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
• 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
// 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;
}
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.
• 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`).
15.5 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    - Superseded by the C++ Standard Library header files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 a class in a program.
<table>
<thead>
<tr>
<th>C++ Standard Library header file</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 23, 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 23.</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;stdlib.h&gt;</code>.</td>
</tr>
</tbody>
</table>
| `<ctime>`                        | Contains function prototypes and types for manipulating the time and date. This header file replaces header file `<ctime.h>`.

**Fig. 15.2**  | C++ Standard Library header files. (Part 1 of 4.)
<table>
<thead>
<tr>
<th>C++ Standard Library header file</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td><code>&lt;cctype&gt;</code></td>
<td>Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. This header file replaces header file <code>&lt;ctype.h&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;cstring&gt;</code></td>
<td>Contains function prototypes for C-style string-processing functions. This header file replaces header file <code>&lt;string.h&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;typeinfo&gt;</code></td>
<td>Contains classes for runtime type identification (determining data types at execution time).</td>
</tr>
<tr>
<td><code>&lt;exception&gt;</code>, <code>&lt;stdexcept&gt;</code></td>
<td>These header files contain classes that are used for exception handling (discussed in Chapter 24).</td>
</tr>
</tbody>
</table>

**Fig. 15.2** | C++ Standard Library header files. (Part 2 of 4.)
<table>
<thead>
<tr>
<th>C++ Standard Library header file</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;memory&gt;</code></td>
<td>Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 24.</td>
</tr>
<tr>
<td><code>&lt;fstream&gt;</code></td>
<td>Contains function prototypes for functions that perform input from files on disk and output to files on disk. This header file replaces header file <code>&lt;fstream.h&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;string&gt;</code></td>
<td>Contains the definition of class <code>string</code> from the C++ Standard Library.</td>
</tr>
<tr>
<td><code>&lt;sstream&gt;</code></td>
<td>Contains function prototypes for functions that perform input from strings in memory and output to strings in memory.</td>
</tr>
<tr>
<td><code>&lt;functional&gt;</code></td>
<td>Contains classes and functions used by C++ Standard Library algorithms.</td>
</tr>
<tr>
<td><code>&lt;iterator&gt;</code></td>
<td>Contains classes for accessing C++ Standard Library container data.</td>
</tr>
<tr>
<td><code>&lt;algorithm&gt;</code></td>
<td>Contains functions for manipulating container data.</td>
</tr>
<tr>
<td><code>&lt;cassert&gt;</code></td>
<td>Contains macros for adding diagnostics that aid program debugging. This replaces header file <code>&lt;assert.h&gt;</code> from pre-standard C++.</td>
</tr>
</tbody>
</table>

**Fig. 15.2** C++ Standard Library header files. (Part 3 of 4.)

*Copyright © Pearson, Inc. 2013. All Rights Reserved.*
### Fig. 15.2 C++ Standard Library Header Files

<table>
<thead>
<tr>
<th>C++ Standard Library header file</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;cfloat&gt;</code></td>
<td>Contains the floating-point size limits of the system. This header file replaces header file <code>&lt;float.h&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;climits&gt;</code></td>
<td>Contains the integral size limits of the system. This header file replaces header file <code>&lt;limits.h&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;cstdio&gt;</code></td>
<td>Contains function prototypes for the C-style standard input/output library functions and information used by them. This header file replaces header file <code>&lt;stdio.h&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;locale&gt;</code></td>
<td>Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, and so on).</td>
</tr>
<tr>
<td><code>&lt;limits&gt;</code></td>
<td>Contains classes for defining the numerical data type limits on each computer platform.</td>
</tr>
<tr>
<td><code>&lt;utility&gt;</code></td>
<td>Contains classes and functions that are used by many C++ Standard Library header files.</td>
</tr>
</tbody>
</table>

**Fig. 15.2** C++ Standard Library header files. (Part 4 of 4.)
15.6 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.
// Fig. 18.3: fig18_03.cpp
// Using an inline function to calculate the volume of a cube.
#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
```
### 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>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and_eq</td>
<td>asm</td>
<td>bitand</td>
<td>bitor</td>
</tr>
<tr>
<td>bool</td>
<td>catch</td>
<td>class</td>
<td>compl</td>
<td>const_cast</td>
</tr>
<tr>
<td>delete</td>
<td>dynamic_cast</td>
<td>explicit</td>
<td>export</td>
<td>false</td>
</tr>
<tr>
<td>friend</td>
<td>inline</td>
<td>mutable</td>
<td>namespace</td>
<td>new</td>
</tr>
<tr>
<td>not</td>
<td>not_eq</td>
<td>operator</td>
<td>or</td>
<td>or_eq</td>
</tr>
<tr>
<td>private</td>
<td>protected</td>
<td>public</td>
<td>reinterpret_cast</td>
<td>static_cast</td>
</tr>
<tr>
<td>template</td>
<td>this</td>
<td>throw</td>
<td>true</td>
<td>try</td>
</tr>
<tr>
<td>typeid</td>
<td>typename</td>
<td>using</td>
<td>virtual</td>
<td>wchar_t</td>
</tr>
<tr>
<td>xor</td>
<td>xor_eq</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 18.4 C++ keywords

Systems Programming

Introduction to C++
15.6 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`.

© 2007 Pearson Ed - All rights reserved.
15.7 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.
// 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";

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

Receives copy of argument in `main`

Receives reference to argument in `main`

Modifies variable in `main`
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`.
// Fig. 18.6: fig18_06.cpp
// References must be initialized.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    int x = 3;
    int &y = x; // y refers to (is an alias for) x
    cout << "x = " << x << endl << "y = " << y << endl;
    y = 7; // actually modifies x
    cout << "x = " << x << endl << "y = " << y << endl;
    return 0; // indicates successful termination
} // end main

x = 3
y = 3
x = 7
y = 7
// 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;
    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
// Three ways in C++
#include <stdio.h>
int main ()
{
    int y = 8;
    int &yref = y;
    int *yptr = &y;

    printf(" y = %d
using ref y = %d
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
• 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.
15.9 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 << "\n\nThe volume of a box with length 10, \n" << "width 1 and height 1 is: " << boxVolume(10);  

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

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

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.
15.10 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.
15.10 Unary Scope Resolution Operator

// Fig. 15.9: fig15_09.cpp
// Using the unary scope resolution operator.
#include <iostream>
using namespace std;

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;
}

// end main

Local double value of number = 10.5
Global int value of number = 7

Fig. 15.9 | Using the unary scope resolution operator.

Unary scope resolution operator used to access global variable number

Copyright © Pearson, Inc. 2013. All Rights Reserved.
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.
// Fig. 15.10: fig15_10.cpp
// Overloaded square functions.
#include <iostream>
using namespace std;

// 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

Fig. 15.10 | Overloaded square functions. (Part 1 of 2.)
Function Overloading

20  int main()
21  {
22      cout << square(7); // calls int version
23      cout << endl;
24      cout << square(7.5); // calls double version
25      cout << endl;
26  } // end main

square of integer 7 is 49
square of double 7.5 is 56.25

Fig. 15.10  |  Overloaded square functions. (Part 2 of 2.)

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;
};
15.12 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.
15.12 Function Templates

- Function-template specializations
  - Generated automatically by the compiler to handle each type of call to the function template.
  - Example for function template \texttt{max} with type parameter \texttt{T} called with \texttt{int} arguments
    - Compiler detects a \texttt{max} invocation in the program code.
    - \texttt{int} is substituted for \texttt{T} throughout the template definition.
    - This produces function-template specialization \texttt{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 maximumValue = value1; // assume value1 is maximum

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

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

    return maximumValue;
} // end function template maximum

Using formal type parameter T in place of data type
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>
using std::cout;
using std::cin;
using std::endl;

#include "maximum.h" // include definition of function template maximum

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 );

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

    cout << "\n\nInput three double values: ";
    cin >> double1 >> double2 >> double3;

    cout << "The maximum double value is: " << maximum( double1, double2, double3 );
}

Invoking maximum with int arguments
// 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

// end main

Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
Review of 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
Review of Introduction to C++

- Default Arguments
- Unary Scope Resolution Operator
- Function Overloading
- Function Templates

Note - I skipped Class template vector! (Read if interested and okay to use vectors in your programs).