COMP322 - Introduction to C++

Winter 2011

Lecture 05 - I/O using the standard library & Introduction to Classes

Milena Scaccia

School of Computer Science
McGill University

February 1, 2011

Final note on pointers

- Pointers to constants

```
double fee = 3.22;
const double *p1 = &fee;
const char *str1 = ‘‘hello’’;
```

We *cannot* change the value that `p1` or `str1` point to.

- Constant Pointers

```
int n = 21;
int * const p2 = &n;
char * const str2 = ‘‘hello’’;
```

We *can* change the value of `n` or of the message, but `p2` and `str2` *cannot* point to anything else.

- Constant Pointers to Constants

```
const int * const p3 = &n;
const char * const str3 = ‘‘hello’’;
```

We *cannot* alter the addresses of `p3`, `str3`, *nor* the value of the data pointed to.

Basic I/O in C++

Recall that in C, we have three file handles which are automatically created by the runtime library:

- stdin - standard input (from the keyboard/terminal)
- stdout - standard output (to the terminal/shell window)
- stderr - standard error

C++creates three 'stream' objects which encapsulate these handles:

- cin - standard input (from the keyboard/terminal)
- cout - standard output (to the terminal/shell window)
- cerr - standard error

Formatted I/O

In C++, the '<<' operator has been overloaded to write formatted data to `stream` objects:

```
#include <iostream>
using namespace std;
int n = 1;
double x = 2.0;
cout << n << " " << x << endl;
```

In C++, the '>>' operator has been overloaded to read formatted data from stream objects.

```
#include <iostream>
using namespace std;
int n;
double x;
cin >> n >> x;
```

Output stream manipulators

We can control the details of numeric output using stream manipulators, for example:

Name	**Temp**	**Description**
`hex`	N	Print integers in base 16
`oct`	N	Print in base 8
`dec`	N	Print in base 10
`fixed`	N	Set fixed precision
`scientific`	N	Set scientific notation
`left`	Y	Left justify in width
`right`	Y	Right justify in width
`setw(n)`	Y	Set *minimum* field width
`setprecision(m)`	N	Set number of decimal places
`setfill('0')`	N	Set fill character

'Temp' manipulators apply only to the next item printed.

Using I/O manipulators

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
  int n = 123;
  double x = 123.4567;

  cout << hex << n << endl
       << dec << right << setw(6) << n << endl
       << setfill('*') << left << setw(6) << n << endl
       << scientific << x << endl
       << fixed << setprecision(2) << x << endl;
}
```

Produces the output:

```
7b
   123
123***
1.234567e+02
123.46
```

File input/output

For I/O with named files, we use objects of class 'ofstream' or 'ifstream':

```
#include <fstream>
ofstream out_file;
ifstream in_file;

in_file.open("in.txt");
out_file.open("out.txt");
```

Now one can read or write formatted data from these stream objects:

```
in_file >> n;
out_file << n << endl;
```

When appropriate, we can close the file and reuse the object:

```
in_file.close();
out_file.close();
```

Verifying file operations

The >> operator returns a boolean which is true if the operation succeeds:

```
int n;
while (in_file >> n) {
  // process new input
}
```

we can also call member functions to retrieve status:

```
float x;
in_file >> x;
if (in_file.fail()) {
  // operation did not work, 'x' is invalid
}
```

Reading unformatted characters

If we naïvely attempt to read characters using the >> operator, this will skip spaces, end of line characters, etc.

You can read individual characters, including spaces, using the `get()` method:

```
char ch;
while (in_file.get(ch)) { // Read next character into 'ch'
  // do something
}
```

We can put a character back into the input stream using `unget()`:

```
in_file.unget(ch); // Pretend we didn't read 'ch'
```

Reading line-by-line input

We can read through an entire file using the `getline()` function:

```
string nextline;
ifstream in_file;
in_file.open("input.txt");
while (getline(in_file, nextline)) {
  // process this line
}
in_file.close();
```

We can then use string functions, or the `istringstream` class, to parse the line.

Introduction to Classes in C++

C++ is a **multi-paradigm** programming language:

- Generic Programming (Templates) (To be seen later)
- Imperative Programming (Procedural) (We have seen this)
- Object-Oriented Programming (Classes) (This lecture)

Classes in C++

A *class* can be thought of as an abstract data type, from which we can create any number of *objects*.

In C++ , a class allows us to hide implementation details from clients, and allows inheritance from one or more base classes, creating a class hierarchy (multiple inheritance).

Defining a Class

A class definition can be broken up into two parts:

1. Class Declaration (Defined in a .h header file)
   - Specifies what data members and member functions the class will have.
2. Class Implementation (Defined in a .cpp source file)
   - Provides the code for the function bodies

This is to separate the interface from the implementation. However, separating the code this way is not required. We may include code for the member functions in the .h file when it involves only one or two statements.

Class Declaration

The following is an example declaration for the complex number class. This should be declared in a file called `complex.h`.

```
class complex{
 private:
  float real; // real part
  float imag; // imaginary part

 public:
  complex(); // default constructor
  complex(float r, float i); // full constructor
  float getReal();
  float getImag();
  void setReal(float r);
  void setImag(float i);
  complex add(const complex &y);
};
```

Class Declaration

```
class complex{
 private:
  float real; // real part
  float imag; // imaginary part

 public:
  complex(); // default constructor
  complex(float r, float i); // full constructor
  float getReal();
  float getImag();
  void setReal(float r);
  void setImag(float i);
  complex add(const complex &y);
};
```

Visibility specifiers for members are supplied by section.
Data members generally have private visibility.
`private` $\Longrightarrow$ visible only to other members of this very class.

Class Declaration

```
class complex{
 private:
  float real; // real part
  float imag; // imaginary part

 public:
  complex(); // default constructor
  complex(float r, float i); // full constructor
  float getReal() const;
  float getImag() const;
  void setReal(float r);
  void setImag(float i);
  complex add(const complex &y);
};
```

Constructors and other member functions have `public` (global) visibility.
Note there is a third kind of visibility modifier:
`protected` $\Longrightarrow$ visible to this class and all of its descendants.

Class declarations end in a semi-colon.

Class Implementation

We can implement the functions in a file `complex.cpp` as follows:

```
#include "complex.h"
complex::complex() { // Default constructor
  real = imag = 0.0;
}
complex::complex(float r, float i) {
  real = r;
  imag = i;
}
float complex::getReal() const{
  return real;
}
void complex::setReal(float r){
  real = r;
}
// ...
complex complex::add(const complex &y) {
  return complex(real + y.real , imag + y.imag);
}
```

Using our class

We can create a main function to use our class.

```
int main(){
  complex a(1.0, 2.0);
  complex b(2.0, 1.0);
  complex c; // invokes default constructor

  // We may also create a pointer to a complex number object
  complex *d = new complex(2.0,3.0);
  complex *e = new complex;

  c = a.add(b);
  cout << c.getReal() << " + " << c.getImag() << "i" << endl;
}
```

This code will print

```
3 + 3i
```

Constructors

Each class can define one or more constructors. These are special functions which have the same name as the class, and have no defined return type.

- Default Constructor
- "Full" Constructor

Constructors

The appropriate constructor is called automatically when an object is created.

The *default* constructor is the constructor with no arguments. It simply fills in a "reasonable" set of values.

In our example `main()` function, the declaration

```
complex a(1.0, 2.0);
```

invokes the "full" constructor, while the declaration

```
complex c;
```

invokes the default constructor.

Member functions

Unless specified otherwise, a member function is invoked by dereferencing a specific object:

```
c = a.add(b);
```

The object through which we invoke the member function is an implicit parameter to the function. It may be accessed simply by using a member name:

```
complex complex::add(const complex &y) {
  return complex(real + y.real , imag + y.imag);
}
```

Const member functions

If a member function is declared `const`, by placing the keyword after the parameter list, this means the member function will not make any changes to the implicit parameter:

```
class complex {
  // ...
  float getImag() const;
  // ...
};

float complex::getImag() const {
  return imag;
}
```

In comparison, consider a function to set the imaginary part:

```
void complex::setImag(float i) {
  imag = i;
}
```

Issues with constructors

Consider the following class declaration:

```
class intStack {
  int top;
  int max;
  int size;
  int *data;

  intStack(int max = 100) {  // Constructor
    intStack::max = max;
    data = new int[max];
    size = 0;
  }

  int pop();
  void push(int i);
};
```

Issues with constructors cont'd

What happens if we initialize a new object with an old one?

```
int main() {
 intStack a;
 a.push(1);
 a.push(2);

 intStack b = a; // initialize new object with an old one

 cout << b.pop() << endl;
 a.push(3);
 cout<< b.pop() << endl;
}
```

Perhaps surprisingly, this prints:

2
3

If no copy constructor is defined, C++ uses the default copy constructor which copies each field (i.e. *shallow* copy).

Copy constructor

The solution to this problem is to provide a *copy constructor*, which copies the entire data structure.

The most generic form of copy constructor is:

```
class X {
  X(const X &src);
  //...
};
```

For our intStack example, it would be:

```
intStack(const intStack &src) {
    max = src.max;
    data = new int[max];
    size = src.size;
    for (int i=0; i<size; i++) {
      data[i] = src.data[i];
    }
 }
```

Deep copy

Destructors

A *destructor* is another "special" member function. The class destructor is called when an object of a given class is deleted. This gives an opportunity for the class to free memory or other resources.

The destructor always has the name $\sim\langle$*classname*$\rangle$:

```
class intStack {
  int top;
  int max;
  int size;
  int *data;
  intStack(int max = 100) {  // Constructor
    intStack::max = max;
    data = new int[max];
    size = 0;
  }
  intStack(const intStack &src) { // Copy Constructor ...}
  ~intStack() { // Destructor
    delete [] data;
  }
  int pop();
  void push(int i);
};
```

More complex destructors

```
class Symtable {
private:
  Symbol *head;
  Symbol *find(string name) {
    for (Symbol *sp = head; sp != NULL; sp = sp->link)
      if (sp->name == name) return sp;
    return NULL;
  }
public:
  Symtable() { head = NULL; } // Empty
  ~Symtable() {
    while (head != NULL) {    // Free the list
      Symbol *sp = head->link;
      delete head;
      head = sp;
    }
  }
  void set(string name, int value);
  int get(string name) {
    Symbol *sp = find(name);
    return (sp == NULL ? 0 : sp->value);
  }
};
```

Recall the Symbol structure

```
struct Symbol{
  struct Symbol *link; // Next in linked list
  int value;           // Value
  char name[32];       // Text name
};
```