Object-Oriented programming in C++

Classes as units of encapsulation
Information Hiding
Inheritance
polymorphism and dynamic dispatching
Storage management
multiple inheritance
Classes

Encapsulation of type and related operations

class point {
    double x,y;                          // private data members

public:
    point (int x0, int y0);              // public methods
    point () { x = 0; y = 0;};          // a constructor
    void move (int dx, int dy);
    void rotate (double alpha);
    int distance (point p);
}

A class is a type: objects are instances

point p1 (10, 20);       // call constructor with given arguments
point p2;               // call default constructor

Methods are functions with an implicit argument
p1.move (1, -1);         // special syntax to indicate object

// in other languages might write  move (p1, 1, -1)
// special syntax inspired by message-passing metaphor:
// objects are autonomous entities that exchange messages.
Implementing methods

No equivalent of a body: each method can be defined separately

```cpp
void point::rotate (double alpha) {
    x = x * cos (alpha) - y * sin (alpha);
    y = y * cos (alpha) + x * cos (alpha);
};
// x and y are the data members of the object on which the
// method is being called.
// if method is defined in class declaration, it is inlined.
```
Constructors

One of the best innovations of C++
special method (s) invoked automatically when an
object of the class is declared

point (int x1, int x2); // cartesian parameters
point (); // default (at origin?)
point (double alpha; double r); // polar coordinates

point p1 (10,10), p2, p3 (pi / 4, 2.5);

• Name of method is name of class
• Declaration has no return type.
The target of an operation

- The implicit parameter in a method call can be retrieved through `this`: reference to object

```cpp
class Collection {
    Collection& insert (thing x) { // return reference
        ... modify data structure
        return *this; // to modified object
    }
};

my_collection.insert (x1).insert (x2);
```
Static members

Need to have computable attributes for class itself, independent of specific object; e.g. number of objects created.
Static qualifier indicates that entity is unique for the class

```cpp
static int num_objects = 0;
point () { num_objects++;} // ditto for other constructors
```
Can access static data using class name or object name:

```cpp
if (point.num_objects != p1.num_objects) error ();//
```
Classes and private types

If all data members are private, class is identical to a private type: visible methods, including assignment.

A struct is by default a class with all public members

How much to reveal is up to programmer

define functions to retrieve (not modify) private data

```cpp
int xcoord () { return x;};
int ycoord () { return y;};
p2.x = 15; // error, data member x is private
```
Destructors

If constructor allocates dynamic storage, need to reclaim it

```cpp
class stack {
    int* contents;  int sz;
public:
    stack (int size) { contents = new int [ sz = size];};
    void push ();
    int pop ();
    int size () { return sz;}; };

stack my_stack (100);    // allocate storage dynamically
// when is my_stack.contents released?
```
If constructor uses resources, class needs a destructor

User cannot deallocate data because data member is private: *system must do it*

```cpp
~stack () {delete[ ] contents;};
```

- inventive syntax: negation of constructor
  Called automatically when object goes out of scope
  Almost never called explicitly
Copy and assignment

point p3 (10,20);
point p5 = p3;  // componentwise copy

This can lead to unwanted sharing:
stack stack1 (200);
stack stack2 = stack1;  // stack1.contents shared
stack2.push (15);  // stack1 is modified

Need to redefine assignment and copy
Copy constructor

stack (const stack& s) {  // reference to existing object
    contents = new int [ sz = s.size()];
    for (int l = 0; l <sz; l++) contents [l] = s.contents [l];
}

stack s1 (100);
...

stack s2 = s1;  // invokes copy constructor

Not to be confused with an assignment!
Redefining assignment

- assignment can also be redefined to avoid unwanted sharing
- operator returns a reference, so it can be used efficiently in chained assignments: one = two = three;
  stack & operator = (const stack& s) {
    if (this != &s) { // beware of self-assignment
      delete [] contents; // discard old value
      contents = new int [sz = s.size ()];
      for (int k = 0; k < sz; k++) contents [k] = s.contents [k];
    }
    return *this; }
  stack s1 (100), s2 (200); … s1 = s2; // transfer contents
Anomalies

An array whose component type is a class can only be declared if there is a parameterless constructor for the class. There is no way to pass parameters to the constructor.

```plaintext
polygon point [10]; // ok
turing stack [2];    // illegal
```