CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 6
September 17, 2018

Construction, Initialization, and Destruction

Initializing an object

Whenever a class object is created, one of its constructors is called.

This applies not only to the “outer” object but also to all of its embedded objects.

If not specified otherwise, the default constructor is called, if defined. This is the one that takes no arguments.
Example: `MyClass mc;` calls default constructor `mc`.

If you do not define any constructors, then the default constructor is defined automatically to be the null constructor.

Which constructor gets used?

A class can have several constructor methods, differing from each other in the number and types of arguments.

When an object is created, the constructor called is the one matching the user-specified arguments.

For example, suppose the user declares two `Parallelogram` objects:

```
Parallelogram tempShape;
Parallelogram yellowShape( 5, 5, 30 );
```

`tempShape` is initialized by calling the null constructor.
`yellowShape` is initialized by calling `Parallelogram(5, 5, 30)`.

Construction rules for a simple class

The rule for constructing an object of a simple class is:

1. Call the constructor/initializer for each data member, in sequence.
2. Call the constructor for the class.

Construction rules for a derived class

The rule for constructing an object of a derived class is:

1. Call the constructor for the base class (which recursively calls the other constructors needed to completely initialize the base class object.)
2. Call the constructor/initializer for each data member of the derived class, in sequence.
3. Call the constructor for the derived class.

Destruction rules

When an object is deleted, the destructors are called in the opposite order before the storage allocated to the object is released back to the system.

The rule for an object of a derived class is:

1. Call the destructor for the dervied class.
2. Call the destructor for each data member of the derived class in reverse sequence.
3. Call the destructor for the base class.

Rules for a simple class are the same except that step 3 is omitted.

Constructor ctors

Ctors (short for constructor/initializors) allow one to supply parameters to implicitly-called constructors.

Example:

```
class Deriv : Base {
  Deriv( int n ) : Base(n) {};
      // Calls Base constructor with argument n
};
```

Initialization ctors

Ctors also can be used to initialze primitive (non-class) variables.

Example:

```
class  Deriv {
  int x;
  const int y;
  Deriv( int n ) : x(n), y(n+1) {}; //Initializes x and y
};
```

Multiple ctors are separated by commas.

Ctors present must be in the same order as the construction takes place – base class ctor first, then data member ctors in the same order as their declarations in the class.

Initialization not same as assignment

Previous example using ctors is not the same as writing
`Deriv( int n ) { y=n+1; x=n; };`

- The order of initialization differs.
- `const` variables can be initialized but not assgined to.
- Initialization uses the constructor (for class objects).
- Initialization from another instance of the same type uses the copy constructor.

Special member functions

A class has six special member functions. These are special because they are defined automatically if the programmer does not redefine them. They are distinguished by their prototypes.

Name	**Prototype**
Default constructor	`MyClass();`
Destructor	`~MyClass();`
Copy constructor	`MyClass( const MyClass& other );`
Move constructor	`MyClass( MyClass&& other );`
Copy assignment	`MyClass& operator=( const T& other );`
Move assignment	`MyClass& operator=( T&& other );`

Special function automatic definitions

Name	**Automatic Definition**
Default constructor	Null constructor does nothing;
Destructor	Function that does nothing
Copy constructor	Does a shallow copy from its argument
Move constructor	(later)
Copy assignment	Does a shallow copy from rhs to lhs
Move assignment	(later)

Copy and assignment have the same default semantics but can be redefined to behave differently.

Deletion

Some of the automatic definitions are omitted if certain special functions are defined by the user.

For example, if you define a constuctor with arguments, then the default constructor is automatically deleted.

You can explicitly remove any automatically-created special function by using `=delete` in place of a definition.

Example: To remove the copy constructor for `MyClass`, write
`MyClass( const MyClass& ) = delete;`

Restoration of automatically deleted definition

If a default definition for a special function is automatically deleted, it can be brought back using `=default` in place of a definition.

For example, if you define a constuctor with arguments, then the default constructor is automatically deleted.

To bring it back, you can write `MyClass() = default;`.

Copy constructors

- A copy constructor is automatically defined for each new class `MyClass` and has prototype `MyClass(const MyClass&)`. It initializes a newly created `MyClass` object by making a shallow copy of its argument.
- Copy constructors are used for call-by-value parameters.
- Assignment uses `operator=()`, which by default copies the data members but does not call the copy constructor.
- The results of the implicitly-defined assignment and copy constructors are the same, but they can be redefined to be different.

Move constructors

C++11 introduced a move constructor. Its purpose is to allow an object to be safely moved from one variable to another while avoiding the “double delete” problem.

We’ll return to this interesting topic later, after we’ve looked more closely at dynamic extensions.

Reference Types

Reference types

Recall: Given `int x`, two types are associated with `x`: an L-value (the reference to `x`) and an R-value (the type of its values).

C++ exposes this distinction through *reference* types and declarators.

A *reference type* is any type `T` followed by `&`, i.e., `T&`.

A reference type is the internal type of an L-value.

Example: Given `int x`, the name `x` is bound to an L-value of type `int&`, whereas the values stored in `x` have type `int`

This generalizes to arbitrary types `T`: If an L-value stores values of type `T`, then the type of the L-value is `T&`.

Reference declarators

The syntax `T&` can be used to declare names, but its meaning is not what one might expect.

```
int x = 3;     // Ordinary int variable
int& y = x;    // y is an alias for x
y = 4;         // Now x == 4.
```

The declaration must include an initializer.

The meaning of `int& y = x;` is that `y` becomes a name for the L-value `x`.

Since `x` is simply the name of an L-value, the effect is to make `y` an alias for `x`.

For this to work, the L-value type (`int&`) of `x` must match the type declarator (`int&`) for `y`, as above.

Use of named references

Named references can be used just like any other variable.

One application is to give names to otherwise unnamed objects.

```
int axis[101];              // values along a graph axis
int& first = axis[0]   ;    // give name to first element
int& last = axis[100];      // give name to last element
first = -50;
last = 50;

// use p to scan through the array
int* p;
for (p=&first; p!=&last; p++) {...}
```

Reference parameters

References are mainly useful for function parameters and return values.

When used to declare a function parameter, they provide call-by-reference semantics.

```
int f( int& x ){...}
```

Within the body of `f`, `x` is an alias for the actual parameter, which must be the L-value of an `int` location.

Reference return values

Functions can also return references.

```
int& g( bool flag, int& x, int& y ) {
    if (flag) return x;
    return y;
}
...
g(x<y, x, y) = x + y;
```

This code returns a reference to the smaller of `x` and `y` and then sets that variable to their sum.