CONSTRUCTORS AND DESTRUCTORS

Today

- Today we will look *constructors* and *destructors*.
- These are important additional concepts in handling classes and objects.
- This material is taken from Pohl, Chapter 5, mainly 5.1–5.3 and 5.6.

Constructors (ctors)

- 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.

- There are two ways to invoke the constructor.
- A constructor is invoked when:
  - An object is declared.
  - An object is created using the C++ keyword `new`.
- Constructors can take arguments.
- Constructors can be *overloaded*, that is given different combinations of parameters.
  - Compiler distinguishes based on *signature*.
- This means programmers can write their own versions, possibly many different versions.
- Constructors do not have data types; they do not return values.

ctor 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`.
- The class implements a fancier version of the stack from the last lecture and uses pointers.

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; }
```

- This constructor sets the value of the variable `i` after doing some manipulation of its value.
- `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.
- A copy constructor says how to set the members of a copy.

- For example, a copy constructor for `point` would be:

```
point::point( const point& pt){
  x = pt.x;
  y = pt.y;
}
```

- This says that to make a copy of `point`, you need to set the variables `x` and `y` of the copy to have the values of the `x` and `y`.
- The thing being copied is the argument `pt`.
- The signature for a copy constructor of class myClass will always be `myClass( const myClass& )`
- Now, this is a rather silly copy constructor, since C++ will make a copy of point fine without a copy constructor.
- Copy constructors are typically needed when the objects being copied have data members that are pointers.

- A better (but more complicated) example is a copy constructor for the class from the file `stack-with-ctors.cpp`.
- The copy constructor is:

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

- Since copy constructors are only really needed when we have classes with things like pointers in them, don't worry about them too much for now.
- They will make more sense when we have covered pointers.

Destructors (dtor)

- A *dtor* (*destructor*) is a member function used to deallocate (free) the object's memory, after the object is no longer needed.
- Defined as the name of the class preceded by a tilde ($\sim$)
- 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.
- Typically you don't need to do this until your objects are making use of *dynamic memory allocation* which we won't get to until next lecture.

- Example, free up the array used by the stack:

```
class charStack {

~charStack() { delete []stack; }

}
```

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

- There are two ways to invoke the destructor:
  - 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`.
- Destructors cannot be overloaded
- Destructors cannot take arguments.
- Dtors do not have data types; they do not return values.

Summary

- This lecture has looked at constructors and destructors.
- Constructors are used for initialisation and other operations that must take place when an object is created.
- We learnt that a class can have many constructors, and that they are distinguished form each other by their *signature*.
- A class may have a copy constructor and a conversion constructor.
- A class only ever has one destructor. It frees up memory when an object is destroyed.