Object Oriented Programming

- We've now seen the *non* Object Oriented features of C++
- We need them to be able to use the Object Oriented features.

class: data members, member functions

- The basic unit of C++ is the *class*.
- A class is a grouping of *data* and *functions* into one unit. Usually the functions (member functions) operate on the data.
- Only some of the member functions (the interface) are available from outside the class – usually *none* of the *data members* are accessible from outside the class.
- An *instance* of a class is called an *object*.

Example:
Use a class to specify a graphics (x, y) point. We first *declare* the class:

```
#ifndef __POINT_HH   // Point1.hh
#define __POINT_HH

#include <iostream.h>

class Point {
public:
  Point();
private:
  int m_x, m_y;
};

#endif // __POINT_HH
```


Point is now a new data type (just as int, float etc.) We next have to complete the *definition* (or implement) the Point class:

```
#include "Point1.hh"   // Point1.cc

Point::Point() {
  m_x = 0;
  m_y = 0;
}
```

Finally, we can use the class Point in our program:

```
#include "Point1.hh"   // Point-1.cc

int main() {
  Point p;
}
```


Points to note:

1. The class `Point` has 2 blocks – one labeled `public:`, the other `private:`
   - class members in the `public:` section can be accessed from outside the class.
   - class members in the `private:` section can *only* be accessed from inside the class.
2. There is a member function with the same name as the class, and *no return type*. This is the class *constructor*.
3. Since the data are in the `private:` section of `Point`, they cannot be accessed from outside the class. This is called *data encapsulation*.
4. In the implementation, the constructor function name is written `Point::Point()`. `::` is the scope resolution operator. The constructor does *not* have a type (not even `void`).

5. An instance of the `Point` class, the object `p`, is declared in the same way as for any other type.
6. There is a problem: this object does nothing! Since the data are `private` they are not accessible, and there are no member functions (apart from the constructor), so we can't *do* anything with `p`.
7. Moreover, since `p` is declared but not used, we will get a compiler warning.
8. We can fix this by adding a public member function.


```
#ifndef __POINT_HH   // Point2.hh
#define __POINT_HH

#include <iostream.h>

class Point {
public:
  Point();
  void print();
private:
  int m_x, m_y;
};

#endif // __POINT_HH
```

we have added the (public) member function, `void print()` which we must now implement.

```
#include "Point2.hh"   // Point2.cc

Point::Point() {
  m_x = 0;
  m_y = 0;
}

void Point::print() {
  cout << "(" << m_x << ", " << m_y << ")";
}
```



```
#include "Point2.hh"   // Point-2.cc

int main() {
  Point p;
  p.print();
  cout << endl;
}
```


1. In `Point::print()`, we have not added an endl, because we may want to print other things. We will then have to flush the buffer.
2. `Point::print()` is a member of `class Point`, so it can access `Point`'s private data.
3. `p`'s print function, which accesses `p`'s data, is invoked with the "`.`" operator.

This is progress, but still doesn't allow us to do very much. What about instantiating `Point` with other values?

To do this, we need a different constructor. But wait! We can use function overloading to define a constructor with a different signature.


```
#ifndef __POINT_HH   // Point3.hh
#define __POINT_HH

#include <iostream.h>

class Point {
public:
  Point();
  Point(int, int);
  void print();
private:
  int m_x, m_y;
};

#endif // __POINT_HH
```

```
#include "Point3.hh"   // Point3.cc

Point::Point() {
  m_x = 0;
  m_y = 0;
}

Point::Point(int initX, int initY) {
  m_x = initX;
  m_y = initY;
}

void Point::print() {
  cout << "(" << m_x << ", " << m_y << ")";
}
```



```
#include "Point3.hh"   // Point-3.cc

int main() {
  Point p;
  p.print();
  cout << endl;
  Point q(10,15);
  q.print();
  cout << endl;
}
```


Notes:

1. Since the signatures are different, the overloaded constructor functions really are 2 different functions.
2. In the `print()` method, p's `print()` accesses p's data, and q's `print()` accesses q's data.
3. Using member functions to hide the member data is called "Data Encapsulation", and is an important feature of Object Oriented programming. In general, the data should *always* be encapsulated, and so should *never* be public.
4. If we want to access the data (and in practice we need to less than we think), then we should do it via *access* functions:


```
#ifndef __POINT_HH   // Point4.hh
#define __POINT_HH

#include <iostream.h>

class Point {
public:
  Point();
  Point(int, int);
  void print();
  int x() { return m_x; }
  int y() { return m_y; }
private:
  int m_x, m_y;
};

#endif // __POINT_HH
```

```
#include "Point4.hh"   // Point-4.cc
#include <math.h>

int main() {
  Point p;
  p.print();
  cout << endl;
  Point q(10,15);
  q.print();
  cout << endl;
  cout << "q's radius: " << sqrt(q.x()*q.x()+q.y()*q.y()) << endl;
}
```

But why not just provide a member function to return the radius?


```
#ifndef __POINT_HH   // Point5.hh
#define __POINT_HH

#include <iostream.h>

class Point {
public:
  Point();
  Point(int, int);
  void print();
  int x() { return m_x; }
  int y() { return m_y; }
  int r();
private:
  int m_x, m_y;
};

#endif // __POINT_HH
```



```
#include "Point5.hh"   // Point5.cc
#include <math.h>

Point::Point() {
  m_x = 0;
  m_y = 0;
}
Point::Point(int initX, int initY) {
  m_x = initX;
  m_y = initY;
}
void Point::print() {
  cout << "(" << m_x << ", " << m_y << ")";
}
int Point::r() {
  return (int)sqrt( m_x*m_x + m_y*m_y );
}
```



```
#include "Point5.hh"   // Point-5.cc

int main() {
  Point p;
  p.print();
  cout << endl;
  Point q(10,15);
  q.print();
  cout << endl;
  cout << "q's radius: " << q.r() << endl;
}
```

Clearly, we could proceed this way adding as many member functions as we could think of.
In particular, we could use function overloading to add "set" functions:

```
  void x(int);
  void y(int);
```

These are *different* functions than:

```
  int x();
  int y();
```

It is the arguments, not the return type, that makes them different.

constructors: initialization

Just as we invoke the constructor for a user-defined class with:

```
  Classname  Objectname(initial parameters);
```

so we can also initialize a built-in type with the same syntax:

```
  Typename  Objectname(initial value);
```

i.e.

```
  int i(6);  // is equivalent to:  int i=6
```


constructors: default arguments

We often want to instantiate an object with default values. We can put the default values in the *declaration*, and use them in the *implementation*.

```
class Point {
public:
  Point(int initX=0, int initY=0);
  ...
};
```

```
Point::Point(int initX, int initY) {
  m_x = initX;
  m_y = initY;
}
```


- If *one* argument is given explicitly, we must also specify all preceding arguments.
- There can be an ambiguity between using *default* arguments, and function *overloading* – choose one or the other for a specific signature.
- Default arguments are useful – but don't go overboard.

Name Conventions

There are few *rules* about variable and function names – but some conventions are useful

- Begin *class* names with an **U**ppercase letter
- Header files for `class Foo` should be called `Foo.hh` and `Foo.cc`
- Begin *member data* names with something distinctive, e.g. `m_`
- Begin *member function* names with a **l**owercase letter
- Use descriptive names – concatenate and capitalize. E.g. `printMyValue`

- Don't use names differing *only* by case.
- Begin *constant* data names with something distinctive, such as `k`. E.g. `const kArraySize`. Sometimes it is useful to capitalize.
- *Instance* variables (of small scope) can be terse
- In a class declaration, list *first* `public` members, then `private` members.

references

Suppose we want to add an `rmoveTo(Point)` method, where we specify the new Point as an argument.
(Of course, we could easily write a `rmoveTo(int,int)` method, but that's not very "Object Oriented".)

We can do this easily enough by changing `Point.hh` and `Point.cc`

```
class Point {
public:
  ...
  void rmoveTo(Point);
};
```



```
void Point::rmoveTo(Point p) {
  m_x += p.m_x;
  m_y += p.m_y;
}
```

Note that although `p` is a *different* object than the current (`this`) object, `rmoveTo(Point)` is still a `Point` member function, so it has access to `Point`'s private data.

```
#include "Point6.hh"   // Point-6.cc

int main() {
  Point p;
  Point q(10,15);
  p.print();
  p.rmoveTo(q);
  p.print();
  p.rmoveTo(Point(3,4));
  p.print();
  cout << endl;
}
```


For the argument of `rmoveTo`, we can:

- either use an existing object, q
- or use the constructor function (since it really returns a `Point` object). In this case, we don't define another object.

This is all just fine, and works – but there is a subtle but important problem:

> **Rule** **In C++, function arguments are passed by *value*.**

That is, a copy of the `Point` object is put on the stack, and this *copy* is used by the function.

- In this case, `Point` is not such a large object, so who cares?
- In general, the object could be arbitrarily large, and this could be *very* inefficient.
- So C++ gives us another way of passing function arguments – by **Reference**.
- Instead of making a copy and putting the copy on the stack, we put an alias name (the reference) on the stack, and access the *original* object.


```
class Point {
public:
  ...
  void rmoveTo(Point&);
};
```

```
void Point::rmoveTo(Point& p) {
  m_x += p.m_x;
  m_y += p.m_y;
}
```

This looks *almost* the same as before, except for the `&`.

We read the declarations backwards, so `rmoveTo(Point& p)` means:

p is a *reference* to a `Point` object

- This solves the problem of unnecessarily copying large objects – but it creates another problem:
- we *could* (accidentally or maliciously) change the contents of the argument object, since in the `rmoveTo` method, we are working with the *same* object, and not a copy.

```
void Point::rmoveTo(Point& p) {
  m_x += p.m_x;
  m_y += p.m_y;
  p.m_x += 7;  // accidental code
}
```


Sometimes we might want to do this – but when we don't, C++ allows us to protect us from ourselves. We can declare not a reference, but a *constant reference.*

```
class Point {
public:
  ...
  void rmoveTo(const Point&);
};
```

```
void Point::rmoveTo(const Point& p) {
  m_x += p.m_x;
  m_y += p.m_y;
}
```


Notes:

1. `void rmoveTo(const Point&);` is a *different* signature than `void rmoveTo(Point&);`
2. in principle, we could use `const Class&` for *all* arguments – even built-in types, but this is usually overkill:
   (a) it is no more overhead to put an `int` on the stack
   (b) with call by value, we protect the argument anyway
   (c) often we might want to change the value *inside* the function
3. using `(type&)` for *built-in* types is a good way of changing parameters (when using a return value is not convenient).
4. we will be using references extensively. References mean that *pointers* (see later) can be largely – tho not completely – avoided.


```
#ifndef __POINT_HH   // Point7.hh
#define __POINT_HH

#include <iostream.h>

class Point {
public:
  Point(int initX=0, int initY=0);
  void print();
  int x() { return m_x; }
  int y() { return m_y; }
  int r();
  void rmoveTo(const Point&);
private:
  int m_x, m_y;
};

#endif // __POINT_HH
```

```
#include <math.h>   // Point7.cc
#include "Point7.hh"

Point::Point(int initX, int initY) {
  m_x = initX;
  m_y = initY;
}
void Point::print() {
  cout << "(" << m_x << ", " << m_y << ")";
}
int Point::r() {
  return (int)sqrt( m_x*m_x + m_y*m_y );
}
void Point::rmoveTo(const Point& p) {
  m_x += p.m_x;
  m_y += p.m_y;
}
```



```
#include "Point7.hh"   // Point-7.cc

int main() {
  Point p;
  Point q(10,15);
  p.print();
  p.rmoveTo(q);
  p.print();
  p.rmoveTo(Point(3,4));
  p.print();
  cout << endl;
}
```


Arrays of Objects

We can have arrays of objects, as well as arrays of built-in types. The syntax is the same:

```
#include "Point.hh"   // Point-Array.cc
#include <stdlib.h>
int main() {
  const int kArraySize=7;
  const int kRandomStep=20;
  Point p[kArraySize];
  for (int i=0; i<kArraySize; i++) {
    p[i].rmoveTo( Point( rand()%kRandomStep, rand()%kRandomStep ) );
    p[i].print();
  }
  cout << endl;
}
```


templates

Templates are sometimes considered an advanced feature of C++, but they are so useful, and are *crucial* to STL, that we introduce them now.

Suppose we have a function `min`, which returns the smaller of its 2 arguments:

```
int min(int a, int b) { return (a<b) ? a : b;}
```

If we want `min` to work for `float`, we would have to overload `min` and use a float signature.

```
float min(float a, float b) { return (a<b) ? a : b;
```

This is not difficult – but soon gets tedious, especially when we want to use `min` for user-defined classes, such as `Point`.

The solution is to use a **function template**:

```
#ifndef __TEMPLATE_HH   // Template.hh
#define __TEMPLATE_HH

#include <iostream.h>

template<class T>
T min(T a, T b) {
  return (a<b) ? a : b;
}

#endif // __TEMPLATE_HH
```


This can now be used in the "obvious" way:

```
#include <math.h>   // Template.cc
#include "Template.hh"

int main() {
  int a(9);
  int b(16);
  cout << "min(" <<  a << "," << b << ") = " << min(a,b)  << endl;
  cout << "min(" << M_PI << "," << M_E << ") = "
       << min(M_E,M_PI) << endl;
}
```


Notes:

1. the templated function must be preceded by
   `template<class T>`
   `T` can be any dummy name. There can be several template arguments.
2. This function is now a *template* for whenever a `min` is needed. The compiler checks the signature, and generates the right `min` for that signature.
3. Each template argument *must* appear at least once in the signature. It *may* appear in the function type, but that cannot be the *only* reference (since the type alone does not generate the signature).
4. The actual arguments are used inside the function *as if* they were of a declared type. (Which they are, since the compiler generates them from the template.)
5. Function templates can be used inside classes, but don't have to be.

But there is a catch (or price):

- We can get code bloat, because every instance of the function has its own code – but we might have had that anyway.
- More seriously: we cannot put the function template in a header file, and the implementation in a different file.
  When the compiler comes across each function usage, it must generate the function with the right signature, so an implementation cannot be compiled ahead of time – it cannot know which signatures will be needed.

  1. the header files can get big
  2. compilation can take longer
  3. we could not hide our source code so easily

Template Classes

- Templates are also used in class definitions.
- Suppose we want to make the `Point` class work with types other than `int`. We can make `Point` a template class:


```
#ifndef __POINT_HH   // Point-Template.hh
#define __POINT_HH
#include <iostream.h>
#include <math.h>
template<class T>
class Point {
public:
  Point(T initX=0, T initY=0) { m_x = initX; m_y = initY; }
  void print() { cout << "(" << m_x << ", " << m_y << ")"; }
  T x() { return m_x; }
  T y() { return m_y; }
  T r() { return (T)sqrt( m_x*m_x + m_y*m_y ); }
  void rmoveTo(const Point<T>& p) { m_x += p.m_x; m_y += p.m_y; }
private:
  T m_x, m_y;
};
#endif // __POINT_HH
```



```
#include "Point-Template.hh"   // Point-Template.cc

int main() {
  Point<int> p(7,6);
  p.print();
  Point<float> q(3.142,2.718);
  q.print();
  cout << endl;
}
```

In general, templates are most useful with classes other than the built-in types. We will revisit later.