Chapter 10 Objects and Classes

10.1 An overview

- Procedural approach: you first concentrate on the procedures you will follow and then think about how to represent the data.
- OOP approach: you concentrate on the object as the user perceives it , thinking about the data you need to describe the object and the operations that will describe the user's interaction with the data. After you develop a description of that interface, you move on to decide how to implement the interface and data storage.

  > OOP (object-oriented programming) is a programming paradigm. There are other paradigms as well, a topic covered by theoretical computer science.

- *Encapsulation*: gathering the implementation details together and separating them from the abstraction. A concept championed by OOP.
- Classes as a user-defined types
  - A type does three things:
    - It determines how much memory in needed for a data object.
    - It determines how the bits in memory are interpreted (e.g. A `long` and `float` may have the same number of bits, but they are translated into different numeric values. The same goes for `unsinged int` and `int`, etc.)
    - It determines what operations, or *method*s, can be performed using the data object (e.g. `int` supports arithmetic operations, but does not support dereferencing).
  - A class is a user-defined type, and it does the same jobs as listed above.
  - A class specification has two parts:
    - A *class declaration*, which describes the data component, in terms of *data member*s, and the operation component, in terms of *member function*s or *method*s.

      > Data member are usually not public; this insulation of data from direct access by a program is called *data hiding*, an instance of *encapsulation*.
      >
      > Calling a member function with an object is termed *sending a message* to that object, which is implicitly passed to the function as an argument.

      ```
      1  class Stock{
      2  private:
      3    char company[30]; int shares;
      4    double shareVal; double totalVal;
      5    void set_total() { totalVal = shares * shareVal;}
      6  public:
      7    void acquire(const char *co, int n, double pr);
      8    void buy(int n, double price); void sell(int n, double price);
      9    void update(double price); void show();
      10 }; // a semicolon is needed, like structures, unlike namespaces.
      ```

        - The *class method definition*s, which describe how certain class methods are implemented. The function headers use their qualified names because a class member has *class scope*.

            > Within the definition, however, qualified names are unnecessary if two members are in the same class scope.

            ```
            void Stock::Stock();       // default constructor
            void Stock::Stock(char *c, int n, double p) {...}; // constructor
            void Stock::acquire(const char *co, int n, double pr) {...}
            void Stock::buy(int n, double price) {...}
            ...
            void Stock::~Stock() {...} // destructor
            ```

        - In common practice, class declarations are stored in headers, and method definitions are stored in separate source files.

            > This programming idiom is also applied to namespaces.
            >
            > Placing function definitions in source files instead of in headers, as discussed in Chapter 9, has two purposes: (1) separate compilation, and (2) abiding by the one definition rule.

    - The default access control for a class member, be it a data member or a member function, is `private`.
    - *Access control labels* are `private` (by default, so it is not necessary), `protected`, and `public`.
- Assigning an object to another is allowed: `objB = objD;`, provided that they are of the same class, OR the class of the assigner `objD` is derived from the class of the assignee `objB`.
- Inline methods are allowed. Note that inline functions MUST be defined in each file in which they are called. Therefore there are two ways to make a method `inline`:
    - If a method definition is placed where the method declaration should be inside the class declaration, then the method is automatically inline.
    - Also you can place the method definition outside, and prefix the definition with `inline`. But the definition should be in the same header in which the class is declared.

        ```
        // file: Amber.h
        class Amber{
          ...
          void func();
        };
        inline void A::func() {...}
        ```

- `mutable` data member is allowed in a class, as with a structure. A data member declared as `mutable` can always be modified, even if the object it belongs to is `const`.

    ```
    class A {private: int a; public: mutable int b; ...};
    const A obj;
    obj.a = 1;  // error. Because obj is const
    obj.b = 1;  // ok. Because obj.b is mutable
    ```

- C++ regards a structure exactly as a class except that the members are `public` by default. Therefore, a structure type may also have access control labels, member functions, fancy mechanisms like friendship and inheritance, etc.

10.2 Class constructors and destructors

- Call constructors to create objects

```
// call the default constructor
Stock myStock1;                 // implicit call, no "()"
Stock *myStock2 = new Date;     // implicit call, no "()"
Stock myStock3 {};              // implicit call, list-init.
Stock myStock4 = Stock();       // explicit call
// MATLAB: myStockml = Stock();     Python: myStockpy = Stock()
```

```
// call the non-default constructor
Stock hisStock1("Warner Bro.", 50, 4.9);              // implicit call
Stock *hisStock2 = new Stock("Warner Bro.", 50, 4.9); // implicit call
Stock hisStock3 {"Warner Bro.", 50, 4.9}; // implicit call, list-init.
Stock hisStock3 = Stock("Warner Bro.", 50, 4.9);      // explicit call
// MATLAB: hisStockml = Stock(...);  Python: hisStockpy = Stock(...)
```

  In addition, C++11 offers a class called `std::initializer_list` that can be used as a type for a function or method input argument. This class can represent a list of arbitrary length, providing all the entries are of the same type or can be converted to the same type. Chapter 16 will return to this topic.

- Constructors
  - Constructors have exactly the same name as the class. They can be overloaded. Like any other methods, you declare their prototypes in the class declaration if you want to define them.
  - Constructors has no return value and therefore no return type (NOT `void`). You should NOT place any type identifier in the front of its prototype or function header.

```
class dog{
  ...
public:
  dog();
  dog(char *name);
  void bark(double sonority);
};
```

  - Constructors are used to allocate memory, and initialize them if necessary.
  - A *default constructor* is a constructor that is used to create an object when you do not explicitly provide initialization values. A default constructor can have no arguments or else it must have default values for all arguments.

```
Noodles::Noodles();
Dumplings::Dumplings(int num = 10, char *filling = "veg");
```

- The compiler automatically provides a default constructor with no arguments, if and only if you do not declare ANY constructors.
  - If you provided a constructor, be it a default one or not, the compiler does not bother helping you. You have to manually provide a default constructor, or there will be a compile error if you try to call the default constructor.
  - It is recommended that you should always provide a default constructor that implicitly initializes all class members, for the sake of program safety. Do not rely on the compiler's helping hand in this regard.
- A *non-default* constructor, by its literal meaning, is a constructor that explicitly provides at least one initialization value. Default arguments are also supported.
- A constructor that you can use with a single argument allows you to use assignment syntax to initialize an object like this: `ClassName object = value;`. This features can cause problems, but it can be blocked, as described in Chapter 11.
- C++ also provides a special syntax for constructors that can be used to initialize data members. This syntax becomes necessary when initializing (1) reference data members, which cannot be assigned because they are references, AND (2) (non-`static`) `const` data members, which cannot be assigned due to their `const`-ness, AND (3) data members of a class type that does not have a default constructor, because all data members are first initialized to a default value and then assigned the given value. Plus, class inheritance may need it, too (see 13.1).
  - The syntax is termed *constructor initializer list*. (*initialize* $\neq$ *assign*)
  - This syntax consists of a colon followed by a comma-separated list of initializers. This is placed after the closing parenthesis of the constructor arguments and before the opening brace of the function body.
  - Each initializer consists of the name of the member being initialized followed by parentheses containing the initialization value (do NOT include the type).
  - These initializations take place when the object is created and before any statements in the constructor body are executed.

    ```
    ClassName::ClassName(ALL_arguments) : data1(val1), ..., ...
    { /* take care of the rest of the arguments */ }
    ```

  - It is recommended to use constructor initializer lists rather than assignment in constructors. Because (1) it is more efficient now that all data members are first initialized to a default value and then assigned, and (2) routinely using this syntax prevent you from being surprised by compile errors when you have class member that cannot be assigned.
- C++11 allows in-class initialization for data members, excluding `static` data members which are not of `const` integral types.

  ```
  class A{private: int n = 0; ... public: char *c = NULL; ...};
  ```

- You can use constructors to declare an array of objects.

```
Stock stocks[2] = {
  Stock(),                              // use default constructor
  Stock("Monolithic Inc.", 100, 4.99) // use non-defualt constructor
};
```

- About a class's *copy constructor*, see 12.3. It is provided automatically if there is no a copy constructor.
- About a class's *move constructor*, see Chapter 18. It is provided automatically when some conditions are met. A class does not necessarily have one.

- Destructors
  - A destructor's name is the name of the class preceded by a tilde `~`. It cannot be overloaded. Like any other methods, you have to declare it in the class declaration if you want to define it.
  - A destructor has no return value and therefore no return type (NOT `void`). And unlike constructors, a destructor has no input arguments (the argument list is an empty parentheses pair).

```
class cat{
  ...
public:
  void meow(double frequency);
  ~cat();
};
```

  - A destructor bears the responsibility of clearing up any debris when an object expires. For example, if the constructor uses `new`, then the destructor should use `delete`.
  - The compiler automatically provides a destructor for each class if and only if you do not provide one. In practice, this task is usually left to the compiler.
    - However, a user-defined destructor that uses `delete` / `delete[]` in its definition becomes necessary if the constructor uses `new` / `new[]`. Therefore you should be careful of using dynamic memory allocation inside a constructor.

      > Side note: it is ok to use either `delete` or `delete[]` with a null pointer (`0`, or with C++11, `nullptr`).

  - It is at the compiler's discretion to decide when to call the destructor if you do not explicitly call it.
    - If you create an automatic storage class object, the destructor is called automatically when the program exits the code block in which the object is declared.

      > Automatic storage variables are stored in the stack, where the memory is managed in a FILO fashion. Therefore, as with non-class variables, objects are destroyed in the reverse order of their creating.

    - If you create a static storage class object, the destructor is called automatically when the program terminates.
    - If you create a dynamic storage class object (using `new`), the destructor is called when you use `delete` to free the memory occupied by the object.

- A program might create temporary objects to carry out certain operations; in that case, the destructor is called automatically when the program finishes using these objects.
- You can explicitly (manually) call the destructor to destroy a class object, e.g. `myStock1.~Stock();`. But the object will be destroyed automatically again as planned - it will cause a runtime error if the destructor has `delete` as you cannot apply the operator to the same block of memory more than once.

10.3 `const` member functions

- From the previous chapters we know that a `const` variable can be passed to a function ONLY IF the corresponding formal argument is also `const`. The `const`ness of the formal argument makes an guarantee that the function will not try to modify the argument's value.

```
1  //prototypes
2  void calc(int a);  void calc_c(const int a);
3  // call
4  int x = 1;  const int y = 1;
5  calc(x);  calc_c(x);  calc_c(y);    // ok
6  calc_c(x);   // error. Invalid conversion from 'contst int' to 'int'
```

- With that in mind, we can understand that the following code snippet is invalid because the method `show()` fails to guarantee that the input argument (the object `myStock`) is `const` - this method has no argument list for `const` to qualify at all. The object it uses is provided implicitly by the method invocation.

```
1  const Stock myStock = Stock("Kustar Architecture", 100, 5.0);
2  myStock.show();   // error.
```

- To get around the problem, C++ allows you to place the `const` qualifier after the member function's parentheses. This way the method is `const` and therefore can be applied to a `const` invoking object.

```
1  // prototype (inside the class declaration)
2  void show() const;
3  // definition
4  void Stock::show() const {...}
```

> In short, the `const` inside the argument list states that the function will not modify the explicitly accessed argument. The `const` that follows the parentheses states that the function will not modify the implicitly accessed argument, viz. the invoking object.
>
> Therefore, a trailing `const` can and only can be applied to a member function, namely, a method, NOT to a regular function.

- Just as you should use `const` references and pointers as formal function argument whenever appropriate, you should make class methods `const` whenever they do not modify the invoking object.

10.4 The `this` pointer

- There is a special C++ pointer called `this`. It is a automatically created and hided private data member of a class. When a method is called with an object, the program automatically points the pointer `this` to the invoking object.
- Strictly speaking, `this` is not a pointer; it just acts as if it is.
  - You may not declare a `this` pointer;
  - You may not manually alter the value of `this` - it acts as if it is `const`;
  - You may not get the address of `this`;
  - `this` is not counted when using `sizeof` to get the size of a class/object.
- A brief example: updating a stock's price

```
// Version #1: without 'this'
void Stock::update(double price){
  share_val = price;
  set_total();
}
// use:
myStock.update(4.99);
myStock.show();
```

```
// Version #2: with 'this'
const Stock & Stock::update(double price){ // return type must be 'const'
  share_val = price;
  set_total();
  return *this;  // return the invoking object as an lvalue
}
// use:
myStock.update(4.99).show();
```

```
// Version #3: with 'this'
Stock Stock::update(double price){
  share_val = price;
  set_total;
  return *this;  // return the invoking object as an lvalue
}
// use:
Stock tempStock = myStock.update(4.99);
tempStock.show();
```

10.5 Class scope

- Class scope applies to name defined in a class, such as names of data members and member functions. A class scope include the class declaration, and member function bodies that may be placed outside the declaration.
- Items that have class scope are known within their classes but not outside. Thus, you can use the same name in different classes or other non-class declarative regions. For instance, you may declare a data member `a` and method `show()` for class A and class B, and you may also declare a regular variable `a` and regular function `show()`.

- Data members and methods being within the class scope means that, if you want to access them from the outside, you have to specify the class with which you are accessing them. (Access control labels also have a say, of course.) There are three ways to specify the class:
  - Use the direct membership operator `.` (with an object name);
  - Use the indirect member ship operator `->` (with a name of a pointer to object);
  - Use the scope resolution operator `::` (with a class name).

  > Constructors' names are recognized when they are called; they are exempt from the class scope rule. After all, you cannot specify an object when you are calling a constructor - the object does not exist yet.

- Within a class declaration or a member function body you can use an unqualified member name.
- Class scope constants: there are two ways to declare a symbolic constant within a class scope.
  - Use the keyword `static` with `const` integral types:

```
class Bakery{
private:
  static const int Months = 12;    // ok
  static const char Mark = 'A';    // ok
  // 1. a member with an in-class initializer must be const.
  static int Months = 12;          // error
  // 2. a member of type "const double" cannot have
  //    an in-class initializer.
  static const double Rent = 2200; // error
  double costs[Months];
  ...
};
```

    - The variable `Months` is stored in the static memory with other variables, rather than in an object's memory block. Hence, there is only ONE `Months` constant, which is shared by all `Bakery` objects.
    - Chapter 12 looks further into static class members.
  - Use enumerations (only valid for integral values, such as `bool`, `char`, `int`):

```
class Bakery{
private:
  enum {Months = 12};   // a type name is unnecessary
  double costs[Months];
  ...
}
```

    - Because the class uses the enumeration merely to create a symbolic constant, with no intent of creating variables of the enumeration type, you can make the enumeration type anonymous by omitting its type name.
    - Declaring an enumeration in this fashion makes the enumeration type only visible within the class scope.
  - You CANNOT declare a constant in this fashion (surprise, huh?):

```
class Bakery{
private:
  const int Months = 12;  // error
  ...
}
```

- Why does not it work? Because declaring a class describes what an object looks like, but does not create an object. Hence, until you create an object, there is no place to store a value.

  > "Why does not the compiler store it in the static memory," you may ask. The answer is, yes, the compiler can do that, but you have to specify your intention by with `static`, i.e. `static const int Months = 12;`.

- Scoped enumerations (C++11): discussed in 4.7. The core idea is that you can use the keyword `class` to declare enumeration types that have the same enumerator names:

```
enum class computers {apple, thinkpad, dell, alienware};
enum class fruits {apple, pear, banana, grape};
fruits myFavorite = fruits::apple;
computers myLaptop = computers::apple;
```

- Unlike regular enumerations, scoped enumerations have no implicit conversions to integer types, and their underlying type is restricted to `int`, as discussed in 4.7.

□