Classes (I)

Classes are an expanded concept of *data structures*: like data structures, they can contain data members, but they can also contain functions as members.

类是数据结构的扩展概念:与数据结构一样，类可以包含数据成员，但也可以包含作为成员的函数。

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

对象是类的实例化。就变量而言，类是类型，对象是变量。

Classes are defined using either keyword `class` or keyword `struct`, with the following syntax:

类可以使用关键字class或关键字struct定义，语法如下:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

Where `class_name` is a valid identifier for the class, `object_names` is an optional list of names for objects of this class. The body of the declaration can contain *members*, which can either be data or function declarations, and optionally *access specifiers*.

class_name是类的有效标识符，object_names是该类对象的可选名称列表。声明体可以包含成员，成员可以是数据声明或函数声明，也可以访问说明符。

Classes have the same format as plain *data structures*, except that they can also include functions and have these new things called *access specifiers*. An *access specifier* is one of the following three keywords: `private`, `public` or `protected`. These specifiers modify the access rights for the members that follow them:

类具有与普通数据结构相同的格式，除了它们还可以包含函数和这些称为访问说明符的新东西之外。访问说明符是以下三个关键字之一:私有、公共或受保护。这些说明符修改其后面成员的访问权限:

- `private` members of a class are accessible only from within other members of the same class (or from their *"friends"*).

  类的私有成员只能从同一类的其他成员(或从他们的“朋友”)中访问。
- `protected` members are accessible from other members of the same class (or from their *"friends"*), but also from members of their derived classes.

  受保护的成员可以从同一类的其他成员(或它们的“朋友”)访问，也可以从它们的派生类的成员访问。
- Finally, `public` members are accessible from anywhere where the object is visible.

  最后，可以从对象可见的任何地方访问public成员。

By default, all members of a class declared with the `class` keyword have private access for all its members. Therefore, any member that is declared before any other *access specifier* has private access automatically. For example:

默认情况下，用class关键字声明的类的所有成员都对其所有成员具有私有访问权。因此，在任何其他访问说明符之前声明的任何成员自动具有私有访问权限。例如:

```
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area (void);
} rect;
```

Declares a class (i.e., a type) called `Rectangle` and an object (i.e., a variable) of this class, called `rect`. This class contains four members: two data members of type `int` (member `width` and member `height`) with *private access*(because private is the default access level) and two member functions with *public access*: the functions `set_values` and `area`, of which for now we have only included their declaration, but not their definition.

声明一个类(例如,一个类型)称为长方形和一个对象(例如,一个变量)这门课,叫做矩形。这个类包含四个成员:两个int类型的数据成员(宽度和高度)和私人访问(因为私人是默认的访问级别)和两个与公共访问成员函数:函数set_values和area，目前我们只包括它们的声明，而没有包括它们的定义。

Notice the difference between the *class name* and the *object name*: In the previous example, `Rectangle` was the *class name* (i.e., the type), whereas `rect` was an object of type `Rectangle`. It is the same relationship `int` and `a` have in the following declaration:

注意类名和对象名之间的区别:在前面的例子中，Rectangle是类名(也就是类型)，而rect是Rectangle类型的对象。在下面的声明中，int和a的关系是相同的:

```
int a;
```

where `int` is the type name (the class) and `a` is the variable name (the object).

其中int是类型名(类)，a是变量名(对象)。

After the declarations of `Rectangle` and `rect`, any of the public members of object `rect` can be accessed as if they were normal functions or normal variables, by simply inserting a dot (`.`) between *object name* and *member name*. This follows the same syntax as accessing the members of plain data structures. For example:

声明Rectangle和rect之后，对象rect的任何公共成员都可以被访问，就像访问普通函数或普通变量一样，只需要在对象名和成员名之间插入一个点(.)。这与访问普通数据结构的成员遵循相同的语法。例如:

```
rect.set_values (3,4);
myarea = rect.area();
```

The only members of `rect` that cannot be accessed from outside the class are `width` and `height`, since they have private access and they can only be referred to from within other members of that same class.

唯一不能从类外部访问的rect成员是width和height，因为它们具有私有访问权限，只能从同一类的其他成员中引用。

Here is the complete example of class Rectangle:

下面是矩形类的完整示例:

```
// classes example
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

This example reintroduces the *scope operator* (`::`, two colons), seen in earlier chapters in relation to namespaces. Here it is used in the definition of function `set_values` to define a member of a class outside the class itself.

这个例子重新介绍了作用域操作符(::，两个冒号)，在前面的章节中与命名空间有关。这里在函数set_values的定义中使用它来在类本身之外定义类的成员。

Notice that the definition of the member function `area` has been included directly within the definition of class `Rectangle` given its extreme simplicity. Conversely, `set_values` it is merely declared with its prototype within the class, but its definition is outside it. In this outside definition, the operator of scope (`::`) is used to specify that the function being defined is a member of the class `Rectangle` and not a regular non-member function.

注意，成员函数区域的定义已经直接包含在矩形类的定义中，因为它非常简单。相反，set_values它只是在类内部用它的原型声明，但它的定义在类外部。在这个外部定义中，scope(::)操作符用于指定所定义的函数是Rectangle类的成员，而不是普通的非成员函数。

The scope operator (`::`) specifies the class to which the member being defined belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, the function `set_values` in the previous example has access to the variables `width` and `height`, which are private members of class `Rectangle`, and thus only accessible from other members of the class, such as this.

作用域操作符(::)指定被定义成员所属的类，授予完全相同的作用域属性，就像这个函数定义直接包含在类定义中一样。例如，上一个示例中的set_values函数可以访问变量width和height，它们是Rectangle类的私有成员，因此只能从该类的其他成员访问，比如this。

The only difference between defining a member function completely within the class definition or to just include its declaration in the function and define it later outside the class, is that in the first case the function is automatically considered an *inline* member function by the compiler, while in the second it is a normal (not-inline) class member function. This causes no differences in behavior, but only on possible compiler optimizations.

完全在类定义内定义成员函数，还是只在函数中包含成员函数的声明，然后在类外定义，两者之间的唯一区别是，在第一种情况下，编译器会自动将该函数视为内联成员函数，而在第二个函数中，它是一个普通的(非内联的)类成员函数。这不会导致行为上的差异，但只会导致可能的编译器优化。

Members `width` and `height` have private access (remember that if nothing else is specified, all members of a class defined with keyword `class` have private access). By declaring them private, access from outside the class is not allowed. This makes sense, since we have already defined a member function to set values for those members within the object: the member function `set_values`. Therefore, the rest of the program does not need to have direct access to them. Perhaps in a so simple example as this, it is difficult to see how restricting access to these variables may be useful, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

成员宽度和高度具有私有访问权限(请记住，如果没有指定其他内容，则用关键字class定义的类的所有成员都具有私有访问权限)。通过声明它们为私有，就不允许从类外部访问。这是有意义的，因为我们已经定义了一个成员函数来为对象中的这些成员设置值:成员函数set_values。因此，程序的其余部分不需要直接访问它们。也许在如此简单的例子中，很难看出限制对这些变量的访问是有用的，但是在更大的项目中，不能以意外的方式(从对象的角度来看是意外的)修改值可能是非常重要的。

The most important property of a class is that it is a type, and as such, we can declare multiple objects of it. For example, following with the previous example of class `Rectangle`, we could have declared the object `rectb` in addition to object `rect`:

类最重要的属性是它是一个类型，因此，我们可以声明它的多个对象。例如，在前面的矩形类的例子中，我们可以在对象rect之外声明对象rectb:

```
// example: one class, two objects
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area () {return width*height;}
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

In this particular case, the class (type of the objects) is `Rectangle`, of which there are two instances (i.e., objects): `rect` and `rectb`. Each one of them has its own member variables and member functions.

在本例中，类(对象的类型)是Rectangle，其中有两个实例(即对象):rect和rectb。每一个都有自己的成员变量和成员函数。

Notice that the call to `rect.area()` does not give the same result as the call to `rectb.area()`. This is because each object of class `Rectangle` has its own variables `width` and `height`, as they -in some way- have also their own function members `set_value` and `area` that operate on the object's own member variables.

注意，调用rect.area()不会得到与调用rectb.area()相同的结果。这是因为矩形类的每个对象都有自己的变量width和height，在某种程度上，它们也有自己的函数成员set_value和area，这些函数成员对对象自己的成员变量进行操作。

Classes allow programming using object-oriented paradigms: Data and functions are both members of the object, reducing the need to pass and carry handlers or other state variables as arguments to functions, because they are part of the object whose member is called. Notice that no arguments were passed on the calls to `rect.area` or `rectb.area`. Those member functions directly used the data members of their respective objects `rect` and `rectb`.

类允许使用面向对象的范例进行编程:数据和函数都是对象的成员，减少了将处理程序或其他状态变量作为参数传递给函数的需要，因为它们是其成员被调用的对象的一部分。注意，调用rect.area或rectb.area时没有传递参数。这些成员函数直接使用它们各自对象rect和rectb的数据成员。

Constructors

What would happen in the previous example if we called the member function `area` before having called `set_values`? An undetermined result, since the members `width` and `height` had never been assigned a value.

在前面的例子中如果我们在调用set_values之前调用成员函数area会发生什么?一个未确定的结果，因为成员的宽度和高度从未被赋值。

In order to avoid that, a class can include a special function called its *constructor*, which is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage.

为了避免这种情况，类可以包含一个称为其构造函数的特殊函数，每当创建该类的新对象时，都会自动调用该构造函数，从而允许类初始化成员变量或分配存储空间。

This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type; not even `void`.

该构造函数的声明类似于普通成员函数，但具有与类名匹配的名称，且没有任何返回类型;甚至空白。

The `Rectangle` class above can easily be improved by implementing a constructor:

上面的Rectangle类可以很容易地通过实现一个构造函数来改进:

```
// example: class constructor
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle (int,int);
    int area () {return (width*height);}
};

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

The results of this example are identical to those of the previous example. But now, class `Rectangle` has no member function `set_values`, and has instead a constructor that performs a similar action: it initializes the values of `width` and `height` with the arguments passed to it.

本示例的结果与前一个示例的结果相同。但是现在，类Rectangle没有成员函数set_values，而是有一个执行类似操作的构造函数:它用传递给它的参数初始化width和height的值。

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

注意在创建类的对象时，这些参数是如何传递给构造函数的:

```
Rectangle rect (3,4);
Rectangle rectb (5,6);
```

Constructors cannot be called explicitly as if they were regular member functions. They are only executed once, when a new object of that class is created.

不能像调用普通成员函数那样显式调用构造函数。它们只在该类的新对象创建时执行一次。

Notice how neither the constructor prototype declaration (within the class) nor the latter constructor definition, have return values; not even `void`: Constructors never return values, they simply initialize the object.

注意构造函数原型声明(在类中)和后一个构造函数定义都没有返回值;甚至不是void:构造函数从不返回值，它们只是初始化对象。

Overloading constructors

Like any other function, a constructor can also be overloaded with different versions taking different parameters: with a different number of parameters and/or parameters of different types. The compiler will automatically call the one whose parameters match the arguments:

与任何其他函数一样，构造函数也可以重载带有不同形参的不同版本:具有不同数量的形参和/或不同类型的形参。编译器会自动调用形参与实参匹配的函数:

```
// overloading class constructors
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle ();
    Rectangle (int,int);
    int area (void) {return (width*height);}
};

Rectangle::Rectangle () {
  width = 5;
  height = 5;
}

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

In the above example, two objects of class `Rectangle` are constructed: `rect` and `rectb`. `rect` is constructed with two arguments, like in the example before.

在上面的例子中，我们构造了类Rectangle的两个对象:rect和rectb。像前面的例子一样，Rect由两个参数构成。

But this example also introduces a special kind constructor: the *default constructor*. The *default constructor* is the constructor that takes no parameters, and it is special because it is called when an object is declared but is not initialized with any arguments. In the example above, the *default constructor* is called for `rectb`. Note how `rectb` is not even constructed with an empty set of parentheses - in fact, empty parentheses cannot be used to call the default constructor:

但是这个例子还引入了一个特殊类型的构造函数:默认构造函数。默认构造函数是不接受形参的构造函数，它的特殊之处在于它在声明对象但没有初始化任何实参时被调用。在上面的例子中，调用了rectb的默认构造函数。注意，rectb甚至不是用一组空括号构造的——事实上，空括号不能用来调用默认构造函数:

```
Rectangle rectb;   // ok, default constructor called
Rectangle rectc(); // oops, default constructor NOT called
```

This is because the empty set of parentheses would make of `rectc` a function declaration instead of an object declaration: It would be a function that takes no arguments and returns a value of type `Rectangle`.

这是因为空的圆括号集将使rectc成为一个函数声明而不是对象声明:它将是一个不带参数并返回类型为Rectangle的值的函数。

Uniform initialization

The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as *functional form*. But constructors can also be called with other syntaxes:

如上所示，通过将构造函数的参数括在圆括号中调用构造函数的方法称为函数形式。但是构造函数也可以用其他语法调用:

First, constructors with a single parameter can be called using the variable initialization syntax (an equal sign followed by the argument):

首先，带有单个形参的构造函数可以使用变量初始化语法(参数后面跟着等号)来调用:

```
class_name object_name = initialization_value;
```

More recently, C++ introduced the possibility of constructors to be called using *uniform initialization*, which essentially is the same as the functional form, but using braces (`{}`) instead of parentheses (`()`):

最近，c++引入了使用统一初始化来调用构造函数的可能性，这本质上与函数形式相同，但使用大括号({})而不是圆括号(():

```
class_name object_name { value, value, value, ... }
```

Optionally, this last syntax can include an equal sign before the braces.

最后一种语法可以在大括号前包含等号。

Here is an example with four ways to construct objects of a class whose constructor takes a single parameter:

下面是一个例子，有四种方法来构造类的对象，该类的构造函数接受一个参数:

```
// classes and uniform initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) { radius = r; }
    double circum() {return 2*radius*3.14159265;}
};

int main () {
  Circle foo (10.0);   // functional form
  Circle bar = 20.0;   // assignment init.
  Circle baz {30.0};   // uniform init.
  Circle qux = {40.0}; // POD-like

  cout << "foo's circumference: " << foo.circum() << '\n';
  return 0;
}
```

An advantage of uniform initialization over functional form is that, unlike parentheses, braces cannot be confused with function declarations, and thus can be used to explicitly call default constructors:

与函数形式相比，统一初始化的一个优点是，与圆括号不同，大括号不能与函数声明混淆，因此可以用于显式调用默认构造函数:

```
Rectangle rectb;   // default constructor called
Rectangle rectc(); // function declaration (default constructor NOT called)
Rectangle rectd{}; // default constructor called
```

The choice of syntax to call constructors is largely a matter of style. Most existing code currently uses functional form, and some newer style guides suggest to choose uniform initialization over the others, even though it also has its potential pitfalls for its preference of `initializer_list` as its type.

调用构造函数的语法选择在很大程度上取决于风格。大多数现有代码目前使用函数形式，一些较新的样式指南建议选择统一初始化而不是其他初始化，尽管它也有其首选initializer_list作为类型的潜在缺陷。

Member initialization in constructors

When a constructor is used to initialize other members, these other members can be initialized directly, without resorting to statements in its body. This is done by inserting, before the constructor's body, a colon (`:`) and a list of initializations for class members. For example, consider a class with the following declaration:

当使用构造函数初始化其他成员时，可以直接初始化这些其他成员，而不需要求助于构造函数体中的语句。这是通过在构造函数的主体之前插入冒号(':')和类成员的初始化列表来实现的。例如，考虑一个具有以下声明的类:

```
class Rectangle {
    int width,height;
  public:
    Rectangle(int,int);
    int area() {return width*height;}
};
```

The constructor for this class could be defined, as usual, as:

这个类的构造函数可以像往常一样定义为:

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

But it could also be defined using *member initialization* as:

但它也可以使用成员初始化来定义:

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

Or even:

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

Note how in this last case, the constructor does nothing else than initialize its members, hence it has an empty function body.

请注意，在最后一种情况下，构造函数除了初始化其成员之外不做其他事情，因此它有一个空函数体。

For members of fundamental types, it makes no difference which of the ways above the constructor is defined, because they are not initialized by default, but for member objects (those whose type is a class), if they are not initialized after the colon, they are default-constructed.

对于基本类型的成员，以上构造函数的定义方式没有区别，因为它们在默认情况下没有初始化，但对于成员对象(其类型是类的对象)，如果它们没有在冒号之后初始化，它们就是默认构造的。

Default-constructing all members of a class may or may always not be convenient: in some cases, this is a waste (when the member is then reinitialized otherwise in the constructor), but in some other cases, default-construction is not even possible (when the class does not have a default constructor). In these cases, members shall be initialized in the member initialization list. For example:

Default-constructing的所有成员可能总是不方便:在某些情况下,这是一种浪费(当会员然后否则在构造函数中初始化),但在其他一些情况下,default-construction甚至是不可能的(当类没有默认构造函数)。在这些情况下，成员必须在成员初始化列表中初始化。例如:

```
// member initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) : radius(r) { }
    double area() {return radius*radius*3.14159265;}
};

class Cylinder {
    Circle base;
    double height;
  public:
    Cylinder(double r, double h) : base (r), height(h) {}
    double volume() {return base.area() * height;}
};

int main () {
  Cylinder foo (10,20);

  cout << "foo's volume: " << foo.volume() << '\n';
  return 0;
}
```

In this example, class `Cylinder` has a member object whose type is another class (`base`'s type is `Circle`). Because objects of class `Circle` can only be constructed with a parameter, `Cylinder`'s constructor needs to call `base`'s constructor, and the only way to do this is in the *member initializer list*.

在本例中，圆柱体类有一个成员对象，其类型是另一个类(基类的类型是Circle)。因为Circle类的对象只能用形参构造，所以Cylinder的构造函数需要调用base的构造函数，而做到这一点的唯一方法是在成员初始化列表中。

These initializations can also use uniform initializer syntax, using braces `{}` instead of parentheses `()`:

这些初始化也可以使用统一的初始化语法，使用大括号{}而不是圆括号():

```
Cylinder::Cylinder (double r, double h) : base{r}, height{h} { }
```

Pointers to classes

Objects can also be pointed to by pointers: Once declared, a class becomes a valid type, so it can be used as the type pointed to by a pointer. For example:

对象也可以由指针指向:一旦声明，类就变成了有效类型，因此它可以用作指针所指向的类型。例如:

```
Rectangle * prect;
```

is a pointer to an object of class `Rectangle`.

是指向类Rectangle对象的指针。

Similarly as with plain data structures, the members of an object can be accessed directly from a pointer by using the arrow operator (`->`). Here is an example with some possible combinations:

与普通数据结构类似，可以通过使用箭头操作符(->)从指针直接访问对象的成员。下面是一些可能组合的例子:

```
// pointer to classes example
#include <iostream>
using namespace std;

class Rectangle {
  int width, height;
public:
  Rectangle(int x, int y) : width(x), height(y) {}
  int area(void) { return width * height; }
};


int main() {
  Rectangle obj (3, 4);
  Rectangle * foo, * bar, * baz;
  foo = &obj;
  bar = new Rectangle (5, 6);
  baz = new Rectangle[2] { {2,5}, {3,6} };
  cout << "obj's area: " << obj.area() << '\n';
  cout << "*foo's area: " << foo->area() << '\n';
  cout << "*bar's area: " << bar->area() << '\n';
  cout << "baz[0]'s area:" << baz[0].area() << '\n';
  cout << "baz[1]'s area:" << baz[1].area() << '\n';
  delete bar;
  delete[] baz;
  return 0;
```

```
}
```

This example makes use of several operators to operate on objects and pointers (operators `*`, `&`, `.`, `->`, `[]`). They can be interpreted as:

这个例子使用了几个操作符来操作对象和指针(操作符' * '，' & '，'。`,`->`,`[]`).它们可以解释为:

expression	can be read as
`*x`	pointed to by `x`
`&x`	address of `x`
`x.y`	member `y` of object `x`
`x->y`	member `y` of object pointed to by `x`
`(*x).y`	member `y` of object pointed to by `x` (equivalent to the previous one)
`x[0]`	first object pointed to by `x`
`x[1]`	second object pointed to by `x`
`x[n]`	(`n+1`)th object pointed to by `x`

Most of these expressions have been introduced in earlier chapters. Most notably, the chapter about arrays introduced the offset operator (`[]`) and the chapter about plain data structures introduced the arrow operator (`->`).

这些表达式中的大多数已经在前面的章节中介绍过了。最值得注意的是，关于数组的一章介绍了偏移操作符([])，关于普通数据结构的一章介绍了箭头操作符(->)。

Classes defined with struct and union

Classes can be defined not only with keyword `class`, but also with keywords `struct` and `union`.

类不仅可以用关键字class定义，还可以用关键字struct和union定义。

The keyword `struct`, generally used to declare plain data structures, can also be used to declare classes that have member functions, with the same syntax as with keyword `class`. The only difference between both is that members of classes declared with the keyword `struct` have `public` access by default, while members of classes declared with the keyword `class` have `private` access by default. For all other purposes both keywords are equivalent in this context.

关键字struct通常用于声明普通数据结构，也可用于声明具有成员函数的类，其语法与关键字class相同。两者之间唯一的区别是，用关键字struct声明的类的成员在默认情况下具有公共访问权限，而用关键字class声明的类的成员在默认情况下具有私有访问权限。对于所有其他目的，这两个关键字在此上下文中是等价的。

Conversely, the concept of *unions* is different from that of classes declared with `struct` and `class`, since unions only store one data member at a time, but nevertheless they are also classes and can thus also hold member functions. The default access in union classes is `public`.

相反，联合的概念与用struct和class声明的类的概念不同，因为联合一次只存储一个数据成员，但它们也是类，因此也可以保存成员函数。联合类中的默认访问是公共的。