C++
Introduction and Overview
Raymond Klefstad, Ph.D.

Object-Oriented Design

- a design is a plan for program structure
- (Would you build a house without a blueprint?)
- start with a problem statement
- sketch out your design
  - identify the objects
  - break them down into components
  - identify key operations
  - EG Automobile, Stereo System, Human
- OO design is so popular because it is natural
- and it really works!
- C++ facilitates object-oriented programming

C++

- an general purpose progrogramming language
- major features include:
  - classes (for defining objects)
  - method and operator overloading
  - templates
  - exceptions
  - inheritance with polymorphism
  - a large standard library of classes
- start with an overview of basic language features

---

iostream

- Simple Input and Output: The iostream Library

```
 #include <iostream>
using namespace std;
int main()
{
  int i = 40;
  cout << "I is " << i << endl;
  return 0;
}
```

- #include brings in declarations
- using namespace std; makes cout and endl directly visible
- linker brings in definitions of cout and << operators
- iostream defines << and >> for all primitive data types

Classes

- Define state and behavior for a set of similar objects
- Similar to a blueprint for a house
- Each class has two parts:
  - Interface
    - part accessible by owners of an object
  - Implementation
    - internal part that makes the object work

Class Definition

- EG Circle

```
class Circle
{
private: // the Implementation
  int radius;
  int centerX;
  int centerY;
  const float PI = 3.14159;
public: // the Interface
  Circle(int newX, int newY, int newRadius)
  {
    centerX = newX;
    centerY = newY;
    radius = newRadius;
  }
  float area()
  {
    return PI * radius * radius;
  }
};
```

Objects (Class Use)

- EG main C++ program using class Circle

```
int main()
{
  Circle c(0,0,1);
  cout << "The area of Circle c is " << c.area() << endl;
  return 0;
}
```

- c is an object of type Circle
- c has a centerX, a centerY, and a radius.
- we can print out the area of object c

Object Life-times

- Objects have a life

- they are born (Allocation then Construction)
- they live and interact with other objects
- they die (Destruction then Deallocation)

- EG

```
{
  Circle c(0,0,1); // c is born
  ... // c is alive - we can ask it's area
} // c dies - no more Circle c
```

Object Allocation

- three kinds of allocation (in C++)
  - static - occurs before program execution
  - stack - occurs when objects are declared in a function
  - dynamic - occurs whenever we call 'new'
- C++ memory model:
  - Program Code, Static Data, Stack Data, Heap Data
- we first use stack allocation
- we will use dynamic allocation (from heap) later

Scope

- all definitions have a scope
  - scope refers to visibility/accessibility
- EG

```
{
  Circle c( 0, 0, 1 );
  // Circle c is  in scope
  {
    int c = 0; // Circle c is  hidden by int c
    // int c is  in scope
  }
  // Circle c is back  in scope
  // int c is  out of scope
}
// both the objects named c are  out of scope
```

Functions

- similar to a mathematical function
- has 4 parts:
  - a name
  - a list of formal parameters
  - a return type
  - a compound statement to compute and return the result value
- EG

```
float square( float x )
{
```

```
  return x * x;
}
```

The *main* function

- Every C++ program must have one function named *main*
- when you run your program, main is called

```
 #include <iostream> // allows use of >> and << for I/O
using namespace std;
int main() //  int is the  exit status for main
{
  cout << "Hello Everyone!\n";
  return 0; // 0 means program terminated  ok
}
```

Declaring Functions

- called a *function declaration* or *function prototype*

```
double average( double x, double y );
double toFarenheight( double centegradeTemp );
double toCentegrade( double farenheightTemp );
```

Defining Functions

- called a *function definition*

```
double average( double x, double y )
{
  return ( x + y ) / 2.0;
}
double toFarenheight( double centegradeTemp )
{
  return 9.0 * centegradeTemp / 5.0 + 32.0;
}
double toCentegrade( double farenheightTemp )
{
  return 5.0 * ( farenheightTemp - 32.0 ) / 9.0;
}
```

- a function definition also acts as a function declaration

Using Functions

- called a *function call*

```
int main()
{
  double x = 10.5;
  double y = 32.6;
  double z = average( x, y );
  double centegradeTemp = 22.3;
  double farenheightTemp = toFarenheight( centegradeTemp
```

```
);
  cout << toFarenheight( centegradeTemp + 2.0 ) << endl;
  cout << average( toFarenheight( 29.7 ),
          toFarenheight( centegradeTemp ) );
  return 0;
}
```

- generally, a function must be *declared* before it is *called*

Function Parameters

- *formals* are those given in the definition
- x and y are the *formal parameters* for average

```
double average( double x, double y )
{
  return ( x + y ) / 2.0;
}
```

- *actuals* are those supplied in a call
- 1.0 and 2.0*x are the *actual parameters* for this call to average

```
double x = 5.0;
double z = average( 1.0, 2.0*x );
```

Default Values for Parameters

- formals may have default values

```
double toCentegrade( double farenheightTemp = 32.0 )
{
  return 5.0 * ( farenheightTemp - 32.0 ) / 9.0;
}
```

- if an actual parameter is omitted, formal takes on the default value

```
double t = toCentegrade(); // gives the Centegrade for
32.0
cout << toCentegrade( 6.0 ); // overrides the default with
6.0
```

The Return Statement

- causes a value to be returned to the function caller immediately
- type of return expression must match declared return type
- EG

```
float square(float x)
{
  float result = x * x;
  cout << "Hello there!\n";
  return result;
  cout << "Hello there again!\n"; // never printed
}
```

```
int main()
{
  cout << square( 2.0 );
  cout << square( square( 2.0 ) );
  return 0;
}
```

Primitive Data Types

- foundational types are *built-in* or *pre-defined*
- other data types are defined with classes
- every data type has a size (in bytes) and a range of values
- includes integral, floating point, character and character string types

---

The Integral Types

- correspond to whole integers
- kinds of integral types:
  - char, 1 byte, -128 through 127
  - short, 2 bytes, -32768 through 32767
  - int, machine word size (either short or long)
  - long, 4 bytes, -2147483648 through 2147483647
  - also unsigned versions of all
- example literals

```
0
1
-1
-1234567
11 // decimal 11
011 // octal 9
0x11 // hex 17
```

The Floating Point Types

- corresponds to floating point real numbers
- three kinds of floating point types:
  - float (usually 4 bytes)
  - double (usually 8 bytes)
  - long double (usually 8 bytes)
- example literals

```
1.0
-3.000001e-10
30.01E40
```

The Character Type

- represents an ASCII character code
- requires one byte
- range of values

```
0 to 255
```

- example literals

```
'\0' null character 0
'\n' newline (or linefeed) 10
'\r' return 13
'\t' tab 9
' ' space 32
'0' 48
'A' 65
'a' 97
```

The Character String Type

- represents a sequence of characters
- usually declared as a char *

```
char * s = "Hello";
```

- example literals

```
char * t = "Hello world!";
cout << "This is another character string.\n";
cout << t << endl; // prints: Hello World!
```

- we will learn all about character strings later

The class string

- a standard library class wrapper
- also a sequence of characters
- may use iterators to traverse

```
int main()
{
  string s = "hello";
  cin >> s;
  cout << s;
  for ( int i = 0; i < s.size(); ++i )
    cout << s[i];
  for ( string::iterator i = s.begin(); i != s.end(); ++i
)
    cout << *i;
}
```

Variables

- variables must be declared before they are used
- variable declaration with initialization

```
int numberOfStudents = 30;
int automobileVelocity = 0;
```

- assignment operator changes the value in a variable

```
numberOfStudents = 0; // got rid of all the students
automobileVelocity += 20; // accelerated the auto
```

Symbolic Constants

- constants have a fixed value

```
const double PI = 3.1415926536;
const char newline = '\n';
```

- constants may not be assigned

```
PI = 3.0; // compiler will give a warning message
```

- it is good style to name literal constants

```
return 3.14159*r*r;
return PI*r*r;
```

Simple Operators

- Numeric Operators
  - +, -, *, /, %, unary +,-
- Assignment Operators (modifies state of object)
  - =, +=, -=, *=, /=, %=, ++, --

    ```
    automobileVelocity = ( acceleration * time * time ) /
    2.0;
    ```

Using Operators Properly

- precedence
- associativity
- parenthesis may over-ride
- memorize these operators, precedence, and associativity
- from highest to lowest
  - ++ -- (unary) + -
  - * / %
  - + -
  - = += -= *= /= %=

Statements

- Declaration Statements
  - introduces a new object
  - object is in scope to the end of enclosing block

    ```
    int main()
    {
      double i = 2 * PI;
      cout << i << endl;
      return 0;
    }
    ```

Expression Statements

- any expression may be used as a statement
- the value is discarded

```
int main()
{
  const double PI = 3.14159;
  double i = 2.0 * PI;
  cout << i << endl;
  i = i / 2.0;
  square( 2.0 ); // be careful of this mistake
  cout << i << endl;
  return 0;
}
```

Other Statements

- if, switch, while, for, return, break are similar to C
- but you can declare local variables in loops

```
int main()
{
  for ( int i=0; i<10; ++i )
    cout << i;
  for ( int i=10; i>=0; --i )
    cout << i;
  return 0;
}
```

Scoping Rules

- a function's parameters are *in scope* only within the function body
- we say they are *local* to the function body
- any variable declared inside a function is also local to the function

```
int f( int a, int j ) // a and j are now in scope
{
  int i = 10; // i is now in scope
  {
    int j = i; // new j is now in scope and hides
parameter j
    int i = 30; // this new i is in scope and hides outer
i
    cout << i * j << endl; // refers to inner i and j
  } // inner j and i are now out of scope
  cout << i << endl;
  return a + j + i; // refers to the parameters and outer
i
} // a, j, i are now out of scope
int main()
```

```
{
  int a = 10;  // a is in scope
  int i = 20;  // i is in scope
  cout << f( i, a ); // calls f with actual values 20 and
10
  return 0;
} // a and i are now out of scope
```

Reference Parameters

- sometimes we may want a function to modify the value in a parameter
- reference parameters allow us to do this
- the '&' means the parameter is *passed by reference*
- EG

```
void increment( int & x )
{
  x = x + 1;
}
int main()
{
  int z = 10;
  increment( z );
  increment( z );
  cout << z;
  return 0;
}
```

- EG swap

Output

- output is done via << operator
- called an 'inserter'
- *cout* stands for the standard output (the console)
- EG

```
int main()
{
  cout << "Hello";
  cout << 10 * 10;
  cout << 'A';
  cout << 3.14159;
  ...
}
```

Input

- input is done via the >> operator
- the >> operator uses reference parameter to modify its value
- called an 'extractor'

- *cin* stands for standard input (the console)
- it waits for a value to be entered (may require a return/enter)
- EG

```
int main()
{
  int i;
  float f;
  char c;
  cin >> i;  // reads a string of digits as an integer
  cin >> f; // reads a string of digits, decimal as a real
number
  cin >> c; // reads a single character
  ...
}
```

Files and Libraries

- .h files include class and function declarations
- .cpp files include definitions of functions and class member function
- .cpp files typically #include the .h files they use
- a .h file must be included where its declarations are used
- each .cpp file is compiled to object module
- object modules are linked together to form a program

Templates

- classes and functions may be templates
- template parameters are types and constants
- allows definition of reusable classes and functions

Definition of the template

- done with template <>

```
template
  <typename Element>
class List
{
private:
  ListNode <Element> head; // another template I would
define
public:
  List();
  Element & operator []( int i );
  int length();
  bool isMember( Element e );
  void insert( Element e );
  ~List();
};
```

Use of the template

- done with name<>

```
int main()
{
  List<int> l;
  for ( int i = 0; i < 10; ++i )
    l.insert( i );
  if ( l.isMember( 5 ) )
    cout << "5 is there\n";
  for ( int i = 0; i < l.length(); ++i )
    cout << l[i] << endl;
  return 0;
}
```

Exceptions

- useful for making code more robust
- typically for handling errors and boundary conditions
- exceptions are thrown and must be caught
- exceptions are sent to appropriate catch by type matching

Catching and Throwing Exceptions

- done with try/catch and throw

```
struct Exception
{
  char * message;
  Exception( char *s ) : message(s) {}
};
void f( int i )
{
  cout << "Entering f\n";
  if ( i % 2 == 0 ) // i is even
    throw Exception("I is even");
  else
    throw Exception("I is odd");
  cout << "Leaving f\n";
}
void testExceptions( int value )
{
  cout << "Entering testExceptions\n";
  f( value );
  cout << "Leaving testExceptions\n";
}
int main()
{
  for (int i=0; i<10; ++i )
    try
```

```
    {
      cout << "going to call testExceptions\n";
      testExceptions( i );
      cout << "back from call to testExceptions\n";
    }
    catch ( Exception e )
    {
      cout << e.message << endl;
    }
}
```

Exception Example Output

- sample output

```
going to call testExceptions
Entering testExceptions
Entering f
I is even
going to call testExceptions
Entering testExceptions
Entering f
I is odd
going to call testExceptions
Entering testExceptions
Entering f
I is even
going to call testExceptions
Entering testExceptions
Entering f
I is odd
```