prevent a header file from being included multiple times.

Circle.cpp

```
#include "Circle.h"

// Construct a default circle object
Circle::Circle()
{
    radius = 1;
}

// Construct a circle object
Circle::Circle(double newRadius)
{
    radius = newRadius;
}

// Return the area of this circle
double Circle::getArea()
{
    return radius * radius * 3.14159;
}
```

The `::` symbol is the *binary scope resolution operator*

TestCircleWithHeader.cpp

```
#include <iostream>
#include "Circle.h"
using namespace std;

int main()
{
    Circle circle1;
    Circle circle2(5.0);

    cout << "The area of the circle of radius "
        << circle1.radius << " is " << circle1.getArea() << endl;
    cout << "The area of the circle of radius "
        << circle2.radius << " is " << circle2.getArea() << endl;

    // Modify circle radius
    circle2.radius = 100;
    cout << "The area of the circle of radius "
        << circle2.radius << " is " << circle2.getArea() << endl;

    return 0;
}
```

```
The area of the circle of radius 1 is 3.14159
The area of the circle of radius 5 is 78.5397
The area of the circle of radius 100 is 31415.9
```

Outline

- Introduction
- Defining Classes for Objects
- Example: Defining Classes and Creating Objects
- Constructors
- Constructing and Using Objects
- Separating Class Definition from Implementation
- Data Field Encapsulation

Data Field Encapsulation

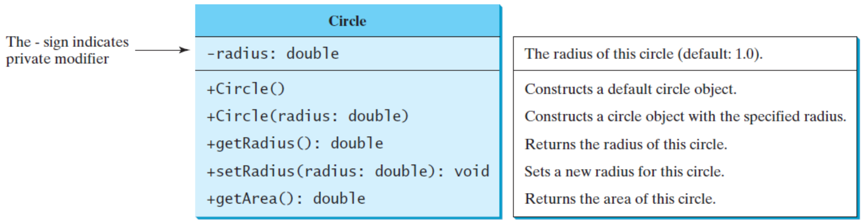
The data fields **`radius`** in the **`Circle`** class can be modified directly (e.g., **`circle1.radius = 5`**). This is not a good practice for two reasons:

1. Data may be tampered.
2. Second, it makes the class difficult to maintain and vulnerable to bugs. Suppose you want to modify the **`Circle`** class to ensure that the radius is non-negative after other programs have already used the class. You have to change not only the **`Circle`** class, but also the programs (*clients*) that use the **`Circle`** class. This is because the clients may have modified the radius directly (e.g., **`myCircle.radius = -5`**).

Accessor and Mutator

- A **`get`** function is referred to as a *getter* (or *accessor*).
- A **`get`** function has the following signature:
  `returnType getPropertyName()`
- If the **`returnType`** is **`bool`**, the **`get`** function should be defined as follows by convention:
  `bool isPropertyName()`
- A **`set`** function is referred to as a *setter* (or *mutator*).
- A **`set`** function has the following signature:
  `public void setPropertyName(dataType propertyValue)`

Example: The Circle Class with Encapsulation



Circle	
`-radius: double`	The radius of this circle (default: 1.0).
`+Circle()`	Constructs a default circle object.
`+Circle(radius: double)`	Constructs a circle object with the specified radius.
`+getRadius(): double`	Returns the radius of this circle.
`+setRadius(radius: double): void`	Sets a new radius for this circle.
`+getArea(): double`	Returns the area of this circle.

CircleWithPrivateDataFields.h

CircleWithPrivateDataFields.cpp

TestCircleWithPrivateDataFields Run

CircleWithPrivateDataFields.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle
{
public:
    Circle();
    Circle(double);
    double getArea();
    double getRadius();
    void setRadius(double);

private:
    double radius;
};

#endif
```

CircleWithPrivateDataFields.cpp

```
#include "CircleWithPrivateDataFields.h"

// Construct a default circle object
Circle::Circle()
{
    radius = 1;
}

// Construct a circle object
Circle::Circle(double newRadius)
{
    radius = newRadius;
}

// Return the area of this circle
double Circle::getArea()
{
    return radius * radius * 3.14159;
}

// Return the radius of this circle
double Circle::getRadius()
{
    return radius;
}

// Set a new radius
void Circle::setRadius(double newRadius)
{
    radius = (newRadius >= 0)
             ? newRadius : 0;
}
```

TestCircleWithPrivateDataFields.cpp

```
The area of the circle of radius 1 is 3.14159
The area of the circle of radius 5 is 78.5397
The area of the circle of radius 100 is 31415.9
```

```cpp
#include <iostream>
#include "CircleWithPrivateDataFields.h"
using namespace std;

int main()
{
    Circle circle1;
    Circle circle2(5.0);

    cout << "The area of the circle of radius "
        << circle1.getRadius() << " is " << circle1.getArea() << endl;
    cout << "The area of the circle of radius "
        << circle2.getRadius() << " is " << circle2.getArea() << endl;

    // Modify circle radius
    circle2.setRadius(100);
    cout << "The area of the circle of radius "
        << circle2.getRadius() << " is " << circle2.getArea() << endl;

    return 0;
}
```

Outline

- Introduction
- Defining Classes for Objects
- Example: Defining Classes and Creating Objects
- Constructors
- Constructing and Using Objects
- Separating Class Definition from Implementation
- Data Field Encapsulation

Chapter 11: Pointers and Dynamic Memory Management

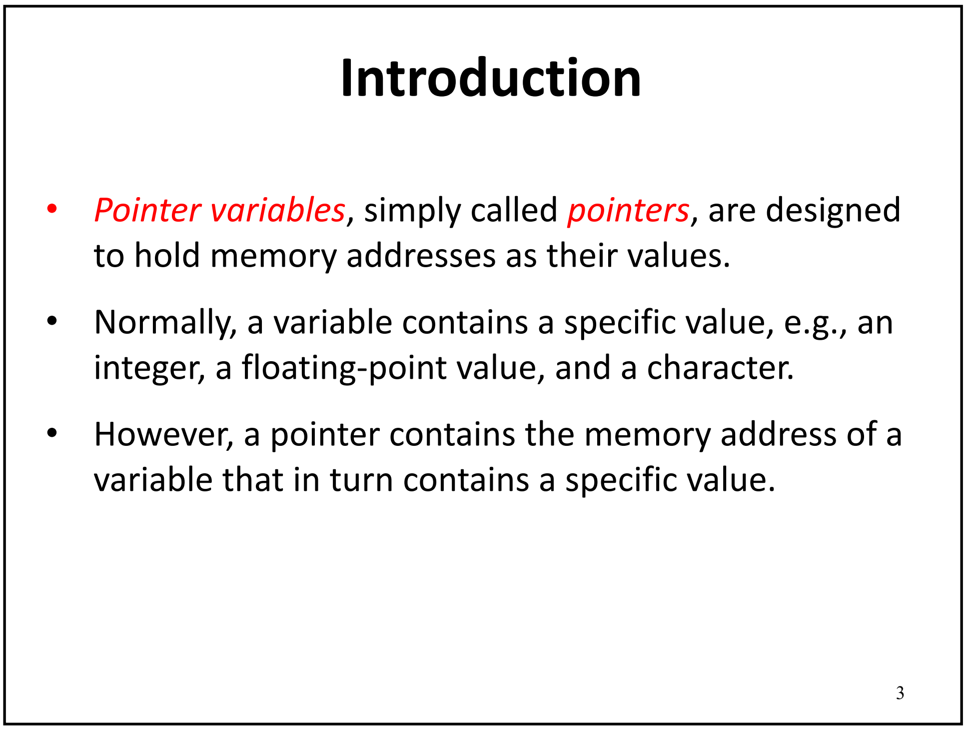
Sections 11.1–11.2, 11.5–11.7

Textbooks: Y. Daniel Liang, Introduction to Programming with C++, 3rd Edition

These slides were adapted by Prof. Gheith Abandah from the Computer Engineering Department of the University of Jordan for the Course: Computer Skills for Engineers (0907101)

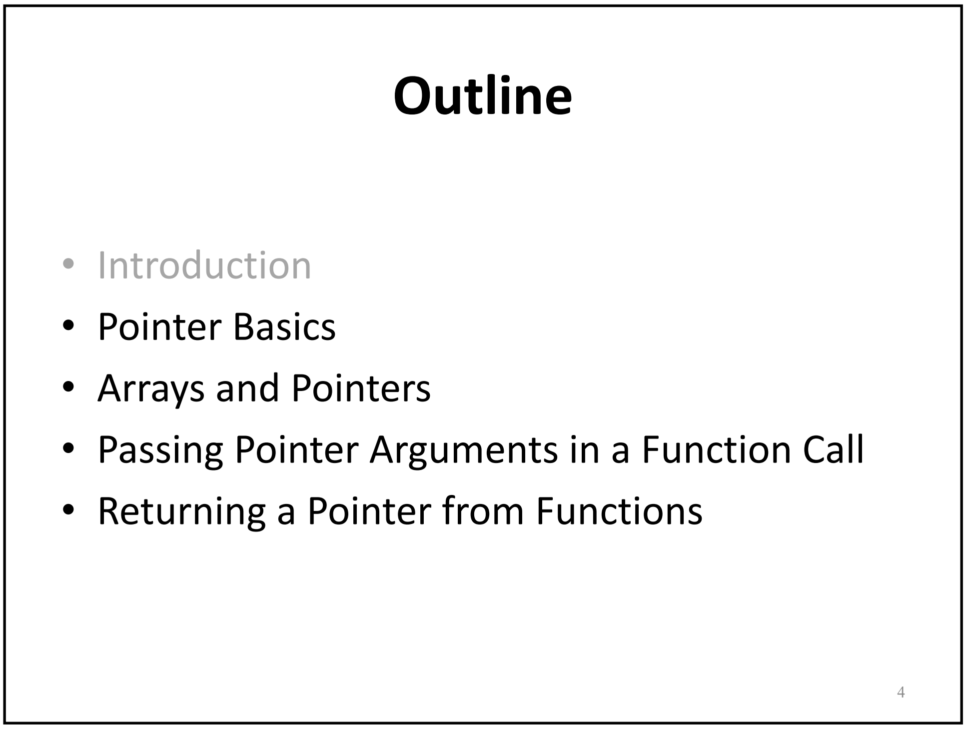
Outline

- Introduction
- Pointer Basics
- Arrays and Pointers
- Passing Pointer Arguments in a Function Call
- Returning a Pointer from Functions



Introduction

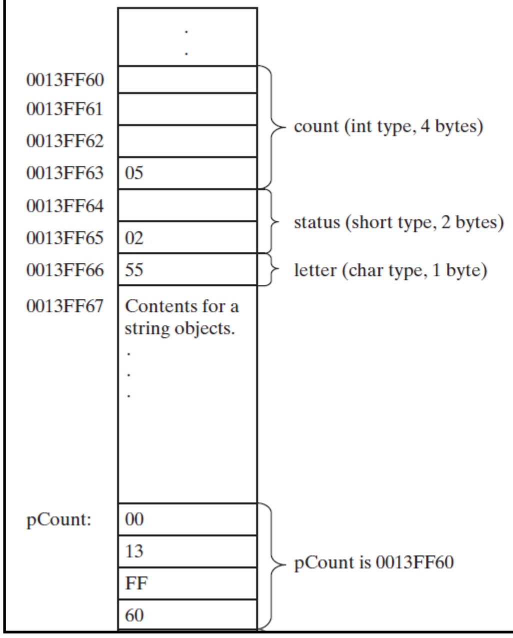
- *Pointer variables*, simply called *pointers*, are designed to hold memory addresses as their values.
- Normally, a variable contains a specific value, e.g., an integer, a floating-point value, and a character.
- However, a pointer contains the memory address of a variable that in turn contains a specific value.



Outline

- Introduction
- Pointer Basics
- Arrays and Pointers
- Passing Pointer Arguments in a Function Call
- Returning a Pointer from Functions

Pointer Basics



```
int count = 5;
short status = 2;
char letter = 'A';
string s = "ABC";

int* pCount = &count;
```

&count means the address of count

*pCount means the value pointed by pCount is assigned to v.

Declare a Pointer

- Like any other variables, pointers must be declared before they can be used. To declare a pointer, use the following syntax:
`dataType* pVarName;`

- Each variable being declared as a pointer must be preceded by an asterisk (*****). For example, the following statement declares a pointer variable named **`pCount`** that can point to an **`int`** variable.
`int* pCount;`

TestPointer.cpp

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

int main()
{
    int count = 5;
    int* pCount = &count;

    cout << "The value of count is " << count << endl;
    cout << "The address of count is " << &count << endl;
    cout << "The address of count is " << pCount << endl;
    cout << "The value of count is " << *pCount << endl;

    return 0;
}
```

```
The value of count is 5
The address of count is 00AFF980
The address of count is 00AFF980
The value of count is 5
```

Dereferencing

- Referencing a value through a pointer is called *indirection*. The syntax for referencing a value from a pointer is:
  `*pointer`

- For example, you can increase **`count`** using:

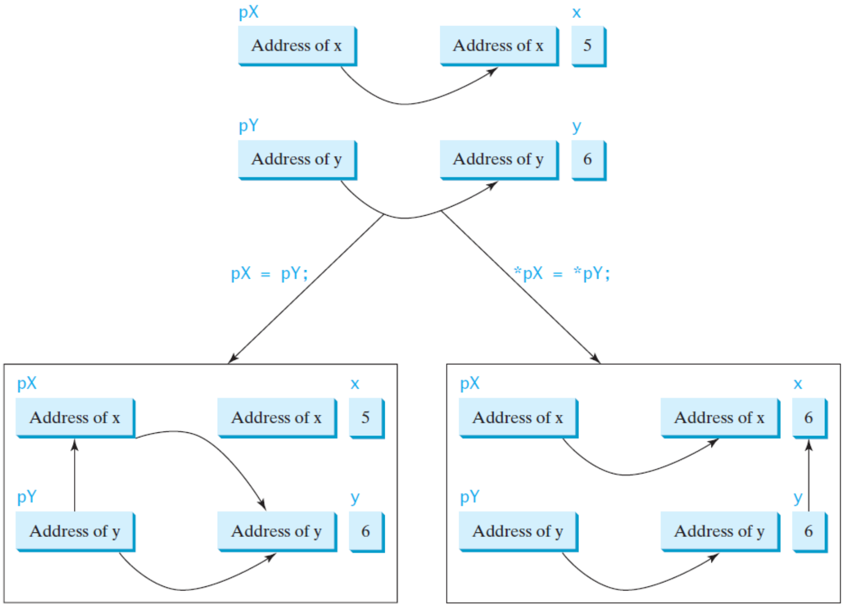
```
count++;       // direct reference
```

or

```
(*pCount)++;  // indirect reference
```

- The asterisk (*****) is the *indirection operator* or *dereference operator*.

(a) pY is assigned to pX; (b) *pY is assigned to *pX.



(a) (b)

Pointer Type

- A pointer variable is declared with a type such as **int**, **double**, etc.
- You have to assign the address of the variable of the same type.
- It is a syntax error if the type of the variable does not match the type of the pointer. For example, the following code is wrong.

```
int area = 1;
double* pArea = &area; // Wrong
```

Initializing Pointer

- Like a local variable, a local pointer is assigned an arbitrary value if you don't initialize it.
- A pointer may be initialized to **`0`**, which is a special value for a pointer to indicate that the pointer points to nothing.
- You should always initialize pointers to prevent errors.
- Dereferencing a pointer that is not initialized could cause fatal runtime error or it could accidentally modify important data.

Caution

- You can declare two variables on the same line. For example, the following line declares two **`int`** variables:

```
int i= 0, j = 1;
```

- Can you declare two pointer variables on the same line as follows?

```
int* pl, pj;
```

- No, the right way is:

```
int *pl, *pj;
```

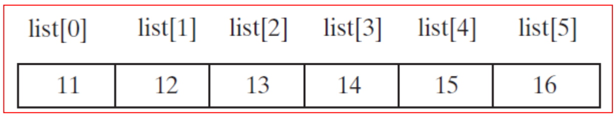
Outline

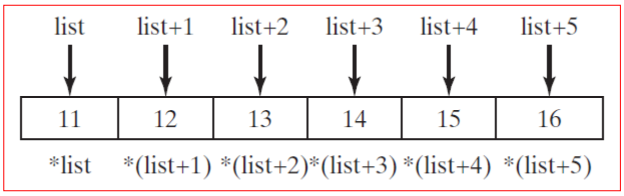
- Introduction
- Pointer Basics
- Arrays and Pointers
- Passing Pointer Arguments in a Function Call
- Returning a Pointer from Functions

Arrays and Pointers

- An array variable without a bracket and a subscript actually represents the starting address of the array.
- The array variable is essentially a pointer. Suppose you declare an array of `int` value as follows:

```
int list[6] = { 11, 12, 13, 14, 15, 16 };
```





Array Pointer

- `*(list + 1)` is different from `*list + 1`. The dereference operator (`*`) has precedence over `+`.
- So, `*list + 1` adds `1` to the value of the first element in the array, while `*(list + 1)` dereference the element at address `(list + 1)` in the array.

ArrayPointer Run

PointerWithIndex Run

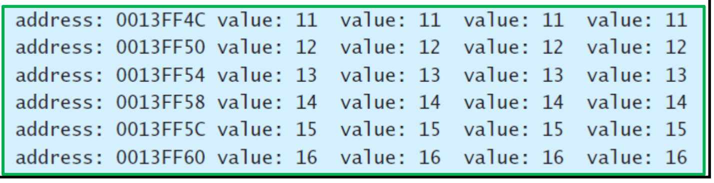
ArrayPointer.cpp

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

int main()
{
    int list[6] = { 11, 12, 13, 14, 15, 16 };

    for (int i = 0; i < 6; i++)
        cout << "address: " << (list + i) <<
        " value: " << *(list + i) << " " <<
        " value: " << list[i] << endl;

    return 0;
}
```

```
address: 0013FF4C value: 11  value: 11
address: 0013FF50 value: 12  value: 12
address: 0013FF54 value: 13  value: 13
address: 0013FF58 value: 14  value: 14
address: 0013FF5C value: 15  value: 15
address: 0013FF60 value: 16  value: 16
```

PointerWithIndex.cpp

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

int main()
{
    int list[6] = { 11, 12, 13, 14, 15, 16 };
    int* p = list;

    for (int i = 0; i < 6; i++)
        cout << "address: " << (list + i) <<
        " value: " << *(list + i) << " " <<
        " value: " << list[i] << " " <<
        " value: " << *(p + i) << " " <<
        " value: " << p[i] << endl;

    return 0;
}
```



```
address: 0013FF4C value: 11  value: 11  value: 11  value: 11
address: 0013FF50 value: 12  value: 12  value: 12  value: 12
address: 0013FF54 value: 13  value: 13  value: 13  value: 13
address: 0013FF58 value: 14  value: 14  value: 14  value: 14
address: 0013FF5C value: 15  value: 15  value: 15  value: 15
address: 0013FF60 value: 16  value: 16  value: 16  value: 16
```

Outline

- Introduction
- Pointer Basics
- Arrays and Pointers
- Passing Pointer Arguments in a Function Call
- Returning a Pointer from Functions

Passing Pointer Arguments

- A pointer argument can be passed by value or by reference. For example, you can define a function as follows:

```
void f(int* p1, int*& p2);
```

- which is equivalently to

```
typedef int* intPointer;
void f(intPointer p1, intPointer & p2);
```

- Here p1 is pass-by-value and p2 is pass-by-reference.

TestPointerArgument Run

TestPointerArgument.cpp 1/5

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

// Function definitions are here

int main()
{
    // Declare and initialize variables
    int num1 = 1;
    int num2 = 2;

    cout << "Before invoking the swap function, num1 is "
        << num1 << " and num2 is " << num2 << endl;
```

```
Before invoking the swap function, num1 is 1 and num2 is 2
```

TestPointerArgument.cpp 2/5

```
// Swap two variables using pass-by-value
void swap1(int n1, int n2)
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
```

```
// Invoke the swap function to attempt to swap two variables
swap1(num1, num2);
cout << "After invoking the swap function, num1 is “
     << num1 << " and num2 is " << num2 << endl;
```

```
After invoking the swap function, num1 is 1 and num2 is 2
```

TestPointerArgument.cpp 3/5

```
// Swap two variables using pass-by-reference
void swap2(int& n1, int& n2)
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
```

```
cout << "Before invoking the swap function, num1 is "
    << num1 << " and num2 is " << num2 << endl;

// Invoke the swap function to attempt to swap two variables
swap2(num1, num2);
cout << "After invoking the swap function, num1 is " << num1
     << " and num2 is " << num2 << endl;
```

```
Before invoking the swap function, num1 is 1 and num2 is 2
After invoking the swap function, num1 is 2 and num2 is 1
```

TestPointerArgument.cpp 4/5

```
// Pass two pointers by value
void swap3(int* p1, int* p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

```
cout << "Before invoking the swap function, num1 is "
     << num1 << " and num2 is " << num2 << endl;
// Invoke the swap function to attempt to swap two variables
swap3(&num1, &num2);
cout << "After invoking the swap function, num1 is " << num1
     << " and num2 is " << num2 << endl;
```

```
Before invoking the swap function, num1 is 2 and num2 is 1
After invoking the swap function, num1 is 1 and num2 is 2
```

TestPointerArgument.cpp 5/5

```
// Pass two pointers by reference
void swap4(int*& p1, int*& p2)
{
    int* temp = p1;
    p1 = p2;
    p2 = temp;
}
```

```
    int* p1 = &num1;
    int* p2 = &num2;
    cout << "Before invoking the swap function, p1 is "
         << p1 << " and p2 is " << p2 << endl;
    // Invoke the swap function to attempt to swap two variables
    swap4(p1, p2);
    cout << "After invoking the swap function, p1 is " << p1 <<
        " and p2 is " << p2 << endl;

    return 0;
}
```

```
Before invoking the swap function, p1 is 0028FB84 and p2 is 0028FB78
After invoking the swap function, p1 is 0028FB78 and p2 is 0028FB84
```

Array Parameter or Pointer Parameter

- An array parameter in a function can always be replaced using a pointer parameter.

`void m(int list[], int size)` can be replaced by `void m(int* list, int size)`

`void m(char c_string[])` can be replaced by `void m(char* c_string)`

`const` Parameter

If an object value does not change, you should declare it **`const`** to prevent it from being modified accidentally.

ConstParameter Run

ConstParameter.cpp

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

void printArray(const int*, const int);

int main()
{
    int list[6] = { 11, 12, 13, 14, 15, 16 };
    printArray(list, 6);

    return 0;
}

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

```
11 12 13 14 15 16
```

Outline

- Introduction
- Pointer Basics
- Arrays and Pointers
- Passing Pointer Arguments in a Function Call
- Returning a Pointer from Functions

Returning a Pointer from Functions

- You can use pointers as parameters in a function.
- A C++ function may return a pointer as well.

ReverseArrayUsingPointer Run

ReverseArrayUsingPointer.cpp 1/2

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

int* reverse(int* list, int size)
{
    for (int i = 0, j = size - 1; i < j; i++, j--)
    {
        // Swap list[i] with list[j]
        int temp = list[j];
        list[j] = list[i];
        list[i] = temp;
    }

    return list;
}
```

ReverseArrayUsingPointer.cpp 2/2

```cpp
void printArray(const int* list, int size)
{
    for (int i = 0; i < size; i++)
        cout << list[i] << " ";
}

int main()
{
    int list[] = { 1, 2, 3, 4, 5, 6 };
    int* p = reverse(list, 6);
    printArray(p, 6);

    return 0;
}
```

```
6 5 4 3 2 1
```

Outline

- Introduction
- Pointer Basics
- Arrays and Pointers
- Passing Pointer Arguments in a Function Call
- Returning a Pointer from Functions