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

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 philisophy}
- Inheritance
- Polymorphism
- Exception handling
- Templates

Simple C++ Example

```
// 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;
}
```

standard output stream object
stream insertion operator

stream extraction operator
standard input stream object

stream manipulator
Concatenating insertion operators

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.

Fig. 18.2 C++ Standard Library header files

C++ Standard Library header files	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and standard output functions. This header file replaces header file `<iostream.h>`. This header is discussed in detail in Chapter 26, Stream Input/Output.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header file replaces header file `<iomanip.h>`. This header is used in Chapter 26, Stream Input/Output.
`<cmath>`	Contains function prototypes for math library functions. This header file replaces header file `<math.h>`.
`<cstdlib>`	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 `<stdlib>`.
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header file replaces header file `<time.h>`.
`<vector>`, `<list>` `<deque>`, `<queue>`, `<stack>`, `<map>`, `<set>`, `<bitset>`	These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution.

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.

Another Simple C++ Program

```
1  // Fig. 18.3: fig18_03.cpp
2  // Using an inline function to calculate the volume of a cube.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  // Definition of inline function cube. Definition of function appears
9  // before function is called, so a function prototype is not required.
10 // First line of function definition acts as the prototype.
11 inline double cube( const double side )
12 {
13    return side * side * side; // calculate the cube of side
14 } // end function cube
15
16 int main()
17 {
18    double sideValue; // stores value entered by user
19
```

inline qualifier

Complete function definition so the compiler knows how to expand a **cube** function call into its inlined code.

Another Simple C++ Program

```
20    for ( int i = 1; i <= 3; i++ )
21    {
22       cout << "\nEnter the side length of your cube: ";
23       cin >> sideValue; // read value from user
24
25       // calculate cube of sideValue and display result
26       cout << "Volume of cube with side "
27          << sideValue << " is " << cube( sideValue ) << endl;
28    }
29
30    return 0; // indicates successful termination
31 } // end main
```

cube function call that could be inlined

```
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
```

Fig. 18.4 C++ keywords

C++ keywords				
Keywords common to the C and C++ programming languages				
`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`			

Fig. 18.4 C++ keywords

C++ keywords				
C++-only keywords				
`and`	`and_eq`	`asm`	`bitand`	`bitor`
`bool`	`catch`	`class`	`compl`	`const_cast`
`delete`	`dynamic_cast`	`explicit`	`export`	`false`
`friend`	`inline`	`mutable`	`namespace`	`new`
`not`	`not_eq`	`operator`	`or`	`or_eq`
`private`	`protected`	`public`	`reinterpret_cast`	`static_cast`
`template`	`this`	`throw`	`true`	`try`
`typeid`	`typename`	`using`	`virtual`	`wchar_t`
`xor`	`xor_eq`			

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.

Call by Reference and Call by Value in C++

```
1  // Fig. 18.5: fig18_05.cpp
2  // Comparing pass-by-value and pass-by-reference with references.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int squareByValue( int ); // function prototype (value pass)
8  void squareByReference( int & ); // function prototype (reference pass)
9
10 int main()
11 {
12    int x = 2; // value to square using squareByValue
13    int z = 4; // value to square using squareByReference
14
15    // demonstrate squareByValue
16    cout << "x = " << x << " before squareByValue\n";
17    cout << "Value returned by squareByValue: "
18       << squareByValue( x ) << endl;
19    cout << "x = " << x << " after squareByValue\n" <<
20
21    // demonstrate squareByReference
22    cout << "z = " << z << " before squareByReference" << endl;
23    squareByReference( z );
24    cout << "z = " << z << " after squareByReference" << endl;
25    return 0; // indicates successful termination
26 } // 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++

```
27
28 // squareByValue multiplies number by itself, stores the
29 // result in number and returns the new value of number
30 int squareByValue( int number )
31 {
32    return number *= number; // caller's argument not modified
33 } // end function squareByValue
34
35 // squareByReference multiplies numberRef by itself and stores the result
36 // in the variable to which numberRef refers in the caller
37 void squareByReference( int &numberRef )
38 {
39    numberRef *= numberRef; // caller's argument modified
40 } // end function squareByReference
```

Receives copy of argument in `main`

Receives reference to argument in `main`

Modifies variable in `main`

```
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.
      - Cannot be reassigned afterward.
  - Example
    - ```
      int count = 1;
      int &cRef = count;
      cRef++;
      ```
      - Increments count through alias cRef
      ```

References and Reference Parameters

```
1  // Fig. 18.6: fig18_06.cpp
2  // References must be initialized.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 3;
10    int &y = x; // y refers to (is an alias for) x
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7; // actually modifies x
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main
```

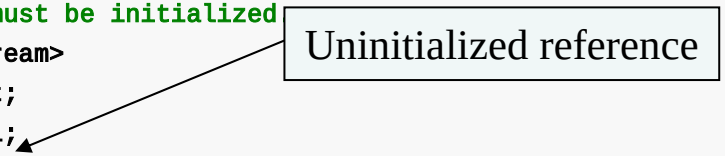
Creating a reference as an alias to another variable in the function

Assign **7** to **x** through alias **y**

```
x = 3
y = 3
x = 7
y = 7
```

References and Reference Parameters

```
1  // Fig. 18.7: fig18_07.cpp
2  // References must be initialized
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 3;
10    int &y; // Error: y must be initialized
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7;
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main
```



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
```

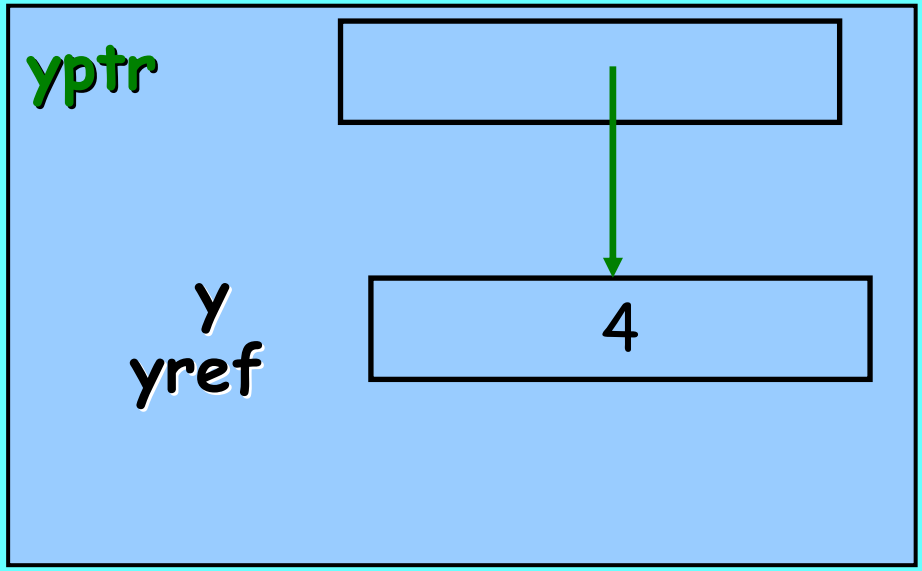
GNU C++ compiler error message:

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

References

```
// 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\n",
          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 the function prototype

Default Arguments

```
1  // Fig. 18.8: fig18_08.cpp
2  // Using default arguments.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function prototype that specifies default arguments
8  int boxVolume( int length = 1, int width = 1, int height = 1 );
9
10 int main()
11 {
12    // no arguments--use default values for all dimensions
13    cout << "The default box volume is: " << boxVolume();
14
15    // specify length; default width and height
16    cout << "\n\nThe volume of a box with length 10,\n"
17       << "width 1 and height 1 is: " << boxVolume( 10 );
18
19    // specify length and width; default height
20    cout << "\n\nThe volume of a box with length 10,\n"
21       << "width 5 and height 1 is: " << boxVolume( 10, 5 );
22
23    // specify all arguments
24    cout << "\n\nThe volume of a box with length 10,\n"
25       << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
26       << endl;
27    return 0; // indicates successful termination
28 } // end main
```


Default arguments

Calling function with no arguments

Calling function with one argument

Calling function with two arguments

Calling function with three arguments

Default Arguments

```
29
30 // function boxVolume calculates the volume of a box
31 int boxVolume( int length, int width, int height )
32 {
33    return length * width * height;
34 } // end function boxVolume
```

```
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

```
1  // Fig. 18.9: fig18_09.cpp
2  // Using the unary scope resolution operator.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int number = 7; // global variable named number
8
9  int main()
10 {
11    double number = 10.5; // local variable named number
12
13    // display values of local and global variables
14    cout << "Local double value of number = " << number
15       << "\nGlobal int value of number = " << ::number << endl;
16    return 0; // indicates successful termination
17 } // end main
```

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

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

```
1  // Fig. 18.10: fig18_10.cpp
2  // Overloaded functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function square for int values
8  int square( int x )
9  {
10    cout << "square of integer " << x << " is ";
11    return x * x;
12 } // end function square with int argument
13
14 // function square for double values
15 double square( double y )
16 {
17    cout << "square of double " << y << " is ";
18    return y * y;
19 } // end function square with double argument
20
21 int main()
22 {
23    cout << square( 7 ); // calls int version
24    cout << endl;
25    cout << square( 7.5 ); // calls double version
26    cout << endl;
27    return 0; // indicates successful termination
28 } // end main
```

Defining a **square** function for **int**s

Defining a **square** function for **double**s

Output confirms that the proper function was called in each case

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

Constructor overload

```
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 >`.

Function Template Example

```
1  // Fig. 18.12: maximum.h
2  // Definition of function template maximum.
3
4  template < class T >  // or template< typename T >
5  T maximum( T value1, T value2, T value3 )
6  {
7     T maximumValue = value1; // assume value1 is maximum
8
9     // determine whether value2 is greater than maximumValue
10    if ( value2 > maximumValue )
11       maximumValue = value2;
12
13    // determine whether value3 is greater than maximumValue
14    if ( value3 > maximumValue )
15       maximumValue = value3;
16
17    return maximumValue;
18 } // 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.

Function Template Example

```
1  // Fig. 18.13: fig18_13.cpp
2  // Function template maximum test program.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include "maximum.h" // include definition of function template maximum
9
10 int main()
11 {
12    // demonstrate maximum with int values
13    int int1, int2, int3;
14
15    cout << "Input three integer values: ";
16    cin >> int1 >> int2 >> int3;
17
18    // invoke int version of maximum
19    cout << "The maximum integer value is: "
20       << maximum( int1, int2, int3 );
21
22    // demonstrate maximum with double values
23    double double1, double2, double3;
24
25    cout << "\n\nInput three double values: ";
26    cin >> double1 >> double2 >> double3;
27
```

Invoking **maximum** with **int** arguments

Function Template Example

```
28    // invoke double version of maximum
29    cout << "The maximum double value is: "
30       << maximum( double1, double2, double3 );
31
32    // demonstrate maximum with char values
33    char char1, char2, char3;
34
35    cout << "\n\nInput three characters: ";
36    cin >> char1 >> char2 >> char3;
37
38    // invoke char version of maximum
39    cout << "The maximum character value is: "
40       << maximum( char1, char2, char3 ) << endl;
41    return 0; // indicates successful termination
42 } // end main
```

Invoking **maximum** with **double** arguments

Invoking **maximum** with **char** arguments

```
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
```