And Even More and More C++

Outline

- C++ Classes
  - Friendship
  - Inheritance
  - Multiple Inheritance
  - Polymorphism
  - Virtual Members
  - Abstract Base Classes
- File Input/Output

Friendship

- Friend functions
  - A non-member function in a class marked as “friend” makes it so that other instantiated objects of the **same** type can access each other’s information

Friend Function Example

```
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

```
24
```

More Friendship

- Friend Classes
  - A friend of a class can access protected and private items within that class

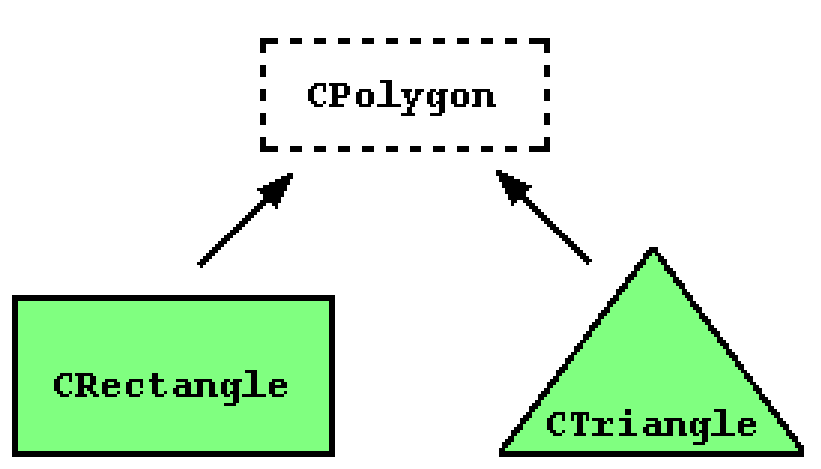
Friend Class Example

```
// friend class
#include <iostream>
using namespace std;

class Square;

class Rectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (Square a);
};

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

```
16
```

Inheritance

- Base class is the parent class
- Derived classes are the children
  - Children inherit the members of its parent
  - Children can also add their own members




```
class derived_class_name: public base_class_name
{ /*...*/ };
```

Inheritance Example

```
// derived classes
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

```
20
10
```

Access Permissions

- External access permission to class data

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

- Inherited members inherit access permissions dependent on how they are declared
  - Public – same access permissions (default for struct inheritance)
  - Protected – public and protected members inherited as protected
  - Private – all inherited members are private(default for class inheritance)

Inheritance

- What gets inherited?
  - A publicly derived class inherits everything except:
    - constructors and destructor
    - assignment (operator=)
    - friends
    - private members
      - this means that private variables are not inherited
      - much like Java, need to provide getters and setters
  - Even though not inherited, constructors and destructor are automatically called by the child class

Inheritance Example

```
// constructors and derived classes
#include <iostream>
using namespace std;

class Mother {
  public:
    Mother ()
      { cout << "Mother: no parameters\n"; }
    Mother (int a)
      { cout << "Mother: int parameter\n"; }
};

class Daughter : public Mother {
  public:
    Daughter (int a)
      { cout << "Daughter: int parameter\n\n"; }
};

class Son : public Mother {
  public:
    Son (int a) : Mother (a)
      { cout << "Son: int parameter\n\n"; }
};

int main () {
  Daughter kelly(0);
  Son bud(0);

  return 0;
}
```

```
Mother: no parameters
Daughter: int parameter

Mother: int parameter
Son: int parameter
```

Multiple Inheritance

- Couldn’t do this in Java!
- Done by specifying more than one base class separated by commas

Multiple Inheritance Example

```
// multiple inheritance
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
};

class Output {
  public:
    static void print (int i);
};

void Output::print (int i) {
  cout << i << '\n';
}

class Rectangle: public Polygon, public Output {
  public:
    Rectangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height; }
};

class Triangle: public Polygon, public Output {
  public:
    Triangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height/2; }
};

int main () {
  Rectangle rect (4,5);
  Triangle trgl (4,5);
  rect.print (rect.area());
  Triangle::print (trgl.area());
  return 0;
}
```

```
20
10
```

Polymorphism

- Key concept:
  - A pointer to a derived class is type-compatible with a pointer to its base class
  - This means we can use base class operations on different types of derived classes
  - And that means… polymorphism!

Polymorphism Example

```
// pointers to base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

```
20
10
```

Virtual Members

- A member function that can be redefined in a derived class
  - Like an abstract method in Java

```
virtual int area ()
  { return 0; }
```

Virtual Member Example

```
// virtual members
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

```
20
10
0
```

Abstract Base Classes

- Very similar concept to Java abstract classes
  - A class that is not intended to be instantiated
  - Can contain "pure virtual" functions
    - Functions with no definition
  - A class with at least one pure virtual function is an abstract class

```
// abstract class CPolygon
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

Abstract Base Class Example

```
// abstract base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  return 0;
}
```

```
20
10
```

Dynamic Allocation and Polymorphism

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    virtual int area (void) =0;
    void printarea()
      { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    Triangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height/2; }
};

int main () {
  Polygon * ppoly1 = new Rectangle (4,5);
  Polygon * ppoly2 = new Triangle (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

```
20
10
```

File Input/Output

- We've already done keyboard input and screen output
  - But that doesn't preserve data
    - The Homework.cpp lab assignment is fairly useless since all the entered data goes away when you quit the program
- Just like in Java, we can read from files and write to files
  - ofstream: stream used to write to file (output file stream)
  - ifstream: stream used to read from a file (input file stream)
  - fstream: stream to both read from and write to a file (file stream)

File Input/Output Example

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

```
[file example.txt]
Writing this to a file.
```

- Declare a file by its operation type
- Open the file
- Perform read/write operations on the file
- Close the file

Opening a File

- A file can be opened in different modes:

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

- Modes can be combined using the bitwise or operator (|):

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

Files

- Can have either text or binary files (just like Java)
- Can create and open file in a single statement:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

- Can then check to see if the file opened successfully:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

- After file input/output is complete, should close the file:

```
myfile.close();
```

Moving Around in a File

- All file streams keep track of at least one position in the file
- get and put positions
  - Input streams keep an internal “get” position
    - This is where the next data item will be read from
  - Output streams keep an internal “put” position
    - This is where the next data item will be written
- tellg() and tellp() are functions that retrieve the position
- seekg() and seekp() allow you to move the position

Moving Around in a File

- Can use absolute positioning or relative positioning
  - seekg(position) is absolute – it will go to that position
  - seekg(offest, direction) is relative – it will move "offset" number of positions past "direction"
    - direction can be:

ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position
ios::end	offset counted from the end of the stream

Example

```
// obtaining file size
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos begin,end;
  ifstream myfile ("example.bin", ios::binary);
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin) << " bytes.\n";
  return 0;
}
```

```
size is: 40 bytes.
```

Binary Files

- We use the >> and << operators to write to text files
- This is not efficient for binary files
  - Use read and write instead

```
// reading an entire binary file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos size;
  char * memblock;

  ifstream file ("example.bin", ios::in|ios::binary|ios::ate);
  if (file.is_open())
  {
    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "the entire file content is in memory";

    delete[] memblock;
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
the entire file content is in memory
```

Buffers and Synchronization

- When we work with files, they are associated with an internal buffer of type streambuf
  - This is so that we don't write to disk for every single piece of data
- When the buffer is flushed, that is when the data is actually written to disk
  - Called synchronization
  - Happens when:
    - File is closed by the program
    - When the buffer is full
    - Explicitly (you can force a buffer flush with the flush() command)
    - Explicitly with sync()

Summary

- C++ Classes
  - Friendship
  - Inheritance
  - Multiple Inheritance
  - Polymorphism
  - Virtual Members
  - Abstract Base Classes
- File Input/Output