Kevin Chen

11/21/15

Class Notes

Classes

* Syntax:

class class\_name {

access\_specifier\_1:

member1;

access\_specifier\_2:

member2;

...

} object\_names;

* Access specifier: private (only accessible by other members of the same class, aka “friends”), protected (only accessible by friends and members of their derived classes), and public (accessible anywhere this object is visible).
  + By default, members of class with “class” keyword are private.
* Can define a member function outside of the class by using the scope operator (::, two colons)
  + Put class name on the left side of the scope operator, and member name on right.
  + Defining function inside the class is *inline* function, while defining outside is normal (not-inline) function. No differences in behavior.
* Object Orientated programming reduces need for passing in parameters.
* Instantiating syntax. Example: Rectangle r(3, 4). (Rectangle is a class that has a constructor that takes in two ints for width and height.)
* Constructors: don’t have return type. Constructor name is class name.
  + Cannot use parenthesis to create instance using default constructor (the constructor that does not take in any parameters). Using parenthesis would make it a function declaration instead of an object declaration.
* Assignment initialization (only for constructors with a single parameter) syntax:
  + [class name] [object name] = [initialization value];
* Uniform initialization syntax:
  + [class name] [object name] = {[value], [value], [value], …}
  + Equals sign above is optional.
* Member initialization in constructors: constructor can initialize variable directly (before the body) using a colon. Rectangle example:

Rectangle::Rectangle (int x, int y) : width(x), height(y) { }

* + Can also use uniform initializer syntax (brackets instead of parenthesis).
  + width(x), height(y) is referred to as the member initializer list.
  + If member objects (those whose type is *class*) are not initialized after the colon (aka the member initializer list), they are default-constructed. If there isn’t a default constructer, then there is an error.
* Pointers to objects can access the objects instead of the just using the object name. Can use arrow operator (->) to access the member of an object.
* structs can also be used to declare classes with member functions. Only difference is that members of structs by default are public.
* unions are also classes. Can only store one data member at a time, but can also hold member functions.
* Overloading operators: example: you can add two objects together. Can be done using operator function.
  + Syntax is: [return type] operator [sign] (parameters) { /\*... body ...\*/ } (optionally no space between operator and sign.)
  + Member operator has an implicit parameter (*this* parameter), so it needs one less parameter than non-member operator. Thus, two-operand operators need 1 parameter; one-operand operators do not need any parameters.
  + “=” is automatically overloaded by the generator. Assigns all fields together and returns the object (*this*).
* Static members: also known as class variables, they must be initialized outside of class.
  + Syntax: Add the word “static” before the type. Example: static int n;
  + Can be referred through any object of the class or directly through class name.
  + Static functions cannot access non-static members. Can access instance of non-static members.
* Const instances: Cannot change fields anytime except during construction.
  + Const instances cannot access non-const methods within the class. Const member functions have “const” before the body. Example:

int get() const {return x;}

* Can return a const type. Specify const at beginning of method. Don’t get this confused with “const” member function which appears before body.
  + const member functions cannot modify non-static data members nor call non-const member functions.
  + Member functions can be overloaded on their constness: one class can have two member functions with identical signatures except one is const and the other is not.
* Class templates: classes within templates allow classes to have members that use template parameters as types. template header (“template <class [type name]>”) comes before the word “class”.
  + Create an instance: [class name]<[template type]> [object name] ([constructor parameters]);
    - Example:

mypair<int> myobject (115, 36);

* + Defining a member function outside of the class template:

[template header]

[return type] [class name]<[template type>::[member function name] ([parameters])

* + - Example:

template <class T>

T mypair<T>::getmax ()

* Template specialization: allow different behavior to be defined for particular type. After defining normal class template, create another template with same name but add <[specialization type parameter]> before body and remove the class template parameter in the original header.
  + Example:

template <> class

mycontainer <char> {…}

* + Must copy all members into specialized template. None of the members are inherited from generic template.

Special Members

* Special member functions are member functions that are implicitly defined as member of classes under certain circumstances.
* Destructor: deletes allocated dynamic memory. Add a “~” before class name method in order to create a destructor.
  + Destructors are called for an object at the end of its lifetime.
* Copy constructor: a constructor that constructs a copy of its own object. First parameter is the object that you want a copy of. Example (for a class called MyClass):
  + MyClass::MyClass(const MyClass& x) : a(x.a), b(x.b), c(x.c) {} //Also called a shallow copy: only copy the member of the copies themselves
  + Example of problem with shallow copy is pointers. Two pointers to same exact object is not what we want. Deep copy solves this.
  + Deep copy: new copy objects are created for any referenced objects
* Copy assignment: objects can be copied not only during construction, but copied via copy assignment. It is an overload of operator=.
  + Implicit version only performs shallow copy.
  + Potential problems: not only two pointers to same memory, but also memory leaks occur: not deleting previous object pointed by pointer.
  + Return value is generally a reference to \*this.
* Move Constructor and Assignment: similar to copy, except source loses its content.
  + Value that is being moved needs to be an unnamed object. Example: return value, type-cast.
  + Must take in *rvalue reference* to the class itself. (rvalue reference is specified by &&.)
  + MyClass (MyClass&&); // move-constructor
    - Can be called like so: MyClass baz = fn(); // where fn() returns a MyClass object
  + MyClass& operator= (MyClass&&); // move-assignment
    - Can be called like so; baz = MyClass();
  + Only copies pointer to object memory. Copy assignment needs to allocate new memory.
  + “Return value optimization” optimizes move-construction.
* Implicit members

|  |  |  |
| --- | --- | --- |
| **Member function** | **implicitly defined:** | **default definition:** |
| [Default constructor](http://www.cplusplus.com/doc/tutorial/classes2/#default_constructor) | if no other constructors | does nothing |
| [Destructor](http://www.cplusplus.com/doc/tutorial/classes2/#destructor) | if no destructor | does nothing |
| [Copy constructor](http://www.cplusplus.com/doc/tutorial/classes2/#copy_constructor) | if no move constructor and no move assignment | copies all members |
| [Copy assignment](http://www.cplusplus.com/doc/tutorial/classes2/#copy_assignment) | if no move constructor and no move assignment | copies all members |
| [Move constructor](http://www.cplusplus.com/doc/tutorial/classes2/#move) | if no destructor, no copy constructor and no copy nor move assignment | moves all members |
| [Move assignment](http://www.cplusplus.com/doc/tutorial/classes2/#move) | if no destructor, no copy constructor and no copy nor move assignment | moves all members |

* + Each class can select explicitly which of these members exist with their default definition or which are deleted by using the keywords “default” or “delete”.
  + Syntax: function\_declaration = default; function\_declaration = delete;

Friendship and inheritance

* Friend functions: allows private and protected members of a class to be accessed outside the class.
  + Specify in the class that a function is a friend by copying the friend function’s signature and adding the keyword “friend” before.
  + Friend functions are not member functions.
* Friend classes: Copy friend class’s declaration and add “friend” before.
  + No converse: if class A is friend of class B, class B is NOT friend of class A.
  + Not transitive: if class A is friends with class B and class B is friends with class C, class A is NOT friend with class C.
* Inheritance
  + Syntax: class derived\_class\_name: access\_specifier base\_class\_name { /\*...\*/ };
  + access\_specifier indicates the least restrictive level of access the inherited members will become.
  + Derived class can access protected, but not private members of base class.
  + If no access level is specified, private is default for classes and public is default for struct.
* What is inherited: EVERYTHING except constructors and its destructor, assignment operator members (operator=), friends, and private members
  + But derived classes automatically call base class’s constructor/destructor.
  + Unless otherwise specified, constructor of derived class calls default constructor of base class. Calls it before running code of constructor of derived class.
  + Calling a different constructor syntax:

derived\_constructor\_name (parameters) : base\_constructor\_name (parameters) {...}

* Multiple inheritances: list out inherited classes separated by commas.

Polymorphism

* Pointers to base class
  + A pointer to a derived class is type-compatible with a pointer to its base class.
  + The pointer can access the base class’s members, but not the derived class’s members.
* Virtual members (allows overwriting methods)
  + A member function that can be redefined in a derived class, while preserving its calling properties through references.
  + Syntax: add “virtual” before its declaration. Example:

virtual int area ()

{ return 0; }

* + A class that declares or inherits a virtual function is called a *polymorphic class*.
* Abstract base classes
  + Have virtual member function(s) without definition (aka *pure virtual functions*)
  + Syntax: replace body with “=0”, and add a semicolon at the end.
    - Example: virtual int area () =0;
  + Cannot be used to instantiate objects, but still can have pointer to it to access the virtual function’s return value.