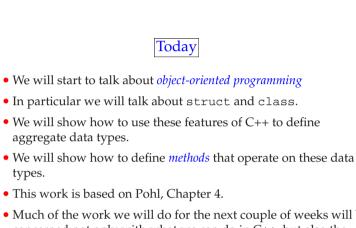
### **OBJECTS AND CLASS DESIGN**

### 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!



• 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.

cis15-fall2007-parsons-lectII.1

### 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.

cis15-fall2007-parsons-lectII.1

• 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 doubles, 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} }
```

- 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*.

cis15-fall2007-parsons-lectII.1

### 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();
```

```
cis15-fall2007-parsons-lectII.1
```

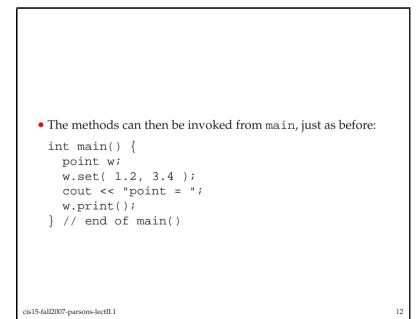
```
#include <iostream>
    using namespace std;
    struct point {
      double x, y;
      void print();
      void set( double u, double v );
    };
    void point::print() {
      cout << "(" << x << "," << y << ")\n";
    } // end of print()
    void point::set( double u, double v ) {
      x = u;
      y = v;
    } // end of set()
cis15-fall2007-parsons-lectII.1
                                                                           11
```

#### • 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.

10

• For example:



### 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).

cis15-fall2007-parsons-lectII.1

#### • For example:

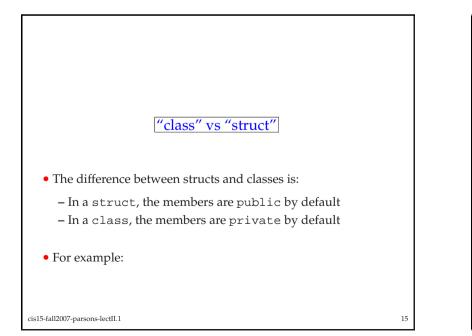
```
struct point {
public:
    void print();
    void set( double u, double v );
private:
    double x, y;
}; // end of struct--don't forget semi-colon!
(the rest of the example code is the same as the previous one)
```

14

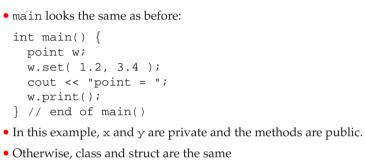
16

cis15-fall2007-parsons-lectII.1

13

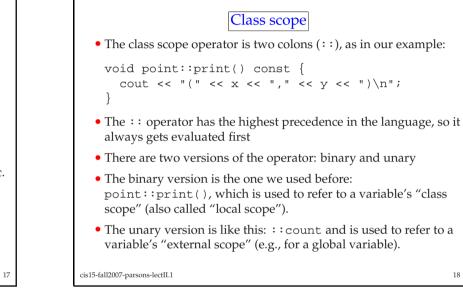


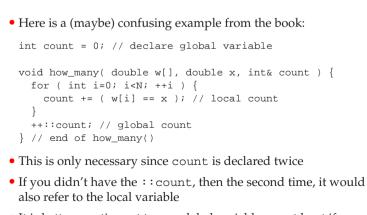
# #include <iostream> using namespace std; class point { double x, y; public: void print(); void set( double u, double v ); }; // end of struct--don't forget semi-colon! void point::print() { cout << "(" << x << "," << y << ")\n"; } // end of print() void point::set( double u, double v ) { x = uiy = v;} // end of set() cis15-fall2007-parsons-lectII.1



```
• But by convention, C++ programmers tend to use class
```

```
cis15-fall2007-parsons-lectII.1
```





• It is better practise not to use global variables; or at least if you do, give them unique names to avoid confusion :-)

### Nested classes

18

20

• 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
    };
};
```

<ul> <li>The scope of the first c is ::c.</li> <li>The scope of the second c is X::c.</li> </ul>	
• 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.	
<ul> <li>If this sounds overly confusing, then don't worry.</li> </ul>	
<ul> <li>You should be able to write all the programs you need witho using nested classes.</li> </ul>	ut
cis15-fall2007-parsons-lectII.1	21

## "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:

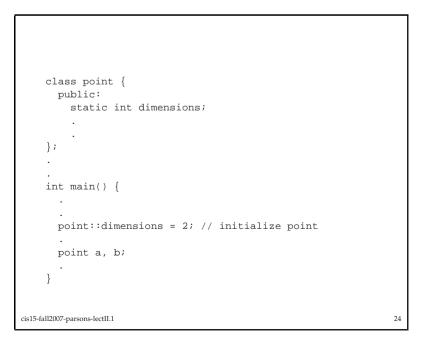
```
point inverse() {
  x = -x;
  y = -y;
  return (*this);
```

• In this example, the function returns a pointer to the object that is calling it

22

• We'll come back to this when we discuss pointers

```
cis15-fall2007-parsons-lectII.1
```



# "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.

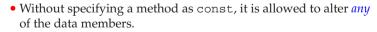
# "const" members and "mutable"

• 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.</pre>
```

cis15-fall2007-parsons-lectII.1



- 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.

• 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.</pre>
```

cis15-fall2007-parsons-lectII.1

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.

27

25

cis15-fall2007-parsons-lectII.1

26

```
#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;
   };
cis15-fall2007-parsons-lectII.1
                                                              29
```

### 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.

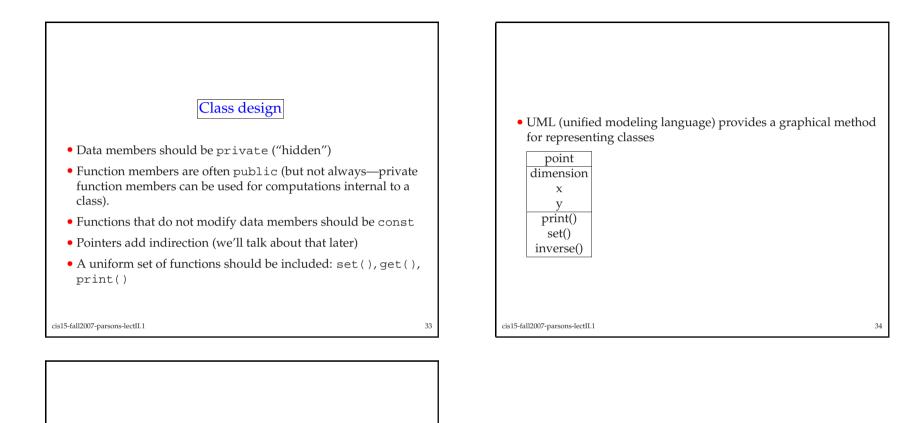
```
int main() {
    ch_stack s;
    char str[40] = { "hello world!" };
    int i = 0;
    cout << "str=" << str << endl;
    s.reset();
    while( str[i] && ! s.full() ) {
        s.push( str[i++] );
    }
    cout << "reversed str=";
    while ( ! s.empty() ) {
        cout << s.pop();
    }
    cout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

cis15-fall2007-parsons-lectII.1

- 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.

cis15-fall2007-parsons-lectII.1

30



### 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.