

# 2-Data Abstraction

## 1. The `const` Keyword

🏆 The general rules for converting between `const` and non-`const` is that the conversion must not enable modifications that are prohibited without the conversion.

```
1  const int x = 3;
2  int &ref = x; // ERROR-enables const object to be modified through ref
3  ref = 4;
```

```
1  const int x = 3;
2  int *ptr = &x; // ERROR-enables const object to be modified through ptr
3  *ptr = 4;
```

```
1  int x = 3;
2  int *ptr1 = &x; // OK -- does not permit const object to be modified
3  const int *ptr2 = ptr1; // OK -- does not permit const object to be modified
4  ptr1 = ptr2; // ERROR -- compiler sees that ptr2 is pointing to a

5                      // a const object, but ptr1 is not a pointer

6                      // to const
```

## 2. Abstract Data Types

对于一个数据类型，它是如何表示的以及它能执行的操作是如何实现的对它的使用者来说并不重要，使用者只需要了解它是什么以及它能执行什么操作；同一个数据类型的表示方式可能有多种选择，它能执行的操作也可能有多种实现方式，对这些具体的实现方式进行抽象，我们就能得到一个抽象数据类型。

🏆 An abstract data type (ADT) is a conceptual model that defines a set of operations and behaviors for a data type, without specifying how these operations are implemented and how data is represented.

由上面的内容可知，抽象体现在两方面：

- **数据抽象**：没有指定数据在底层该如何表示。
- **过程抽象**：只定义操作的接口，没有指定操作该如何实现。

## 2.1 Abstract Data Types in C++

在C++里面定义或实现ADTs，我们可以使用类。

🏆 A C++ class includes both *member variables*, which define the data representation, as well as *member methods/functions* that operate on the data.

下面是一个例子：

```
1  class Triangle {
2      double a;
3      double b;
4      double c;
5
6  public:
7      Triangle(double a_in, double b_in, double c_in);
8
9      double perimeter() const {
10         return this->a + this->b + this->c;
11     }
12
13     void scale(double s) {
14         this->a *= s;
15         this->b *= s;
16         this->c *= s;
17     }
18
19 };
```

这里，我们实现了一个抽象数据类型——三角形类型：成员变量 `a, b, c` 表示三条边的长度，它们是三角形的底层数据表示；成员方法 `perimeter, scale` 表示三角形类型所能执行的操作，并给出了具体的实现。

下面是实例化三角形类型的例子：

```
1  int main() {
2      Triangle t1(2, 4, 5);
3      t1.scale(2);
4      cout << t1.perimeter() << endl;
5  }
```

访问实例化的对象中的成员，我们可以使用 `.` 运算符： `<object>.<member>`。

对于 `scale` 方法，想要修改 `t1`，肯定需要知道其地址，但我们并没有显式地将 `t1` 的地址传给 `scale`，那该如何修改 `t1` 呢？——C++会为每个成员方法添加一个隐式的 `this` 指针指向当前调用该方法的 `Triangle` 对象，比如这里的 `t1`，这样，就可以在 `scale` 方法中修改 `t1` 对象了。

但有时我们并不希望某个方法修改调用该方法的对象，也就是在 `this` 指针“眼中”，它需要指向一个 `const` 对象。比如这里的 `perimeter` 方法，它并不需要修改 `t1` 对象，但 `this` 指针是被隐式添加的，我们无法在函数的形参列表中为其添加 `const`。C++是这样解决这个问题的：将 `const` 关键字放在函数的形参列表后，实现体之前，就像这里的 `perimeter` 方法，它相当于

```
1  double perimeter(const Triangle *this) {
2      return this->a + this->b + this->c;
3  }
```

### 2.1.1 Implicit `this->`

🏆 Since member variables and member functions are both located within the scope of a class, C++ allows us to refer to members from within a member function without the explicit `this->` syntax. The compiler automatically inserts the member dereference for us.

比如，

```
1  class Triangle {
2      double a;
3      double b;
4      double c;
5      ...
6      double perimeter() const {
7          return a + b + c; // Equivalent to: this->a + this->b + this->c
8      }
```

```
9     };
```

在这种情形下，编译器能判断出我们正在访问类的成员，因此能够自动添加 `this->`。然而，如果在更小的作用域中存在于成员相同的名称，我们必须自己显式地添加 `this->`，否则更小的作用域中的名称会将类的成员名称给覆盖掉。比如，

```
1  class Triangle {
2      double a;
3      ...
4      double set_side1(double a) {
5          this->a = a;
6      }
7  };
```

## 2.1.2 Member Accessibility

对于一个类的成员变量，使用者应该避免直接访问它们，基于此，C++为我们提供了成员访问修饰符。

🏆 Declaring members as *private* prevents access from outside of the class, while declaring them as *public* allows outside access. We give a set of members a particular access level by placing `private:` or `public:` before the members – that access level applies to subsequent members until a new access specifier is encountered. With the `class` keyword, the default access level is private.

## 2.1.3 Constructors

🏆 A *constructor* is similar to a member function. Its purpose is to initialize a class-type object. In most cases, C++ guarantees that a constructor is called when creating an object of class type.

比如，

```
1  Triangle t2(3, 4, 5);           // calls three-argument constructor
2  Triangle t3 = Triangle(3, 4, 5); // calls three-argument constructor
3  // examples with "uniform initialization syntax":
4  Triangle t4{3, 4, 5};           // calls three-argument constructor
5  Triangle t5 = {3, 4, 5};        // calls three-argument constructor
6  Triangle t6 = Triangle{3, 4, 5}; // calls three-argument constructor
```

构造函数和成员函数类似，但是它

- 没有返回类型
- 构造函数的名字和类的名字一样

同样，构造函数也有一个隐式的 `this` 指针。

在了解了什么是构造函数过后，我们可以实现之前的 `Triangle(double a_in, double b_in, double c_in);` 了：

```
1  // poor implementation
2  Triangle(double a_in, double b_in, double c_in) {
3      a = a_in;
4      b = b_in;
5      c = c_in;
6  }
```

然而，上面的实现有一个问题：函数体的语句是在对成员变量 `a, b, c` 进行赋值，而不是初始化。在实例化对象时，这三个成员变量会先被默认初始化，然后在被赋上新值。在本例中，这只是个小问题，但在其他情况下可能会更严重，特别是有几种变量允许初始化但不允许赋值：

- arrays
- references
- `const` variables
- class-type variables that disable assignment (e.g. streams)

另外，如果一种类型进行赋值会做很多操作，上面的实现将会影响一些效率。



C++ provides two mechanisms for initializing a member variable:

- directly in the declaration of the variable, similar to initializing a non-member variable
- through a member-initializer list

A *member-initializer list* is syntax specific to a constructor. It is a list of initializations that appear between a colon symbol and the constructor body. An individual initialization consists of a member-variable name, followed by an initialization expression enclosed by parentheses (or curly braces).

比如，

```
1  // good implementation
```

```
2 Triangle(double a_in, double b_in, double c_in)
3     :a(a_in), b(b_in), c(c_in) {}
```

上面的构造函数将成员 `a` 初始化为 `a_in` 的值，将 `b` 初始化为 `b_in` 的值，将 `c` 初始化为 `c_in` 的值。构造函数的函数体是空的，因为它没有更多的工作要做。

🏆 If a member is initialized in both its declaration and a member-initializer list, the latter takes precedence, so that the initialization in the member declaration is ignored.

另外，当一个类类型对象的生命周期结束时，编译器会自动调用它的析构函数，更多细节在后面介绍。

## 2.1.4 Default Initialization and Default Constructors

🏆 Every object in C++ is initialized upon creation, whether the object is of class type or not. If no explicit initialization is provided, it undergoes *default initialization*. Default initialization does the following:

- Objects of atomic type (e.g. `int`, `double`, pointers) are default initialized by doing nothing. This means they retain whatever value was already there in memory. Put another way, atomic objects have undefined values when they are default initialized.
- An array is default initialized by in turn default initializing its elements. Thus, an array of atomic objects is default initialized by doing nothing, resulting in undefined element values.
- A class-type object is default initialized by calling the *default constructor*, which is the constructor that takes no arguments. If no such constructor exists, or if it is inaccessible (e.g. it is private), a compile-time error results.
- An array of class-type objects is default initialized by calling the default constructor on each element. Thus, the element type must have an accessible default constructor in order to create an array of that type.

Within a class, if a member variable is neither initialized at declaration nor in the member-initializer list of a constructor, it is default initialized.

The default constructor is so named because it is invoked in default initialization, and it takes no arguments.

我们可以为 `Triangle` 类定义一个默认构造函数：

```
1  class Triangle {
2      double a;
3      double b;
4      double c;
5  public:
6      // default constructor
7      Triangle()
8          : a(1), b(1), c(1) {}
9      // non-default constructor
10     Triangle(double a_in, double b_in, double c_in)
11         : a(a_in), b(b_in), c(c_in) {}
12 };
```

一个类有多个构造函数，这是函数重载的一种形式，我们将在后面介绍它。当创建一个对象时，编译器会根据参数来决定调用哪一个构造函数。



If a class declares no constructors at all, the compiler provides an *implicit default constructor*. The behavior of this constructor is as if it were empty, so that it default initializes each member variable.

比如，

```
1  class Person {
2      std::string name;
3      int age;
4      bool is_ninja;
5      // implicit default constructor
6      // Person() {} // default initializes each member variable
7  };
8
9  int main() {
10     Person elise;           // calls implicit default constructor
11     cout << elise.name;     // prints nothing: default ctor for string makes it
                             // empty
12     cout << elise.age;      // prints undefined value
13     cout << elise.is_ninja; // prints undefined value
14 };
```

如果一个类声明了任何构造函数，编译器则不会提供隐式的默认构造函数。比如，

```

1  class Triangle {
2      double a;
3      double b;
4      double c;
5
6  public:
7      Triangle(double a_in, double b_in, double c_in);
8
9      double perimeter() const {
10         return this->a + this->b + this->c;
11     }
12 };
13 int main() {
14     Triangle t1; // ERROR: no implicit or explicit default constructor
15     return 0;
16 }

```

## 2.1.5 Get and Set Functions

🏆 With C++ classes, member variables are usually declared private, since they are implementation details. However, many C++ ADTs provide a means of accessing the abstract data through *get* and *set functions* (also called *getters* and *setters* or *accessor functions*). These are provided as part of the interface as an abstraction over the underlying data.

## 2.1.6 Information Hiding

当用户需要查看一个类的接口时，接口的实现对用户来算是冗余信息，所以我们可以将类的声明和类的实现分离开来：将类的声明放在头文件中，将类的实现放在源文件中。

比如，

```

1  // Triangle.h
2  // A class that represents a triangle ADT
3  class Triangle {
4  public:
5      // EFFECTS: Initializes this to a 1x1x1 triangle.
6      Triangle();
7
8      // EFFECTS: Initializes this with the given side lengths.
9      Triangle(double a, double b, double c);
10

```




```

11      // EFFECTS: Returns the perimeter of this triangle.
12      double perimeter() const;
13
14      // REQUIRES: s > 0
15      // MODIFIES: *this
16      // EFFECTS: Scales the sides of this triangle by the factor
17      void scale(int factor);
18
19  private:
20      double a;
21      double b;
22      double c;
23      // INVARIANTS:
24      // positive side lengths: a > 0 && b > 0 && c > 0
25      // triangle inequality: a + b > c && a + c > b && b + c > a
26  };

```

另外，最好在类中先声明公共成员，然后再声明私有成员，这样用户就不必跳过实现细节去寻找公共接口，而是首先看到公共接口。

 We then define the constructors and member functions outside of the class definition, in the corresponding source file. In order to define a member function outside of a class, we need two things:

1. A declaration of the function within the class, so that the compiler (and other programmers) can tell that the member exists.
2. Syntax in the definition that tells the compiler that the function is a member of the associated class and not a top-level function.

The latter is accomplished by prefixing the member name with the class name, followed by the *scope-resolution operator*.

对于上面的例子，源文件的实现是这样的：

```

1  #include "Triangle.h"
2  Triangle::Triangle()
3      :a(1), b(1), c(1) {}
4
5  Triangle::Triangle(double a, double b, double c)
6      :a(a), b(b), c(c) {}
7
8  double Triangle::perimeter() const {
9      return this->a + this->b + this->c;


```

```


10  }
11
12  void Triangle::scale(int factor) {
13      this->a *= factor;
14      this->b *= factor;
15      this->c *= factor;
16  }

```

## 2.1.7 Member-Initialization Order

 Member variables are always initialized in the order in which they are declared in the class. This is the case regardless if some members are initialized at the declaration point and others are not, or if a constructor's member-initializer list is out of order.

## 2.1.8 Delegating Constructors

 When a class has multiple constructors, it can be useful to invoke one constructor from another. This allows us to avoid code duplication, and it also makes our code more maintainable by reducing the number of places where we hardcode implementation details.


In order to delegate to another constructor, we must do so in the member-initializer list. The member-initializer list must consist solely of the call to the other constructor.

比如，

```

1  // EFFECTS: Initializes this to be an equilateral triangle with the give side
    length.
2  Triangle(double side);
3
4  Triangle::Triangle(double side)
5      :Triangle(side, side, side) {}

```

 The delegation must be in the member-initializer list. If we invoke a different constructor from within the body, it does not do delegation; rather, it creates a new, temporary object and then throws it away.

比如，

```

1  Triangle(double side_in) { // default initializes members
2      Triangle(side_in, side_in, side_in); // creates a new Triangle object
    that                                     // lives in the
    activation record for                     // this
    constructor
3  }

```

### 3. Derived Classes and Inheritance

C++中的继承语法如下：

```

1  class <derived class> : public <base class> {...};

```



The syntax for deriving from a base class is to put on a colon after the name of the derived class, then the `public` keyword, then the name of the base class.

在公共继承中，派生类被认为是一种基类，基类的接口自然是派生类的接口。还有 `private`, `protected` 继承，这里不做介绍，默认是私有继承。



In order to properly initialize a derived-class object, its constructor must ensure that its base-class subobject is also appropriately initialized. The base class may have private member variables, which cannot be accessed from the derived class, so the derived class does not initialize the inherited members directly. Instead, it invokes a base-class constructor in the member-initializer list of its own constructors.

比如，

```

1  class Bird {
2      public:
3          Bird(const string &name)
4              :age(0), name(name) {}
5
6          string get_name() const {
7              return name;
8          }
9
10         int get_age() const {
11             return age;

```

```

12     }
13
14     void have_birthday() {
15         age++;
16     }
17
18     void talk() const {
19         std::cout << "tweet" << std::endl;
20     }
21 private:
22     int age;
23     string name;
24 };
25 class Chicken: public Bird {
26 public:
27     Chicken(const string &name)
28         :Bird(name), roads_crossed(0) {}
29
30     void cross_road() {
31         roads_crossed++;
32     }
33     void talk() const {
34         std::cout << "bawwk" << std::endl;
35     }
36 private:
37     int roads_crossed;
38 };

```

🏆 In C++, a derived-class constructor **always** invokes a constructor for the base class. If an explicit invocation does not appear in the member-initializer list, there is an implicit call to the default constructor. If the base class has no default constructor, an error results.

## 3.1 Ordering of Constructors and Destructors

🏆 When there are multiple objects that are constructed and destructed, C++ follows a “socks-and-shoes” ordering: when we put on socks and shoes in the morning, we put on socks first, then our shoes. In the evening, however, when we take them off, we do so in the reverse order: first our shoes, then our socks. In the case of a derived class, C++ will always construct the base-class subobject before initializing the derived-class pieces. Destruction is in the reverse order: first the derived-class destructor runs, then the base-class one.

## 3.2 Name Looking up and Hiding

🏆 When a member access is applied to an object, the compiler follows a specific process to look up the given name:

- The compiler starts by looking for a member with that name in the compile-time or *static type* of the object. (We will discuss static and dynamic types next time.)
- If no member with that name is found, the compiler repeats the process on the base class of the given type. If the type has no base class, a compiler error is generated.
- If a member with the given name is found, the compiler then checks whether or not the member is accessible and whether the member can be used in the given context. If not, a compiler error is generated – the lookup process does not proceed further.

🏆 On occasion, we wish to access a hidden member rather than the member that hides it. The most common case is when a derived-class version of a function calls the base-class version as part of its functionality.

比如，在 `Chicken` 类中，

```
1 void talk() const {
2     if (age >= 1) { // ERROR: age is private in Bird
3         std::cout << "bawwk" << std::endl;
4     } else {
5         // baby chicks make more of a tweeting rather than clucking noise
6         Bird::talk(); // call Bird's version of talk()
7     }
8 }
```

🏆 By using the scope-resolution operator, we are specifically asking for the `Bird` version of `talk()`, enabling access to it even though it is hidden.

上面的代码有一个小问题：`Chicken` 是无法访问私有的 `age` 成员的，有两种方法解决这个问题：

- 在基类中将 `age` 声明为 `protected`，这允许基类的派生类能够访问它，但外界无法访问它。
- 基类提供访问 `age` 的 `public` 或 `protected` 接口

第一种方式是不可取的，因为 `age` 属于基类的实现细节，理应封装隐藏起来，所以我们考虑提供 `public` 或者 `protected` 接口。这里，`Bird` 已经提供了 `public` 接口，可以直接使用：

```
1 void talk() const {
2     if (get_age() >= 1) {
3         std::cout << "bawwk" << std::endl;
4     } else {
5         // baby chicks make more of a tweeting rather than clucking noise
6         Bird::talk(); // call Bird's version of talk()
7     }
8 }
```

## 4. Polymorphism



The word *polymorphism* literally means “many forms.” In the context of programming, there are several forms of polymorphism:

- *ad hoc polymorphism*, which refers to function overloading
- *parametric polymorphism* in the form of templates
- *subtype polymorphism*, which allows a derived-class object to be used where a base-class object is expected

The unqualified term “polymorphism” usually refers to subtype polymorphism. We proceed to discuss ad hoc and subtype polymorphism, deferring parametric polymorphism until later.

### 4.1 Function Overloading



Ad hoc polymorphism refers to *function overloading*, which is the ability to use a single name to refer to many different functions **in a single scope**. C++ allows both top-level functions and member functions to be overloaded.

When we invoke an overloaded function, the compiler resolves the function call by comparing the types of the arguments to the parameters of the candidate functions and finding the best match.

Function overloading requires the signatures of the functions to differ, so that overload resolution can choose the overload with the most appropriate signature. Here, “signature” refers to the function name and parameter types – the return type is not part of the signature and is not considered in overload resolution.

## 4.2 Subtype Polymorphism

在大多数面向对象编程语言中，继承是实现多态的手段，C++也不例外。下面逐点介绍。



1. C++ allows a reference or pointer of a base type to refer to an object of a derived type. It allows implicit *upcasts*, which are conversions that go upward in the inheritance hierarchy.

比如，

```
1 Bird &bird_ref = chicken;
2 Bird *bird_ptr = &chicken;
```

不过，`implicit downcasts` 是不允许的，比如，

```
1 Chicken &chicken_ref = bird_ref; // ERROR: implicit downcast
2 Chicken *chicken_ptr = bird_ptr; // ERROR: implicit downcast
```

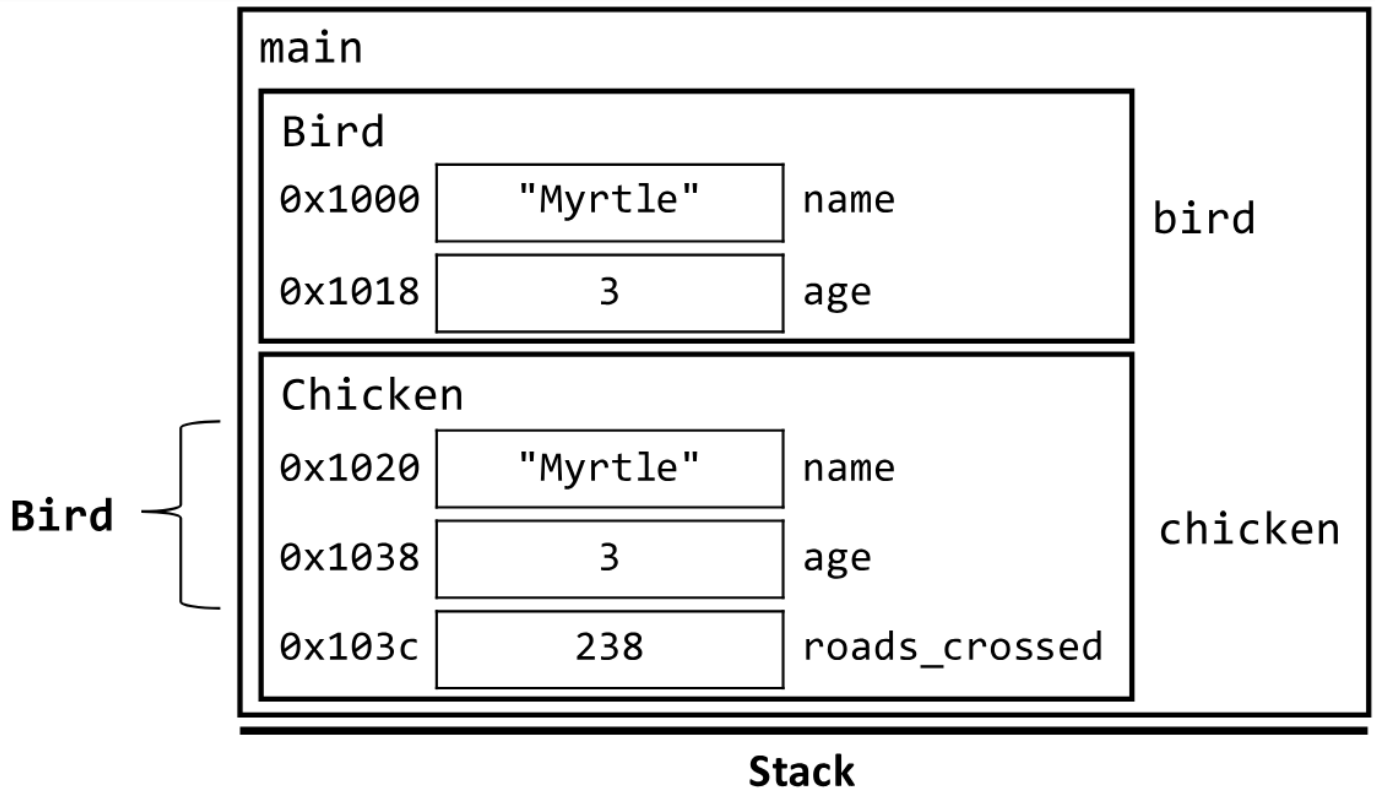
这可能不安全，因为子类可能扩展了父类的内容，子类对象所占的内存更大，而父类对象所占的内存更小，把父类对象当做子类对象使用时可能超出了内存范围了，所以这是不被允许的。我们可以使用 `static_cast` 强制转换，如

```
1 Chicken &chicken_ref = static_cast<Chicken &>(bird_ref);
2 Chicken *chicken_ptr = static_cast<Chicken *>(bird_ptr);
```

不过要确保 `bird_ref` 所指的是对象是 `Chicken` 型的，否则还是不完全的行为。

需要注意的是，把一个子类对象直接赋值给父类类型的变量是可以的，但此时会发生 `object slicing`：定义在父类中的成员会被拷贝，而定义在子类中的成员会被丢弃。比如

```
1 int main() {
2     Chicken chicken("Myrtle");
3     // ...
4     Bird bird = chicken;
5 }
```



所以为了避免这种情况，我们要使用指针或引用。

🏆 In order to be able to bind a base-class reference or pointer to a derived-class object, the inheritance relationship must be accessible. From outside the classes, this means that the derived class must publicly inherit from the derived class. Otherwise, the outside world is not allowed to take advantage of the inheritance relationship.

比如，

```
1 class A {
2 };
3 class B : A {    // default is private when using the class keyword
4 };
5 int main() {
6     B b;
7     A *a_ptr = &b; // ERROR: inheritance relationship is private
8 }
```





2. For a particular member function, C++ gives us the option of either static binding where the compiler determines which function to call based on the static type (declared type) of the object, or dynamic binding, where the program also takes the dynamic type (runtime type) of the object into account. The default is static binding, since it is more efficient and can be done entirely at compile time.

🏆 In order to get dynamic binding instead, we need to declare the member function as *virtual* in the base class.

比如，

```
1  class Bird {
2      //...
3      virtual void talk() const {
4          cout << "tweet" << endl;
5      }
6  };
```

再考虑如下代码：

```
1  void all_talk(Bird *birds[], int length) {
2      for (int i = 0; i < length; ++i) {
3          array[i]->talk();
4      }
5  }
6  int main() {
7      Chicken c1 = /* ... */;
8      Duck d = /* ... */;
9      Chicken c2 = /* ... */;
10     Bird *array[] = { &c1, &d, &c2 };
11     all_talk(array, 3);
12 }
```

因为在父类中，`talk` 被声明为 `virtual`，所以编译器会使用 `array[i]` 的动态类型，分别为 `Chicken`，`Duck`，`Chicken`，也就是说，`array[i]->talk()` 分别调用的是这些子类的 `talk` 方法。

同一个接口，不同的对象，具体的行为也不一样，这就是多态。

🏆 The `virtual` keyword is necessary in the base class, but optional in the derived classes. It can only be applied to the declaration within a class; if the function is subsequently defined outside of the class, the definition cannot include the `virtual` keyword.

如,

```
1  class Bird {
2      ...
3      virtual void talk() const;
4  };
5
6  void Bird::talk() const {
7      cout << "bawwk" << endl;
8  }
```

## 4.3 Member Lookup Revisited

🏆 We have already seen that when a member is accessed on an object, the compiler first looks in the object's class for a member of that name before proceeding to its base class. With indirection, the following is the full lookup process:

1. The compiler looks up the member in the **static type** of the receiver object, using the lookup process we discussed before (starting in the class itself, then looking in the base class if necessary). It is an error if no member of the given name is found in the static type or its base types.
2. If the member found is an overloaded function, then the arguments of the function call are used to determine which overload is called.
3. If the member is a variable or non-virtual function (including static member functions, which we will see later), the access is statically bound at compile time.
4. If the member is a virtual function, the access uses dynamic binding. At runtime, the program will look for a **function of the same signature**, starting at the dynamic type of the receiver, then proceeding to its base type if necessary.

## 4.4 The `override` Keyword

🏆 A common mistake when attempting to override a function is to inadvertently change the signature, so that the derived-class version hides rather than overrides the base-class one.

比如，

```
1  class Chicken : public Bird {
2      ...
3      virtual void talk() {
4          cout << "bawwk" << endl;
5      }
6  };
7
8  int main() {
9      Chicken chicken("Myrtle");
10     Bird *b_ptr = &chicken;
11     b_ptr->talk();
12 }
```

对于这个例子，控制台会输出 `tweet`，而不是 `bawwk`。因为 `Chicken::talk` 的函数签名和 `Bird::talk` 的函数签名不同，后者有 `const` 修饰。为了解决这种隐藏较深的bug，C++为我们提供了关键字 `override`。

🏆 Specifically, we can place the `override` keyword after the signature of a member function to let the compiler know we intended to override a base-class member function. If the derived-class function doesn't actually do so, the compiler will report error.

比如下面的代码就会报错：

```
1  class Chicken : public Bird {
2      ...
3      void talk() override { // 可以省略virtual, 因为只有virtual function才能被
                                override
4          cout << "bawwk" << endl;
5      }
6  };
```

## 4.5 Abstract Classes and Interfaces

有些时候，基类的目的就只是为派生类定义接口规范，并不提供任何的实现，接口的实现交给派生类完成，这时基类也不应该能被实例化，C++提供了纯虚函数来实现这一机制。



The syntax for declaring a function as pure virtual is to put `= 0;` after its signature.

当一个类中声明了纯虚函数，这个类就不能实例化，称这样的类为抽象类。



With a virtual function, a base class provides its derived classes with the option of overriding the function's behavior. With a pure virtual function, the base class **requires** its derived classes to override the function, since the base class does not provide an implementation itself. If a derived class fails to override the function, the derived class is itself abstract, and objects of that class cannot be created.

当一个类中的成员函数全部被声明为纯虚函数，这时它就是一个标准的接口。

## 5. Command-Line Arguments



Command-line arguments are arguments that are passed to a program when it is invoked from a shell or terminal.

考虑如下例子：

```
1 g++ -Wall -O1 -std=c++17 -pedantic test.cpp -o test
```

在这个例子中，`g++` 是我们正在调用的程序，剩余的参数告诉 `g++` 做些什么。比如，`-Wall` 参数告诉 `g++` 编译器警告代码中任何可能的问题，`-O1` 参数告诉编译器使用一级优化等等。

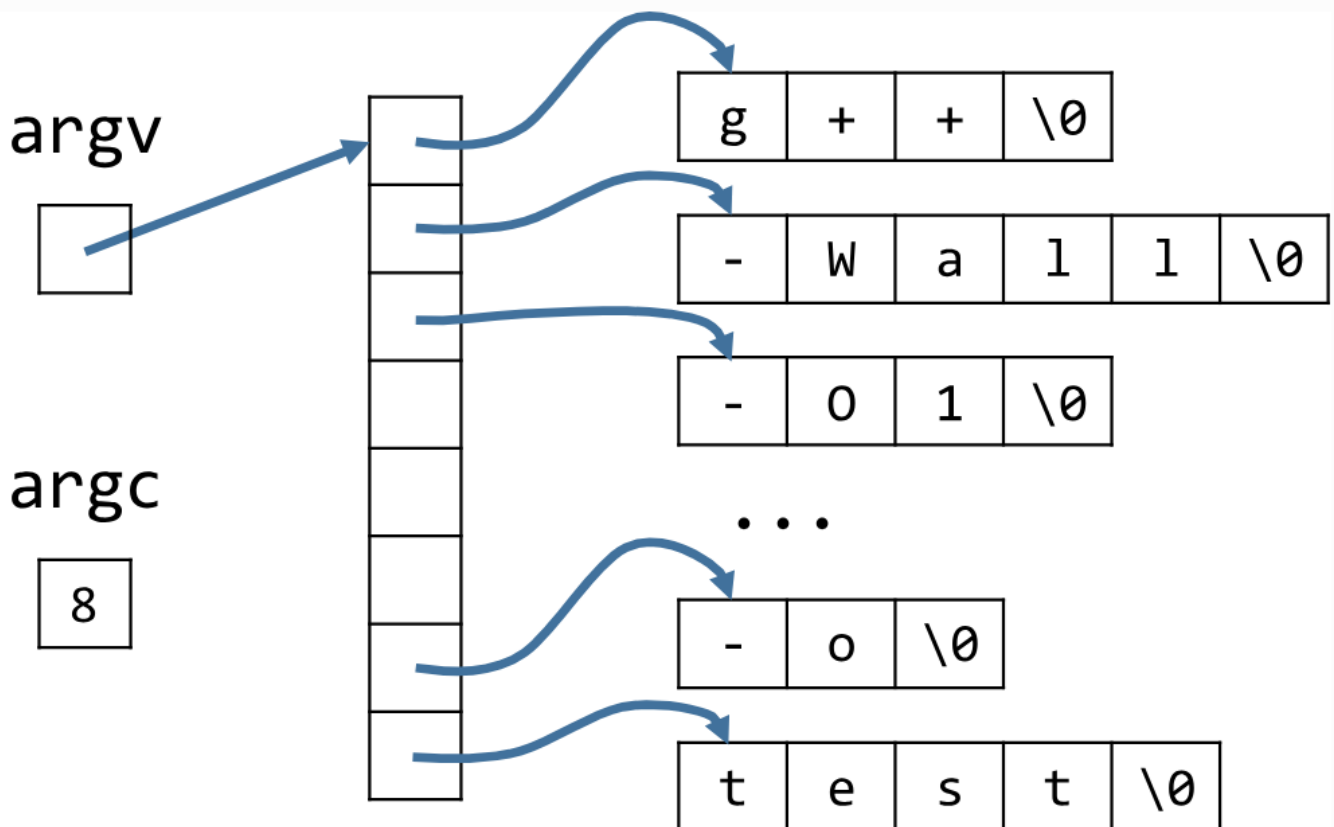


Command-line arguments are passed to the program through arguments to `main()`. The `main()` function may have zero parameter, in which case the command-line arguments are discarded. It can also have two parameters. So the signature has the following form:

```
1 int main(int argc, char *argv[]);
```

🏆 The first parameter is the number of command-line arguments passed to the program, and it is conventionally named `argc(argument count)`. The second, conventionally named `argv(argument vector)`, contains each command-line argument as C-style string.

The arguments passed to the `main` function also include the name of the program as the first argument - this is often used in printing out error messages from the program.



## 6. Input and Output(I/O)

除了通过命令行参数来获取用户输入，也可以通过标准输入获取用户的输入，标准输入接收用户在控制台中输入的数据。

🏆 In C++, the `cin` stream reads data from *standard input*. Data is extracted into an object by using the *extraction operator* `>>`, and the extraction interprets the raw character data according to the target data type.

而标准输出中的数据可以展示在控制台中。



In C++, the `cout` stream writes data to *standard output*. We can use the *insertion operator* `<<` inserts the text representation of a value into it.