# OBJECTS AND CLASS DESIGN

## Today

- We will start to talk about *object-oriented programming*
- In particular we will talk about `struct` and `class`.
- We will show how to use these features of C++ to define aggregate data types.
- We will show how to define *methods* that operate on these data types.
- This work is based on Pohl, Chapter 4.
- Much of the work we will do for the next couple of weeks will be concerned not only with what we can do in C++, but also the *style* in which we do it.



## Aggregate data types

- New today: `class` and `struct`
- `struct` comes from C
- `class` is new in C++
- Both are aggregate types, meaning that they group together multiple fields of data.
- For example:

```
struct point {
  double x, y;
};
```

- Don't forget to put a semi-colon at the end of the structure definition!



## Aside: why is `point` useful?

- The idea behind `point` is that it represents information about the location of something.
- Think of it as a pair of (Cartesian) coordinates.
- We group the coordinates together because they make no sense separately — if we have the x coordinate of a thing, then it has a y coordinate also.
- We will use `point` when we write a simulation of small eco-system. We will do this in some of the homeworks.

## Back to aggregate data types

- In C, the tag (`point`) is optional and does not constitute a data type (you need to use `typedef` as well).
- In C++, the tag is considered a data type, hence the above example is a data type definition.
- This means that you can use `point` as a data type, e.g.:

```
point p;
```

- In other words, you can declare a variable `p` which is of type `point`.



- The fields or elements of an aggregate data type are called *members*.
- Members are referred to using "dot notation", e.g.:

```
p.x = 7.0; p.y = 10.3;
```

- You can also use a *pointer* to access members of an aggregate data type, e.g.:

```
p->x = 12.3;
```

  but we will discuss pointers in the next unit, so don't worry about this now...
- The fields or elements of an aggregate data type are called *members*.



- You can also declare a structure and a variable of that type in the same statement, e.g.:

```
struct {
  double x, y;
} myPoints[3] = { {1, 2}, {3, 4}, {5, 6} };
```

  defines the array `myPoints` to hold three elements each of which is a `struct` which holds two `double`s, and sets the values of these.
- This is not the clearest way of doing things. I would prefer:

```
struct point {
  double x, y;
};

point myPoints[3] = { {1, 2}, {3, 4}, {5, 6} };
```



## Member functions

- In C++, members of aggregate data types can be functions
- (C only allows data members)
- In object-oriented programming (OOP) lingo, the word "method" is often used instead of "function"
- The reason to define functions inside an aggregate data type is to follow the OOP principle of *encapsulation*—operations should be packaged with data
- This is a *style* thing.
- For example:

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

struct point {
  double x, y;
  void print() {
    cout << "(" << x << "," << y << ")\n";
  }
  void set( double u, double v ) {
    x = u;
    y = v;
  }
}; // end of struct--don't forget semi-colon!

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



- Notes:
  - Notice that the set method changes the values of the data members—this is considered good OOP practise
  - Defining the methods inside the `struct` definition is called "in-line declaration"; this is generally only okay for short, concise methods
- The *class scope* operator can be used when in-line declarations are inappropriate.
- For example:



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

struct point {
  double x, y;
  void print();
  void set( double u, double v );
};

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

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



- The methods can then be invoked from `main`, just as before:

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

## Public and private access

- Members of structures 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 term)
- 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:

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

(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
- For example:



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

class point {
  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`



## Class scope

- The class scope operator is two colons (`::`), as in our example:

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

- The `::` operator has the highest precedence in the language, so it always gets evaluated first
- There are two versions of the operator: binary and unary
- The binary version is the one we used before: `point::print()`, which is used to refer to a variable's "class scope" (also called "local scope").
- The unary version is like this: `::count` and is used to refer to a variable's "external scope" (e.g., for a global variable).



- Here is a (maybe) confusing example from the book:

```
int count = 0; // declare global variable

void how_many( double w[], double x, int& count ) {
  for ( int i=0; i<N; ++i ) {
    count += ( w[i] == x ); // local count
  }
  ++::count; // global count
} // end of how_many()
```

- This is only necessary since `count` is declared twice
- If you didn't have the `::count`, then the second time, it would also refer to the local variable
- It is better practise not to use global variables; or at least if you do, give them unique names to avoid confusion :-)



## 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 `Y`, 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.



## "this" pointer

- The keyword `this` is used to refer to an instance of a class from within itself.
- It is a *pointer* — something we will discuss at length in the next unit
- For example:

```
point inverse() {
  x = -x;
  y = -y;
  return (*this);
}
```

- In this example, the function returns a pointer to the object that is calling it
- We'll come back to this when we discuss pointers



## "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`.



## Special types of classes: “containers”

- There are several special types of classes in C++.
- The first we will discuss is called a *container*.
- It is a class designed to hold large numbers of objects.
- An example of a container is a *stack*, a class 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.
- Our example will hold characters, and you can find it on the class webpage — it is the program `basic-stack.cpp`.

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

class ch_stack {
  public:
    void reset() { top = EMPTY; }
    void push( char c ) { s[++top] = c; }
    char pop() { return s[top--]; }
    char top_of() const { return s[top]; }
    bool empty() const { return( top==EMPTY ); }
    bool full() const { return( top==FULL ); }
  private:
    enum{ max_len = 100, EMPTY = -1, FULL = max_len - 1 };
    char s[max_len];
    int top;
};
```



```
int main() {
  ch_stack s;
  char str[40] = { "hello world!" };
  int i = 0;
  cout << "str=" << str << endl;
  s.reset();
  while( str[i] && ! s.full() ) {
    s.push( str[i++] );
  }
  cout << "reversed str=";
  while ( ! s.empty() ) {
    cout << s.pop();
  }
  cout << endl;
} // end of main()
```



## 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.
- 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<br>x<br>y |
| print()<br>set()<br>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.