CONSTRUCTORS AND DESTRUCTORS

Today

- Today we will look *constructors* and *destructors*.
- These are important additional concepts in handling classes and objects.
- We will also briefly cover *polymorphism* and *overloading*, and mention friend classes, composition and derivation.
- This material is taken from Pohl, Chapter 5, mainly 5.1–5.3, 5.7 and 5.10.

ctors and dtors

- An *object* is a *class instance*.
- House metaphor: the blueprint for the house is like a class; the constructed house is like an object).
- The allocation of memory to create (instantiate) an object is called *construction*; freeing memory (aka deallocation) when the program is done using the object is called *destruction*.
- A *ctor* (*constructor*) is a member function used to allocate the memory required by an object.
- A constructor always has the same name as the class it constructs.
- A *dtor* (*destructor*) is a member function used to deallocate (free) the object's memory, after the object is no longer needed.

- There are two ways to invoke the ctor and dtor.
- A constructor is invoked when:
  - An object is declared.
  - An object is created using the C++ keyword `new`.
- A destructor is invoked when:
  - Program execution reaches the end of the block of code in which the object was created.
  - The object is deleted using the C++ keyword `delete`.
- Constructors can be *overloaded* (i.e., programmers can write their own versions); destructors cannot.
- Constructors can take arguments; destructors cannot.
- ctors and dtors do not have data types; they do not return values.

ctor and dtors for "point"

- Here's our old friend `point`.

```
class point {
private:
  double x, y;
public:
  // These are constructors
  point() { x = 0; y = 0; }
  point( double u ) { x= u; y = 0; }
  point( double u, double v ) { x = u; y = v; }
  // End of contructors
  void print() const;
  void set( double u, double v );
};
```

- You can find an example that is very much like this in `point-with-constructor.cpp`.

Constructor details

- All constructors have the same name as the class (`point` in this case) and have no return type.
- The default constructor.
  - The default constructor is the one that takes no arguments.
  - If you don't define one, the system creates the default.
  - You can overload the default constructor with or without arguments of your own.
- Constructor initializer.
  - You can use a constructor to initialize class data members.
  - This is the main reason for having constructors.

- A constructor is called when you create an instance of a class.
- Given the definition above,

  `point p;`

  will create a point object, called `p` with its data members set to 0;
- Similarly the call:

  `point p(1);`

  will create a point object with its `x` value set to 1 and its `y` value set to 0;
- while:

  `point p(3, 4);`

  will create a point object with its `x` value set to 3 and its `y` value set to 4;

- Constructors have a special syntax for initialising variables.
- For example, instead of:

  `point::point( double u ) { x = u; }`

  you can use a constructor initializer like this:

  `point::point( double u ) : x(u) { }`

  and instead of:

  `point::point( double u, double v ) { x = u; y = v; }`

  you can use:

  `point::point( double u ) : x(u), y(v) { }`
- The syntax is as follows:

  member-name (expression-list),member-name (expression-list)

  where each member is initialized to the expression in parenthesis

Conversion constructors

- Constructors can be used to convert data from one type to another.
- For example (in program `printChar.cpp`) :

```
class pr_char {
private:
  int c;
  static const char* rep[5];

public:
  pr_char( int i=0 ) : c( i % 5 ) { }
  void print() const { cout << rep[c]; }
};
```

- The constructor here performs a conversion from integer to `pr_char`.

- The conversion constructor makes it possible to write:

```
for ( int i=0; i<5; i++ ) {
  c = i; // NOTE how this is done
  c.print();
}
```

- Having conversion constructors isn't necessarily good practice.
- It only works where the constructor is initializing one data element.
- By default, *any* constructor with a single argument is assumed to be a conversion constructor.

- To control this, we use the keyword `explicit`.
- Placing this in front of a constructor definition tells the compiler that is isn't safe to allow the constructor to be used for conversion:

```
explicit charStack( int size ) : max_len(size), top(EMPTY)
          { s = new char[size];}
```

- Example comes from `stack-with-ctors.cpp`

Another constructor example

- Example from book:

```
class counter {
private:
int value; // 0 to 99
public:
counter( int i ); // ctor declaration
void reset() { value = 0; }
int get() const { return value; }
void print() const { cout << value << '\t'; }
void click() { value = (value+1) % 100; }
}
// constructor definition:
inline counter::counter( int i ) { value = i % 100; }
```

- `inline` is (another) new keyword.
- It means that the compiler can try to replace the function call by the function body code; this avoids function call invokation and can speed up program execution;
- `inline` isn't required here, nor is it required by constructors in general

Copy constructors

- This is a somewhat complicated detail that has to do with what happens when an object is used as a call-by-value argument to a function.
- We mentioned briefly about the use of the run-time stack and how memory is allocated and deallocated when functions are called.
- When the arguments to functions are primitive data types (e.g., `int`), then this is easy.
- But when the arguments to functions are objects, what happens locally inside the function? how is a "local copy" made for use inside the function?.
- This is where a *copy constructor* is needed.

- This is defined by using a call-by-value argument to a version of a constructor
- For example:

```
charStack::charStack( const charStack& stk )
  : top(stk.top), FULL(stk.FULL), length(stk.length) {
   stack = new char[stk.length];
   memcpy(stack, stk.stack, length);
}
```

- This is another example from `stack-with-ctors.cpp`.
- Copy constructors are typically needed when the objects being copied have data members that are pointers.
- The signature for a copy constructor of class myClass will always be `myClass( const myClass& )`

Destructors

- Defined as the name of the class preceded by a tilde (∼)
- The default destructor will delete an object when the program reaches the end of the scope of that object (block where it is declared).
- You can write your own destructor to free up additional memory used by the object.
- Example, free up the array used by the stack:

```
class charStack {

~charStack() { delete []stack; }

}
```

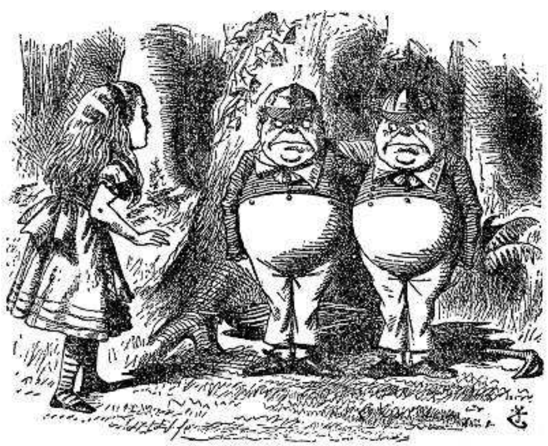
- Again, this is in `stack-with-ctors.cpp`.

Polymorphism and overloading

- *polymorphism*—giving different meanings to the same function or operator, i.e., "having many forms". Lets us use different implementations of a single class
- *overloading*—creating new versions of functions with the same or different arguments
- When you overload a function, the name of the function is the same, but what is does is different from the default
- Operators can also be overloaded
- *signature matching* is what the compiler uses when there are multiple versions of a function (or operator) to determine which version should be invoked
- Textbook goes into a LOT of detail about this—we'll come back to it more later in the semester.

Friend classes

- Allows two or more classes to share private members and functions
  - e.g., container and iterator classes
- Friendship is not transitive.
- Since friendship violates the usual rules about hiding members, you need to use it with care.
- In fact you should try *not* to use `friend`.
  - When writing code from scratch you should be able to avoid it.
  - It tends to be used when quickly patching code.




```
class tweedledee {
...
  friend class tweedledum;

  int cheshire();
...
};
```

- This allows any instance of `tweedledum` to access any member of any instance of `tweedledee`.
- However no instance of `tweedledee` can access any private member of `tweedledum`.

Friend functions

- Friendship can also be at the individual function level.
- A non-member friend function can have access to the private components in a class.
- Extending the previous example:

```
void alice() {
...
}

class tweedledum {
...
  friend void alice()   // prototypes for friend functions
  friend int tweedledee::cheshire ();
...
};
```


- This allows `alice` and `cheshire` to access the data in instances of `tweedledum`.
- For concrete example see the program `robots.cpp`
- If this example seems contrived, that's because it is :-)
- `friend` is like that — unless you realy need it, it seems rather superfluous.

Hierarchy with composition and derivation

- Composition:
  - Creating objects with other objects as members
- Derivation:
  - Defining classes by expanding other classes

```
class roomba: public robot {
private:
  string type;

public:
  void setType(string s);
  void vacuum(double x, double y);
};
```

- Like "extends" in java.

- "Base class" (`robot`)
- "Derived class" (`roomba`)
- Complete example in `robots.cpp`
- Derived class can only access *public* members of base class
- `public` vs `private` derivation:
  - `public` derivation means that users of the derived class can access the public portions of the base class
  - `private` derivation means that all of the base class is inaccessible to anything outside the derived class
  - `private` is the default

Derivation and friendship

- Friendship is not the same as derivation!
- Example:
  - $b2$ is a friend of $b1$
  - $d1$ is derived from $b1$
  - $d2$ is derived from $b2$
- In this case:
  - $b2$ has special access to private members of $b1$, as a friend
  - But $d2$ does not inherit this special access
  - Nor does $b2$ get special access to $d1$ (derived from friend $b1$)
- `arrays.cpp` gives a more interesting example than `robots.cpp`, but you need to be comfortable with pointers.
- We'll talk about derivation more later in the course.

Summary

- This lecture has looked at:
  - Constructors and destructors
  - Polymorphism, overloading
  - Friends
  - Composition and derivation
- For most of these topics, it has been a first look; we will come back to them over and over again through the semester.