Introduction to C++
Introduction to C++

- Syntax differences between C and C++
- A Simple C++ Example
  - C++ Input/Output
- C++ Libraries
  - C++ Header Files
- Another Simple C++ Example
  - Inline Functions
- Call by Reference in C++
- References and Reference Parameters
Introduction to C++

- Default Arguments
- Unary Scope Resolution Operator
- Function Overloading
- Function Templates
Introduction to C++

- C++ was developed by Bjarne Stroustrup at Bell Laboratories
  - Originally called “C with classes”
  - The name C++ includes C’s increment operator (++)
    - Indicate that C++ is an enhanced version of C
- C++ programs
  - Built from pieces called classes and functions.
- C++ Standard Library
  - Rich collections of existing classes and functions
Why use C++

- Many claim it is a better C because it is all of C with additions:
  - Objects {and object-oriented philosophy}
  - Inheritance
  - Polymorphism
  - Exception handling
  - Templates
```cpp
// C++ simple example
#include <iostream>  // for C++ Input and Output
int main ()
{
    int number3;
    std::cout << "Enter a number:";
    std::cin >> number3;
    int number2, sum;
    std::cout << "Enter another number:";
    std::cin >> number2;
    sum = number2 + number3;
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```
A Simple C++ Program

- C++ file names can have one of several extensions
  - Such as: `.cpp`, `.cxx` or `.C` (uppercase)

- Commenting
  - A `//` comment is a maximum of one line long.
  - A `/* */` C-style comments can be more than one line long.

- `iostream`
  - Must be included for any program that outputs data to the screen or inputs data from the keyboard using C++ style stream input/output.

- C++ requires you to specify the return type, possibly `void`, for all functions.
  - Specifying a parameter list with empty parentheses is equivalent to specifying a `void` parameter list in C.
A Simple C++ Program

- Stream manipulator **std::endl**
  - Outputs a newline.
  - Flushes the output buffer.
- The notation **std::cout** specifies that we are using a name (**cout**) that belongs to a "namespace" (**std**).
18.5 Header Files

- C++ Standard Library header files
  - Each contains a portion of the Standard Library.
    - Function prototypes for the related functions
    - Definitions of various class types and functions
    - Constants needed by those functions
  - “Instruct” the compiler on how to interface with library and user-written components.
  - Header file names ending in .h
    - Are “old-style” header files
    - Superseded by the C++ Standard Library header files
  - Use `#include` directive to include class in a program.
### C++ Standard Library header files

<table>
<thead>
<tr>
<th>C++ Standard Library header files</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;iostream&gt;</code></td>
<td>Contains function prototypes for the C++ standard input and standard output functions. This header file replaces header file <code>&lt;iostream.h&gt;</code>. This header is discussed in detail in Chapter 26, Stream Input/Output.</td>
</tr>
<tr>
<td><code>&lt;iomanip&gt;</code></td>
<td>Contains function prototypes for stream manipulators that format streams of data. This header file replaces header file <code>&lt;iomanip.h&gt;</code>. This header is used in Chapter 26, Stream Input/Output.</td>
</tr>
<tr>
<td><code>&lt;cmath&gt;</code></td>
<td>Contains function prototypes for math library functions. This header file replaces header file <code>&lt;cmath.h&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;cstdlib&gt;</code></td>
<td>Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. This header file replaces header file <code>&lt;cstdlib&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;ctime&gt;</code></td>
<td>Contains function prototypes and types for manipulating the time and date. This header file replaces header file <code>&lt;ctime.h&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;vector&gt;</code>, <code>&lt;list&gt;</code>, <code>&lt;deque&gt;</code>, <code>&lt;queue&gt;</code>, <code>&lt;stack&gt;</code>, <code>&lt;map&gt;</code>, <code>&lt;set&gt;</code>, <code>&lt;bitset&gt;</code></td>
<td>These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program’s execution.</td>
</tr>
</tbody>
</table>
18.6 Inline Functions

- Inline functions
  - Reduce function call overhead—especially for small functions.
  - Qualifier `inline` before a function's return type in the function definition
    - “Advises” the compiler to generate a copy of the function's code in place (when appropriate) to avoid a function call.
- Trade-off of inline functions
  - Multiple copies of the function code are inserted in the program (often making the program larger).
  - The compiler can ignore the `inline` qualifier and typically does so for all but the smallest functions.
```
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

// Definition of inline function cube. Definition of function appears
// before function is called, so a function prototype is not required.
// First line of function definition acts as the prototype.
inline double cube( const double side )
{
    return side * side * side; // calculate the cube of side
} // end function cube

int main()
{
    double sideValue; // stores value entered by user
```
```cpp
for ( int i = 1; i <= 3; i++ )
{
    cout << "Enter the side length of your cube: ";
    cin >> sideValue; // read value from user

    // calculate cube of sideValue and display result
    cout << "Volume of cube with side " << sideValue << " is " << cube( sideValue ) << endl;
}
return 0; // indicates successful termination
```

Enter the side length of your cube: 1.0
Volume of cube with side 1 is 1

Enter the side length of your cube: 2.3
Volume of cube with side 2.3 is 12.167

Enter the side length of your cube: 5.4
Volume of cube with side 5.4 is 157.464
### C++ keywords

*Keywords common to the C and C++ programming languages*

<table>
<thead>
<tr>
<th>auto</th>
<th>break</th>
<th>case</th>
<th>char</th>
<th>const</th>
</tr>
</thead>
<tbody>
<tr>
<td>continue</td>
<td>default</td>
<td>do</td>
<td>double</td>
<td>else</td>
</tr>
<tr>
<td>enum</td>
<td>extern</td>
<td>float</td>
<td>for</td>
<td>goto</td>
</tr>
<tr>
<td>if</td>
<td>int</td>
<td>long</td>
<td>register</td>
<td>return</td>
</tr>
<tr>
<td>short</td>
<td>signed</td>
<td>sizeof</td>
<td>static</td>
<td>struct</td>
</tr>
<tr>
<td>switch</td>
<td>typedef</td>
<td>union</td>
<td>unsigned</td>
<td>void</td>
</tr>
<tr>
<td>volatile</td>
<td>while</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### C++ keywords

#### C++-only keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and</code></td>
<td><code>and_eq</code></td>
<td><code>asm</code></td>
<td><code>bitand</code></td>
</tr>
<tr>
<td><code>bool</code></td>
<td><code>catch</code></td>
<td><code>class</code></td>
<td><code>compl</code></td>
</tr>
<tr>
<td><code>delete</code></td>
<td><code>dynamic_cast</code></td>
<td><code>explicit</code></td>
<td><code>export</code></td>
</tr>
<tr>
<td><code>friend</code></td>
<td><code>inline</code></td>
<td><code>mutable</code></td>
<td><code>namespace</code></td>
</tr>
<tr>
<td><code>not</code></td>
<td><code>not_eq</code></td>
<td><code>operator</code></td>
<td><code>or</code></td>
</tr>
<tr>
<td><code>private</code></td>
<td><code>protected</code></td>
<td><code>public</code></td>
<td><code>reinterpret_cast</code></td>
</tr>
<tr>
<td><code>template</code></td>
<td><code>this</code></td>
<td><code>throw</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>typeid</code></td>
<td><code>typename</code></td>
<td><code>using</code></td>
<td><code>virtual</code></td>
</tr>
<tr>
<td><code>xor</code></td>
<td><code>xor_eq</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
18.6 Inline Functions (Cont.)

- **using** statements help eliminate the need to repeat the namespace prefix
  - Ex: `std::`

- **for** statement's condition evaluates to either 0 (false) or nonzero (true)
  - Type `bool` represents boolean (true/false) values.
    - The two possible values of a `bool` are the keywords `true` and `false`.
      - When true and false are converted to integers, they become the values 1 and 0, respectively.
      - When non-boolean values are converted to type `bool`, non-zero values become `true`, and zero or `null` pointer values become `false`. 
18.7 References and Reference Parameters

- Reference Parameter
  - An alias for its corresponding argument in a function call.
  - & placed after the parameter type in the function prototype and function header
  - Example
    - `int &count` in a function header
      - Pronounced as “count is a reference to an int”
  - Parameter name in the called function body actually refers to the original variable in the calling function.
```cpp
// Fig. 18.5: fig18_05.cpp
// Comparing pass-by-value and pass-by-reference with references.
#include <iostream>
using std::cout;
using std::endl;

int squareByValue( int ); // function prototype (value pass)
void squareByReference( int & ); // function prototype (reference pass)

int main()
{
    int x = 2; // value to square using squareByValue
    int z = 4; // value to square using squareByReference

    // demonstrate squareByValue
    cout << "x = " << x << " before squareByValue\n";
    cout << "Value returned by squareByValue: "
        << squareByValue( x ) << endl;
    cout << "x = " << x << " after squareByValue\n"
        << endl;

    // demonstrate squareByReference
    cout << "z = " << z << " before squareByReference" << endl;
    squareByReference( z );
    cout << "z = " << z << " after squareByReference" << endl;
    return 0; // indicates successful termination
} // end main
```

Function illustrating pass-by-value

Function illustrating pass-by-reference

Variable is simply mentioned by name in both function calls
Call by Reference and Call by Value in C++

```cpp
// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue( int number )
{
    return number *= number; // caller's argument not modified
} // end function squareByValue

// squareByReference multiplies numberRef by itself and stores the result
// in the variable to which numberRef refers in the caller
void squareByReference( int &numberRef )
{
    numberRef *= numberRef; // caller's argument modified
} // end function squareByReference

x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```
18.7 References and Reference Parameters

- References
  - are used as aliases for other variables within a function.
    - All operations supposedly performed on the alias (i.e., the reference) are actually performed on the original variable.
    - An alias is simply another name for the original variable.
    - Must be initialized in their declarations.
      - It cannot be reassigned afterward.
  - Example
    - int count = 1;
      int &cRef = count;
      cRef++;  
      - Increments count through alias cRef.
References and Reference Parameters

Creating a reference as an alias to another variable in the function

Assign 7 to x through alias y

Systems Programming: Introduction to C++
// Fig. 18.7: fig18_07.cpp
// References must be initialized.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    int x = 3;
    int &y; // Error: y must be initialized
    cout << "x = " << x << endl << "y = " << y << endl;
    y = 7; // Error: y must be initialized
    cout << "x = " << x << endl << "y = " << y << endl;
    return 0; // indicates successful termination
}

Borland C++ command-line compiler error message:

Error E2304 C:\examples\ch18\Fig18_07\fig18_07.cpp 10:
    Reference variable 'y' must be initialized in function main()

Microsoft Visual C++ compiler error message:

C:\examples\ch18\Fig18_07\fig18_07.cpp(10) : error C2530: 'y' : references must be initialized

GNU C++ compiler error message:

fig18_07.cpp:10: error: 'y' declared as a reference but not initialized

© 2007 Pearson Ed - All rights reserved.
// Three ways in C++
#include <stdio.h>
int main ()
{
    int y = 8;
    int &yref = y;
    int *yptr = &y;

    printf(" y = %d\n using ref y = %d\n using pointer y = %d", y, yref, *yptr);
    return 0;
}

$ g++ -o ref ref.cpp
$ ./ref
y = 8
using ref y = 8
using pointer y = 8
References and Reference Parameters

- Returning a reference from a function
  - Functions can return references to variables.
    - Should only be used when the variable is static.
  - A Dangling reference
    - Returning a reference to an automatic variable
      - That variable no longer exists after the function ends.
18.9 Default Arguments

- Default argument
  - A default value to be passed to a parameter.
    - Used when the function call does not specify an argument for that parameter.
  - Must be the rightmost argument(s) in a function's parameter list.
  - Should be specified with the first occurrence of the function name.
    - Typically in the function prototype.
// Fig. 18.8: fig18_08.cpp
// Using default arguments.
#include <iostream>
using std::cout;
using std::endl;

// function prototype that specifies default arguments
int boxVolume( int length = 1, int width = 1, int height = 1 );

int main()
{
    // no arguments--use default values for all dimensions
    cout << "The default box volume is: " << boxVolume();

    // specify length; default width and height
    cout << "The volume of a box with length 10, width 1, height 1 is: " << boxVolume( 10 );

    // specify length and width; default height
    cout << "The volume of a box with length 10, width 5, height 1 is: " << boxVolume( 10, 5 );

    // specify all arguments
    cout << "The volume of a box with length 10, width 5, height 2 is: " << boxVolume( 10, 5, 2 );
    return 0; // indicates successful termination
} // end main

Default arguments
Calling function with no arguments
Calling function with one argument
Calling function with two arguments
Calling function with three arguments
// function boxVolume calculates the volume of a box
int boxVolume( int length, int width, int height )
{
    return length * width * height;
}

The default box volume is: 1
The volume of a box with length 10, width 1 and height 1 is: 10
The volume of a box with length 10, width 5 and height 1 is: 50
The volume of a box with length 10, width 5 and height 2 is: 100

Note that default arguments were specified in the function prototype, so they are not specified in the function header.
18.10 Unary Scope Resolution Operator

- Unary scope resolution operator (::)
  - Used to access a global variable when a local variable of the same name is in scope.
  - Cannot be used to access a local variable of the same name in an outer block.
18.10 Unary Scope Resolution Operator

```cpp
#include <iostream>

using std::cout;
using std::endl;

int number = 7; // global variable named number

int main()
{
    double number = 10.5; // local variable named number

    // display values of local and global variables
    cout << "Local double value of number = " << number
        << "\nGlobal int value of number = " << ::number << endl;

    return 0; // indicates successful termination
} // end main
```

Unary scope resolution operator used to access global variable `number`
18.11 Function Overloading

- Overloaded functions
  - Overloaded functions have
    - The same name
    - But different sets of parameters
  - Compiler selects proper function to execute based on number, types and order of arguments in the function call.
  - Commonly used to create several functions of the same name that perform similar tasks, but on different data types.
Function Overloading

```cpp
// Fig. 18.10: fig18_10.cpp
// Overloaded functions.
#include <iostream>
using std::cout;
using std::endl;

// function square for int values
int square( int x )
{
    cout << "square of integer " << x << " is ";
    return x * x;
} // end function square with int argument

// function square for double values
double square( double y )
{
    cout << "square of double " << y << " is ";
    return y * y;
} // end function square with double argument

int main()
{
    cout << square( 7 ); // calls int version
    cout << endl;
    cout << square( 7.5 ); // calls double version
    cout << endl;
    return 0; // indicates successful termination
} // end main
```

Defining a `square` function for `ints`

Defining a `square` function for `doubles`

Output confirms that the proper function was called in each case
class Listnode
{
    Listnode ()
    {
        link = NULL;
    }
    Listnode( string word)
    {
        link = NULL;
        lword = word;
    }
    ...
    Private:
        Listnode* link;
        string   lword;
};
18.12 Function Templates

- A more compact and convenient form of overloading.
  - Identical program logic and operations for each data type.

- Function template definition
  - Written by programmer once.
  - Essentially defines a whole family of overloaded functions.
  - Begins with the `template` keyword.
  - Contains a template parameter list of formal type and the parameters for the function template are enclosed in angle brackets (`<>`).
  - Formal type parameters
    - Preceded by keyword `typename` or keyword `class`.
    - Placeholders for fundamental types or user-defined types.
18.12 Function Templates

- Function-template specializations
  - Generated automatically by the compiler to handle each type of call to the function template.
  - Example for function template `max` with type parameter `T` called with `int` arguments
    - Compiler detects a `max` invocation in the program code.
    - `int` is substituted for `T` throughout the template definition.
    - This produces function-template specialization `max< int >`. 
// Fig. 18.12: maximum.h
// Definition of function template maximum

template < class T >  // or template< typename T >
T maximum( T value1, T value2, T value3 )
{
    T maxValue = value1; // assume value1 is maximum

    // determine whether value2 is greater than maxValue
    if ( value2 > maxValue )
        maxValue = value2;

    // determine whether value3 is greater than maxValue
    if ( value3 > maxValue )
        maxValue = value3;

    return maxValue;
} // end function template maximum

Using formal type parameter T in place of data type
Common Programming Error 18.11

- Not placing keyword `class` or keyword `typename` before every formal type parameter of a function template (e.g., writing `< class S, T >` instead of `< class S, class T >`) is a syntax error.
// Fig. 18.13: fig18_13.cpp
// Function template maximum test program
#include <iostream>
#include <maximum.h>

using std::cout;
using std::cin;
using std::endl;

int main()
{
    // demonstrate maximum with int values
    int int1, int2, int3;

    cout << "Input three integer values: ";
    cin >> int1 >> int2 >> int3;

    // invoke int version of maximum
    cout << "The maximum integer value is: 
        " << maximum( int1, int2, int3 ) << endl;

    // demonstrate maximum with double values
    double double1, double2, double3;

    cout << "Input three double values: ";
    cin >> double1 >> double2 >> double3;

    // invoke double version of maximum
    cout << "The maximum double value is: 
        " << maximum( double1, double2, double3 ) << endl;
}

Invoking maximum with int arguments
Function Template Example

```cpp
// invoke double version of maximum
cout << "The maximum double value is: " << maximum( double1, double2, double3 );

// demonstrate maximum with char values
char char1, char2, char3;

cout << "\n\nInput three characters: ";
cin >> char1 >> char2 >> char3;

// invoke char version of maximum
cout << "The maximum character value is: " << maximum( char1, char2, char3 ) << endl;
return 0; // indicates successful termination
```