C++
Raymond Klefstad, Ph.D.
Classes and Enumerations
Selection Statements
Iteration and Simple Arrays

Review

Functions Parameters Return Statement Primitive Datatypes Variables and Constants Reference Parameters

Outline

Classes Member data Member functions Class instances (objects) Defining the << operator Enumerations

Introduction

- given a programming problem
- first we design our set of objects and identify some likely operations
- next we define classes to describe these objects
- a class definition introduced a new data type
- EG class CDPlayer, class Automobile, class Engine

Example Class

- the class of all complex numbers

```
class Complex
{
public:
  float re, im;
  Complex( float newRe = 0.0, float newIm = 0.0 )
    : re( newRe ), im( newIm )
  {
  }
  Complex add( Complex c )
  {
    return Complex( re + c.re, im + c.im );
  }
  void print( ostream & out )
  {
    out << "(" << re << "+" << im << "i)";
  }
  ~Complex()
  {
  }
};
```

Class Members

- definitions made inside a class are called *class members*
- *member data* is data contained in each object of this class type
  - they may be constant or variable

    ```
    float re, im;
    ```

- a *member function* operates on an object of this class type

  ```
  int main()
  {
    Complex c( 1.0 , 2.5 );
    c.print( cout );
  }
  ```

The Hidden 'this' Parameter

- every member function has a hidden extra parameter
- the parameter is named 'this'
- 'this' is an object whose type is the one defined by this class
- members of 'this' are directly visible (in scope) inside the member function

Accessing Members From Inside the Class

- member functions may access members of 'this' directly
- EG

  ```
  class Complex
  {
  public:
    float re, im;
    void print( ostream & out )
    {
      out << "(";
      out << re;  // re of this
      out << "+";
      out << im;  // im of this
      out << "i)";
    }
    ...
  };
  ```

Defining a Constructor

- a constructor must build and initialize our objects
- it is called automatically just after allocation of 'this' object
- the constructor's 'init list' allows construction of data members

  ```
  class Complex
  {
  public:
    float re, im;
  ```

```
  Complex( float newRe = 0.0, float newIm = 0.0 )
    : re( newRe ), im( newIm )
  {
    cout << "Complex number ";
    print( cout );
    cout << " is born.\n";
  }
};
```

Defining Member Functions

- member functions are similar to regular functions
- members of 'this' are directly visible
- they may have additional parameters as declared
- they may access private and public class members

```
class Complex
{
public:
  float re, im;
  Complex add( Complex c )
  {
    return Complex( re + c.re, im + c.im );
  }
};
```

Defining a Destructor

- a destructor must clean-up after our objects
- it is called automatically just before deallocation of *this* object
- EG

```
class Complex
{
public:
  float re, im;
  ~Complex()
  {
    cout << "Complex number: ";
    print( cout );
    cout << " has died.\n";
  }
};
```

Defining Objects Outside the Class

- Each instance has its own data members
- public members may be accessed using the dot operator
- EG Complex

```
 #include "Complex.h"
int main()
```

```
{
  Complex c1( 1.5, 5.3); /// c1 is born
  Complex c2( 2.5, 2.7 ); /// c2 is born
  c1.print( cout );
  c2.print( cout );
  {
    Complex result; /// What happens here?
    result.print( cout );
    result = c1.add( c2 );  /// and here??
    result.print( cout );
  } /// and here???
  c1 = Complex( 2.0, 3.0 ); /// a literal Complex number
  c1.print( cout );
  return 0;
} /// and here????
```

Defining operator <<

- C++ allows us to define a << operator for our new type
- operator << can not be a member function!
- EG

```
ostream & operator << ( ostream & out, Complex c )
{
  c.print( out );
  return out;
}
```

- EG

```
int main()
{
  Complex mySink( 3.4, 2.2 );
  cout << "Sink at location: " << mySink << endl;
  ...
}
```

Public or Private

- public: section defines the class interface (controls)
    - these members are accessible to everyone
- private: section defines the class implementation (internals)
    - these members are accessible only to class member functions
- simple rule for now:
    - make data members private
    - make member functions (and constructors/destructors) public

Declaring a Class

- classes are typically declared in a .h file
- class declarations are analogous to function prototypes

- EG Complex.h

```
class Complex
{
private:
  float re, im;
public:
  Complex( float newRe, float newIm );
  Complex add( Complex c );
  void print( ostream & out );
  ...
};
ostream & operator << ( ostream & out, Complex c );
```

Defining Class Member Functions

- We prefer to define class member functions in the .cpp file using the scope qualifier :: to qualify their name
- scope rules are the same as when you define them inside the class declaration
- EG Complex.cpp

```
 #include "Complex.h"
Complex :: Complex( float newRe, float newIm )
  : re( newRe ), im( newIm )
{
}
Complex Complex :: add( Complex c )
{
  return Complex( re + c.re, im + c.im );
}
void Complex :: print( ostream & out )
{
  out << "(" << re "+" << im << "i)";
}
...
```

Enumerations

- a shorthand for defining a list of constants
- instead of

```
const int MON=0, TUE=1, WED=2, THU=3, FRI=4,
  SAT=5, SUN=6;
```

- enum allows more consice and less error prone definition

```
enum Day {MON, TUE, WED, THU, FRI, SAT, SUN};
```

- Day is a new type with values MON, ..., SUN

```
Day d = MON;
```

Complete Example: Integer

- EG Integer.h

```
 #include <iostream.h>
class Integer
{
private:
  int value;
public:
  Integer( int i = 0 );
  void print( ostream & out );
  int getValue();
  void setValue( int i );
  ~Integer();
};
ostream & operator << ( ostream & out, Integer i );
```

- EG Integer.cpp

```
 #include "Integer.h"
Integer :: Integer( int i )
  : value(i)
{
  cout << "Integer " << value << " was just born.\n";
}
void Integer :: print( ostream & out )
{
  out << value;
}
int Integer :: getValue()
{
  return value;
}
void Integer :: setValue( int i )
{
  value = i;
}
Integer :: ~Integer()
{
  cout << "Integer " << value << " has just died.\n";
}
ostream & operator << ( ostream & out, Integer i )
{
  i.print( out );
  return out;
}
```

- EG main.cpp

```
 #include "Integer.h"
int main()
```

```
{
  Integer i = 20;
  {
    Integer j = 30;
    {
      Integer k(40);
      k.setValue( i.getValue() + j.getValue() );
      cout << "k is " << k << endl;
    }
    cout << "j is " << j << endl;
  }
  cout << "i is " << i << endl;
  return 0;
}
```

-

EG console output

```
Integer 20 was just born.
Integer 30 was just born.
Integer 40 was just born.
k is 50
Integer 50 has just died.
j is 30
Integer 30 has just died.
i is 20
Integer 20 has just died.
```

Boolean expressions

- they return 0 (false) or 1 (true)
- in general, non-zero is also considered true
- boolean expressions consist of
  - constants or variables
  - unary or binary expressions involving boolean expressions
  - EG

    ```
    !a && b < c || d == 0
    ```

Primitive type bool

- predefined type "bool" is short for "boolean"
- has values false and true
- useful for conditions

```
bool isEqual(int x, int y)
{
  return x == y;
}
int main()
{
  bool b = true;
```

```
  b = isEqual(3, 4);
  b = false;
  return 0;
}
```

Equality Operators

- a == b
  - returns true iff a and b contain the same value
- a != b
  - returns true iff a and b contain different values
- no default == or != for classes

Relational Operators

- a < b
- a > b
- a <= b
- a >= b
- no default relational operators for classes

Logical Operators (short circuited)

- a && b

```
bool cond = divisor > 0 && numerator / divisor > 0.1;
```

- a || b

```
bool notADigit = c < '0' || c > '9';
```

- !a

```
bool isAChild = age < 18;
bool isAdult = !isAChild;
```

Precedence rules

- from highest to lowest (see appendix B page 417 for table)
  - ! ++ -- (unary) + -
  - * / %
  - + -
  - < <= > >=
  - == !=
  - &&
  - ||
  - = += -= *= /= %=

The if Statement

- conditional execution of a statement

```
int main()
{
  int a = 1;
  int b = 2;
  if ( a < b )
    cout << "a < b\n";
  else if ( a > b )
    cout << "a > b\n";
  else
    cout << "a == b\n";
  if ( a > 0 )
    cout << "a is positive\n";
}
```

Nesting if Statements

- else's match nearest unmatched if
- indentation is not considered (be careful!)

```
int maxOfThree( int a, int b, int c )
{
  if ( a < b )
    if ( b < c )
      return c;
    else
      return b;
  else if ( a < c )
    return c;
  else
    return a;
}
```

if Statement Caveats

- a syntax error that changes meaning of if statement

```
if ( e ); // extra semicoln means empty statements
  cout << "Hello"; // prints "Hello" even if e is false
```

- an awkward use of if statement

```
if ( e )
  ; // nothing
else
  cout << "Hello";
```

- natural, but very harmful, mistake

```
int a = 0;
if ( a = 0 )
  cout << "Hello"; // never happens! Why?
```

- another awkward use of if statement

```
if ( a < b )
  return true;
```

```
else
  return false;
```

- better to say
  - return a < b;

The switch Statement

- for selecting among a set of integral values

```
int main()
{
  int i = getIntegerFromUser();
  cout << "Some stuff here\n";
  switch ( i )
  {
    case 1:
    case 3:
    case 5:
    case 7:
    case 9:
      cout << i << " is odd\n";
      break;
    case 0:
    case 2:
    case 4:
    case 6:
    case 8:
      cout << i << " is even\n";
      break;
    default:
      cout << i << " isn't in range 0 to 9\n";
      break;
  }
  cout << "Some more stuff here\n";
}
```

Another switch Statement Example

- break isn't required with return

```
bool isDigit( char c )
{
  switch ( c )
  {
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
```

```
    case '7':
    case '8':
    case '9':
      return true;
    default:
      return false;
  }
}
```

switch Statement Caveats

- forgetting the break!

```
int main()
{
  int score = getScoreFromUser();
  char grade = computeStudentsGrade( score );
  switch ( grade )
  {
    case 'A':
      cout << "Excellent!\n";
    case 'B':
      cout << "Good.\n";
    case 'C':
      cout << "Fair - just passed.\n";
    case 'D':
      cout << "Poor - See you next quarter.\n";
    case 'F':
      cout << "Failed - off to OCC.\n";
    defaut:
      cout "Invalid Grade " << grade << endl;
  }
}
```

Another switch Statement Caveat

- There are no ranges for integral values

```
bool isDigit(char c)
{
  switch ( c )
  {
    case '0'-'9': // will subtract '9' from '0'
      return true;
    default:
      return false;
  }
}
```

- Must be listed separately

```
bool isDigit(char c)
```

```
{
  switch ( c )
  {
    case '0':
    case '1':
    case '2':
      /// do something here
    default:
      return false;
  }
}
```

assert.h

- allows statement of assumptions
- if the assertion is false, the program aborts with an error message
- good programming practice: state your assumptions with assert
- should be able to delete them without affecting the program execution

```
 #include <assert.h>
class Coins
{
  Coins(int q, int n, int d, int p)
  {
    assert(q >= 0 && n >= 0 && d >= 0 && p >= 0);
  }
  Coins extractChange(int amount)
  {
    assert( amount > 0 );
    ...
  }
  ...
};
```

Simple Menu User Interface

- a simple user interface will do the following:
  - present a menu
  - read a character command from the user
  - evaluate the command appropriately

Menu Presentation

- EG

```
void presentMenu()
{
  cout << "\n\n   * * * * * * * * * * * * * * * * * * * *
* * *\n"
    << "   *                PIGGY BANK MENU
*\n"
```

```
    << "   *
*\n"
    << "   *     OPTION                          ENTER  *\n"
    << "   *
*\n"
    << "   *   Show Balance (in $)            B or b   *\n"
    << "   *   Show Coins in the Bank          C or c   *\n"
    << "   *   Deposit Coins                    D or d
*\n"
    << "   *   Get Coins for Purchase          P or p   *\n"
    << "   *
*\n"
    << "   *   Quit                              Q or q
*\n"
    << "   *
*\n"
    << "   * * * * * * * * * * * * * * * * * * * * * *
*\n\n";
}
```

Reading the Command Character

- the prompt parameter allows us to specify a message for the user
- EG

```
char getChoice( char * prompt )
{
  char ch;
  cout << prompt << " (followed by enter): ";
  cin >> ch;
  return ch;
}
```

Evaluation of the command

- EG

```
void evaluateCommand( Coins & piggyBank, char choice )
{
  switch ( choice )
  {
    case 'B': case 'b':
      cout << "Balance is $ " << piggyBank.total() <<
endl;
      break;
    case 'C': case 'c':
      cout << piggyBank << endl;
      break;
    case 'D': case 'd':
      cout << "How many quarters?  ";
      ...
```

```
        break;
      case 'P': case 'p':
      ...
      case 'Q': case 'q':
        cout << "Done with Piggy Bank.\n\n";
        exit(0); /// causes the program to terminate
      default:
        cout << "Invalid command " << choice << endl;
        break;
    }
  }
```

Putting it all together

- EG

```
 #include <iostream.h>
 #include "Coins.h"
int main()
{
  Coins piggyBank;
  while ( true )
  {
    presentMenu();
    char command = getChoice("Enter a command character");
    evaluateCommand( piggyBank, command );
  }
}
```

The Concept of Iteration

- also called `looping'
- allows repeating a similar action several times
- the *break* statement will exit any loop
- the *return* statement will also exit the loop

The for Statement

- the most common loop statement
- Natural for initializing, testing, then advancing
- abstract examples

```
for ( each student, s, in this class )
  assignGradeTo( s );
for ( each day, d, of the quarter )
  studyHardOnDay( d );
for (each station, s, on the radio tuner )
{
  radio.tuneTo( s );
  if ( youLikeTheSong( radio.listen() )
    break; /// terminates this for loop
}
for ( each integer, i, in the range 0 to 9 )
  cout << i << endl;
```

- real examples

```
// print out numbers 0 through 9
for ( int i = 0; i < 10; ++i )
  cout << i << endl;
// read 10 integers from the input and print the sum
int main()
{
  int valueRead = 0;
  int sumTotal = 0;
  for ( int i = 0; i < 10; i++ )
  {
    cin >> valueRead;
    sumTotal += valueRead;
  }
  cout << "The total is: " << sumTotal << endl;
}
```

The while Statement

- Natural for testing BEFORE doing an action that involves repetition
- EG

```
while ( coolade.isTooSour() )
  coolade.addATeaspoonOfSugar();
while ( bathtub.waterIsTooCold() )
  bathtub.addAGallonOfHotWater();
while ( ! student.understandTheHomeworkAssignment() )
{
  student.readTheHomeworkHandout();
  student.askQuestions( TA );
}
while ( student.isStillAwake() )
  student.study();
```

The do-while Statement

- Natural for doing an action then testing for completion before repetition
- EG

```
do
  car.turnIgnition();
while (! car.started() );
do
  phone.pressANumber();
while (! phone.haveAConnection() );
do
{
  student.readTheHomeworkHandout();
  student.askSomeQuestions(TA);
} while ( !student.understands( materialForWeek( w ) ) );
do
  person.eat( pintOfIceCream );
while ( !person.sick() );
```

Nested loops

- EG // print out a calendar

```
const int JAN = 1, DEC = 12;
int main()
{
  for ( int y = 2000; y <= 2010; y++ )
    for ( int m = JAN; m <= DEC; m++ )
    {
      for ( int d = 1; d <= DAYS_PER_MONTH; d++ )
        cout << m << "/" << d "/" << y << ' ';
      cout << endl;
    }
}
```

Loop Caveats

- loop control variable is only in scope over loop body

```
for (int i = 0; i < 10; i++ )
  cout << i;
cout << i; /// i is no longer in scope
```

- some errors may cause an infinite loop

```
for (int i = 0; i < 10; i+1 ) /// i+1 is not advancing
  cout << i;
...
int i; /// may forget to initialize
while ( i < 10 )
  cout << i; /// not advancing!
```

- some errors may cause wrong values for i or incorrect number of loops

```
for (int i = 0; i <= 10; i++ ) /// wrong < operator
  cout << i;
...
for (int i = 1; i < 10; i++ ) /// wrong initial value
  cout << i;
```

Simple Arrays

- a fixed size, single-dimensional array of elements of the same type
- EG an array of three integers

```
int a[3] = {0, 1, 2};
```

- processed naturally with a for loop

```
for ( int i = 0; i < 3; i++ )
  a[i] += 5; // add 5 to each element of array a
```

- can access individual elements directly

```
a[2] = a[0]; // assign value at a[0] into memory at a[2]
```

-

---

can print them out

```
for ( int i = 0; i < 3; i++ )
  cout << a[i] << endl;
```

- you must keep track of the array size

```
const int A_LENGTH = 3;
class ArrayHolder
{
private:
  int a[A_LENGTH];
  ...
public:
  void print( ostream & out )
  {
    for ( int i = 0; i < A_LENGTH; i++ )
      out << a[i] << endl;
  }
};
```

Extended Example

- EG class TimeSheet

```
 #include <iostream.h>
 #include <assert.h>
const int DAYS_PER_WEEK = 7;
class TimeSheet
{
private:
  int hoursWorked[DAYS_PER_WEEK];
public:
  TimeSheet()
  {
    for ( int i = 0; i < DAYS_PER_WEEK; i++ )
      hoursWorked[i] = 0;
  }
  void print( ostream & out )
  {
    for ( int i = 0; i < DAYS_PER_WEEK; i++ )
      out << "On day "
        << i
        << " worked "
        << hoursWorked[i]
        << " hours\n";
  }
  void recordHours(int i, int hours)
  {
    assert( i >= 0 && i < DAYS_PER_WEEK );
    assert( hours >= 0 );
    hoursWorked[i] = hours;
  }
  int totalHours()
  {
    int totalHours = 0;
    for ( int i = 0; i < DAYS_PER_WEEK; i++ )
      totalHours += hoursWorked[i];
    assert( totalHours >= 0 );
    return totalHours;
  }
};
```

- EG using class TimeSheet

```
int main()
{
  TimeSheet mySheet;
  mySheet.recordHours(MON, 8);
  mySheet.recordHours(TUE, 9);
  mySheet.recordHours(WED, 6);
  mySheet.recordHours(THU, 9);
  mySheet.recordHours(FRI, 4);
  mySheet.print( cout );
  cout << "Worked "
     << mySheet.totalHours()
     << " total hours this week\n";
  return 0;
}
```

Character Arrays (AKA character strings)

- character strings are arrays of characters terminated by '\0'
- tricky thing is you need an extra element for the terminator
- Three examples (of the string containing "abc")

```
char s1[4] = {'a','b','c','\0'};
char s2[4] = "abc";
char s3[] = "abc";
```

Searching a character string for a specified character

- to find the index of an element containing a specified value

```
int findIndexOfChar(char c, char s[])
```

```
{
  for ( int i = 0; s[i] != '\0'; i++ )
    if ( s[i] == c )
      return i;
  return -1;
}
```

- example of use

```
int main()
{
  char s[] = "Hello There";
  int posT = findIndexOfChar( 'T', s );
  if ( posT == -1 )
    cout << "T is not in " << s << endl;
  else
    cout << "T is at position " << posT << endl;
  s[posT] = 'W';
  cout << s << endl; // prints: Hello Where
}
```

String Library Functions

- important low-level C-string utilities

```
 #include <string.h>
int    strlen(const char s[]);
int    strcmp(const char s1[], const char s2[]);
char [] strdup(const char s[]);
char [] strcpy(char s1[], const char s2[]);
char [] strcat (char s1[], const char s2[]);
```

String Class

- always useful to use a class around a character array

```
 #include <assert.h>
 #include <iostream.h>
const int STRING_LENGTH = 128; // max length of a string
class String
{
private:
  char buffer[STRING_LENGTH];
```

- public:

```
  String( char s[] = "" )
  {
    assert( s != 0 );
    int i;
    for ( i = 0; s[i] != '\0' && i < STRING_LENGTH - 1;
i++ )
      buffer[i] = s[i];
    buffer[i] = '\0';
```

```
 }
 bool equals( String w )
 {
  int i;
  for ( i = 0; w.buffer[i] != '\0' && buffer[i] != '\0';
i++ )
   if ( w.buffer[i] != buffer[i] )
    return false;
  return w.buffer[i] == buffer[i];
 }
 void print( ostream & out )
 {
  out << buffer;
 }
 void read( istream & in )
 {
  in >> buffer;
 }
};
```

-
  ```
  bool operator == ( String w1, String w2 )
  {
   return w1.equals( w2 );
  }
  istream & operator >> ( istream & in, String & w )
  {
   w.read( in );
   return in;
  }
  ostream & operator << ( ostream & out, String w )
  {
   w.print( out );
   return out;
  }
  ```