CLASS DESIGN

Today

- We will look in more detail at classes.
- The mian thing we will consider is limiting access to members of classes.
- This work is based on Pohl, Chapter 4.
- As before, much of this work will be concerned not only with what we can do in C++, but also the *style* in which we do it.

Public and private access

- Members of `class`es and `struct`s can be `public` or `private`
- `public` means that any code can access the members
- `private` means that only code inside the class or struct can access the members
  - (or "friend" classes, to be discussed later in the semester)
- Typically, following good OOP practice, all data members are `private` and only function members are `public`
  - (but not all—only those that need to be accessed outside of the struct or class).

- For example:

```
class point {
public:
  void print();
  void set( double u, double v );
private:
  double x, y;
}; // end of class--don't forget semi-colon!
```

(the rest of the example code is the same as the previous one)

- We could also write

```
struct point {
public:
  void print();
  void set( double u, double v );
private:
  double x, y;
}; // end of struct--don’t forget semi-colon!
```

(again, the rest of the example code is the same as the previous one)

“class” vs “struct”

- The difference between structs and classes is:
  - In a `struct`, the members are `public` by default
  - In a `class`, the members are `private` by default
- So, we could write our example as:

```
#include <iostream>
using namespace std;

class point {
// No private: is needed
  double x, y;
public:
  void print();
  void set( double u, double v );
}; // end of struct--don’t forget semi-colon!

void point::print() {
  cout << "(" << x << "," << y << ")\n";
} // end of print()

void point::set( double u, double v ) {
  x = u;
  y = v;
} // end of set()
```

- `main` looks the same as before:

```
int main() {
  point w;
  w.set( 1.2, 3.4 );
  cout << "point = ";
  w.print();
} // end of main()
```

- In this example, `x` and `y` are private and the methods are public.
- Otherwise, class and struct are the same
- But by convention, C++ programmers tend to use `class`

Nested classes

- Classes can be nested — one class is placed inside another.
- Here's another confusing example from the book:

```
char c;  // global scope

class X {
  public:
    char c;   // local scope in class X
    class Y {
      public:
        void foo( char e ) { X t; ::c = t.c = c = e; }
      private:
        char c; // local scope in class Y
    };
};
```

- The scope of the first `c` is `::c`.
- The scope of the second `c` is `X::c`.
- The scope of the third (last) `c` is `X::Y::c`
- The inner class, `Y` can only be referenced from within `X`.
- So, you can only create instances of `Y` within `X`, and you can only access even the public the data members of `Y` from within `X`.
- If this sounds overly confusing, then don't worry.
- You should be able to write all the programs you need *without* using nested classes.

"static" members

- The keyword `static` is used to refer to data members of a class that are the same across all instances of the class.
- In other words, it is independent of any class variable
- For example in the following program, `a.dimensions` and `b.dimensions` both have value 2.

```
class point {
  public:
    static int dimensions;
    .
    .
};
.
.
int main() {
  .
  .
  point::dimensions = 2; // initialize point
  .
  point a, b;
  .
}
```

“const” members and “mutable”

- Data members with the `const` keyword in their definition cannot be modified.
- For example:

```
class point {
  double x, y;
  public:
    const int dimensions = 2;
    void print() const;
};

void point::print() {
  cout << "(" << x << "," << y << ")\n";
} // end of print()
```

- `dimensions` cannot be modified.

- Confusingly, you can use the same keyword `const` along with function members.
- For example:

```
class point {
  double x, y;
  public:
    const int dimensions = 2;
    void print() const;
};

void point::print() const{
  cout << "(" << x << "," << y << ")\n";
} // end of print()
```

- This says that `print` is not allowed to modify any of the data members of `point`.

- Without specifying a method as `const`, it is allowed to alter *any* of the data members.
- Just to confuse the picture even further we have the keyword `mutable`.
- If, in some class definition, we define:

```
  mutable int delta;
```

  it means that `delta` can be modified by *any* method for that class, even if the method is defined as being `const`.

A more complex kind of class

- An example of another class is given in `basic-stack.cpp`.
- This implements a *stack*.
- A *stack* is a datastructure which can hold information in such a way that the first thing placed into the stack is the last thing to be removed from the stack.
- We think of a stack as allowing you to *push* information onto the stack.
- You can also *pop* information off the stack.

- Thus a stack is rather like a Pez dispenser:

- The example code will show you how to program this kind of behavior.
- It will also give you an idea what a more complex class than `point` looks like.

Aside: why is `stack` useful?

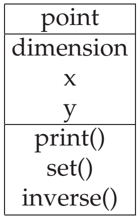
- There are several reasons.
- First, it is the simplest example of a *dynamic* data-structure — one where the memory that is uses is determined at *run-time* not *compile-time*.
- You will meet many other kinds of dynamic data-structure in the future, and understanding a stack will help you in understanding those others.
- (Of course, the basic stack isn't really dynamic, it is just a dressed up array, but soon we'll see how to make it really dynamic).

- Second, a *run-time stack system* is a system of memory allocation commonly used on most computers to keep track of how much memory is available to a program and allocates pieces of it as they are needed.
- When a function is called, the memory required for the function (e.g., its local variables) is allocated from (*pushed onto*) the stack; when the function exits, the memory is freed from (*popped off*) the stack
- Thus stacks are fundamental to the way that all computer programs work.

Class design

- Data members should be `private` ("hidden")
- Function members are often `public` (but not always—private function members can be used for computations internal to a class).
- Functions that do not modify data members should be `const`
- Pointers add indirection (we'll talk about that later)
- A uniform set of functions should be included: `set()`, `get()`, `print()`

- UML (unified modeling language) provides a graphical method for representing classes



point

dimension
x
y

print()
set()
inverse()

Summary

- This lecture introduced the basics of object-oriented programming.
- It showed how `struct` and `class` can be used to create aggregate datatypes and the methods for those types.
- It discussed public and private methods, and how these should be used in good class design.
- The lecture also looked at `static`, `const` and `mutable`, and mentioned features such as class nesting, and the `this` pointer.