C++ 面向对象

极夜酱

目录

Chapter 1 C++ 基础

1.1 数据类型

1.1.1 C++ 简介

C++ 由 Bjarne Stroustrup 于 1979 年在贝尔实验室发明，C++ 在 C 语言的基础上引起并扩充了面向对象的概念，最初命名为带类的 C（C with classes），后更名为 C++。C++ 应用非常广泛，常用于系统开发、引擎开发等领域，支持类、封装、继承、多态等特性。

Hello World!

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```

运行结果

```
Hello World!
```

头文件 iostream 库声明了用于在标准输入输出设备上操作的对象，头文件中包含 cin（标准输入流）、cout（标准输出流）、cerr（标准错误流）、clog（标准日志流）对象。

using namespace std 告诉编译器使用 std 命名空间，命名空间是 C++ 中新的概念。

1.1.2 关键字

C++ 中的关键字不能作为常量名、变量名、或其它标识符名称。

asm	else	new	this	auto
enum	operator	throw	bool	explicit
private	true	break	export	protected
try	case	extern	public	typedef
catch	false	register	typeid	char
float	reinterpret_cast	typename	class	for
return	union	const	friend	short
unsigned	const_cast	goto	signed	using
continue	if	sizeof	virtual	default
inline	static	void	delete	int
static_cast	volatile	do	long	struct
wchar_t	double	mutable	switch	while
dynamic_cast	namespace	template		

表 1.1: 关键字

1.1.3 数据类型

一些基本数据类型可以使用 signed 和 unsigned 修饰符进行修饰。

类型	占用内存	范围
char	1 字节	$-128 \sim 127$
unsigned char	1 字节	$0 \sim 255$
signed char	1 字节	$-128 \sim 127$
int	4 字节	$-2147483648 \sim 2147483647$
unsigned int	4 字节	$0 \sim 4294967295$
signed int	4 字节	$-2147483648 \sim 2147483647$
short	2 字节	$-32768 \sim 32767$
unsigned short	2 字节	$0 \sim 65535$
signed short	2 字节	$-32768 \sim 32767$
long	4 字节	$-2147483648 \sim 2147483647$
signed long	4 字节	$0 \sim 4294967295$
unsigned long	4 字节	$-2147483648 \sim 2147483647$
long long	8 字节	$-9223372036854775808 \sim 9223372036854775807$
signed long long	8 字节	$-9223372036854775808 \sim 9223372036854775807$
unsigned long long	8 字节	$0 \sim 18446744073709551615$
float	4 字节	$-3.4e38 \sim 3.4e38$
double	8 字节	$-1..7e308 \sim 1.7e308$

表 1.2: 数据类型

数据类型

```cpp
#include <iostream>
#include <limits>
using namespace std;

int main() {
    cout << "int" << endl;
    cout << "size: " << sizeof(int) << endl;
    cout << "min: " << numeric_limits<int>::min() << endl;
    cout << "max: " << numeric_limits<int>::max() << endl;
    cout << "--------------------------------" << endl;
    cout << "short" << endl;
```

```
12      cout << "size: " << sizeof(short) << endl;
13      cout << "min: " << numeric_limits<short>::min() << endl;
14      cout << "max: " << numeric_limits<short>::max() << endl;
15      cout << "----------------------------------" << endl;
16      cout << "long" << endl;
17      cout << "size: " << sizeof(long) << endl;
18      cout << "min: " << numeric_limits<long>::min() << endl;
19      cout << "max: " << numeric_limits<long>::max() << endl;
20      cout << "----------------------------------" << endl;
21      cout << "long long" << endl;
22      cout << "size: " << sizeof(long long) << endl;
23      cout << "min: "
24          << numeric_limits<long long>::min() << endl;
25      cout << "max: "
26          << numeric_limits<long long>::max() << endl;
27      cout << "----------------------------------" << endl;
28      cout << "float" << endl;
29      cout << "size: " << sizeof(float) << endl;
30      cout << "min: " << numeric_limits<float>::min() << endl;
31      cout << "max: " << numeric_limits<float>::max() << endl;
32      cout << "----------------------------------" << endl;
33      cout << "double" << endl;
34      cout << "size: " << sizeof(double) << endl;
35      cout << "min: " << numeric_limits<double>::min() << endl;
36      cout << "max: " << numeric_limits<double>::max() << endl;
37      cout << "----------------------------------" << endl;
38      cout << "char" << endl;
39      cout << "size: " << sizeof(char) << endl;
40      cout << "----------------------------------" << endl;
41      cout << "bool" << endl;
42      cout << "size: " << sizeof(bool) << endl;
43      cout << "min: " << numeric_limits<bool>::min() << endl;
44      cout << "max: " << numeric_limits<bool>::max() << endl;
45      cout << "----------------------------------" << endl;
46
47      return 0;
48  }
```

1.2 命名空间

1.2.1 命名空间（namespace）

多人共同开发时，经常会出现变量和函数定义冲突，例如编写了相同名称的函数。命名空间用于解决这个问题，它可以作为附加信息来区分不同库中的变量和函数。

namespace 表示了标识符的可见范围，C++ 标准库中的所有标识符都定义于一个名为 std 的 namespace 中。如果不使用 using namespace std，就必须使用 std::cout、std::endl 等表示其中的标识符。

命名空间

```cpp
#include <iostream>

namespace foo {
    int n = 123;
    void f() {
        std::cout << "foo::f()" << std::endl;
    }
};

namespace bar {
    int n = 456;
    void f() {
        std::cout << "bar::f()" << std::endl;
    }
};

int main() {
    std::cout << foo::n << std::endl;
    foo::f();
    std::cout << bar::n << std::endl;
    bar::f();
```

```
22      return 0;
23  }
```

运行结果

```
123
foo::f()
456
bar::f()
```

1.3 结构体与共同体

1.3.1 结构体

结构体是一种用户自定义的数据类型，它允许存储不同类型的数据项。结构体的声明使用关键字 struct，声明通常定义为全局变量，这样就可以被多个函数所使用的了。

通过成员运算符【.】可以访问一个结构体之中的成员变量。

关键字 typedef 可以用来给数据类型定义别名，通过使用 typedef 可以简化结构体的声明，不用每次都加上 struct 关键字了。

结构体

```
#include <iostream>

using namespace std;

typedef struct {
    int year;
    int month;
    int day;
} Date;

int main() {
    Date date;
    date.year = 2021;
    date.month = 8;
    date.day = 11;
    cout << date.year << "/"
        << date.month << "/"
        << date.day << endl;
    return 0;
}
```

运行结果

```
2021/8/11
```

结构体也可以作为函数参数进行传递。如果是按值传递，那么在函数中会新创建一个结构体变量，并复制调用者的结构体的值。如果是按址传递，则需要传递结构体的指针。

间接引用运算符【->】可以直接访问结构体指针所指的结构变量中的成员。

1.3.2 共同体

在存储多个成员信息时，编译器会自动给 struct 每个成员分配存储空间，struct 可以存储多个成员信息。而 union 每个成员会用同一个存储空间，只能存储一个成员的信息。

在任何同一时刻，union 只存放了一个被先选中的成员，而结构体的所有成员都存在。对于 ‘union‘的不同成员赋值，将会对其它成员重写，原来成员的值就不存在了，而对于 struct 的不同成员赋值是互不影响的。

1.4 常量指针与指针常量

1.4.1 常（量）指针

常（量）指针是指在定义指针变量时，在数据类型前用 const 修饰。

定义一个常指针后，就不能通过指针去更改所指向的变量的值，但是指针的指向可以改变。

常指针

```
#include <iostream>

using namespace std;

int main() {
    int a = 123;
    const int *p = &a;  // 等价于int const *p
    *p = 456;
    return 0;
}
```

运行结果

```
error: assignment of read-only location '* p'
```

1.4.2 指针常量

指针常量是指在定义指针时，在指针变量前用 const 修饰。

指针常量不允许修改，必须要在定义时初始化，之后不能修改指针的指向。

指针常量

```cpp
#include <iostream>

using namespace std;

int main() {
    int a = 123;
    int b = 456;
    int * const p = &a;
    p = &b;
    return 0;
}
```

运行结果

```
error: assignment of read-only variable 'p'
```

1.4.3 常指针常量

常指针常量表示指针的指向不能改变，同时指针指向的值也不能改变。

常指针常量

```cpp
#include <iostream>

using namespace std;

int main() {
    int a = 123;
    int b = 456;
    const int * const p = &a;   // 等价于int const * const p
    *p = 789;
    p = &b;
    return 0;
```

```
12 }
```

运行结果

```
error: assignment of read-only location '*(const int*)p'
error: assignment of read-only variable 'p'
```

1.5 内联函数

1.5.1 内联函数（Inline Function）

内联函数是在编译时期展开，不用执行进入函数的步骤，直接执行函数体，相当于把内联函数里面的内容写在调用内联函数处。编译器一般不内联包含循环、递归、switch 等复杂操作的函数。是否内联，程序员不可控，内联函数只是对编译器的建议，是否对函数内联，决定权在于编译器。

内联函数

```cpp
#include <iostream>

using namespace std;

inline int max(int num1, int num2) {
    return num1 > num2 ? num1 : num2;
}

int main() {
    cout << max(92, 28) << endl;
    return 0;
}
```

运行结果

```
92
```

编译器对内联函数的处理首先是将内联函数复制到调用处，并为局部变量分配内存，将输入参数和返回值映射到局部变量空间中。如果内联函数有多个返回点，将会使用 goto 语句跳转到代码块的末尾。

使用内联函数的优点在于同宏函数一样将在被调用处进行代码展开，省去了参数压栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度。并且内联函数相

比宏函数来说，在代码展开时，会做安全检查或自动类型转换，而宏定义则不会。在类中声明同时定义的成员函数，自动转化为内联函数，因此内联函数可以访问类的成员变量。

但是内联是以代码膨胀（复制）为代价，消除函数调用带来的开销，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。并且内联函数无法随着函数库的升级而升级，内联函数的改变需要重新编译，不像普通函数可以直接链接。

1.6 重载函数

1.6.1 函数默认参数

在进行函数参数定义的时候，也可以设置默认值。当参数没有传递的时候就利用默认值来进行参数内容的填充，如果在参数上定义了默认值，那么该参数一定要放在参数列表的最后。

函数默认参数

```
#include <iostream>

using namespace std;

void setDate(int year = 1970, int month = 1, int day = 1) {
    cout << year << "/" << month << "/" << day << endl;
}

int main() {
    setDate(2021, 8, 15);
    setDate(2021, 7);
    setDate(2021);
    setDate();
    return 0;
}
```

运行结果

```
2021/8/15
2021/7/1
2021/1/1
1970/1/1
```

1.6.2 重载函数

重载（overload）表示在同一个作用域中声明了一个与之前声明过的函数具有相同名称的函数，但是它们的参数列表不同。当调用一个重载函数时，编译器通过传递的参数类型，选用最合适的定义。

重载函数

```
#include <iostream>
using namespace std;

int max(int num1, int num2) {
    return num1 > num2 ? num1 : num2;
}

double max(double num1, double num2) {
    return num1 > num2 ? num1 : num2;
}

char max(char num1, char num2) {
    return num1 > num2 ? num1 : num2;
}

int main() {
    cout << max(2, 8) << endl;
    cout << max(3.14, 2.71) << endl;
    cout << max('H', 'D') << endl;
    return 0;
}
```

运行结果

```
8
3.14
H
```

1.7 引用

1.7.1 引用（Reference）

在 C 语言中【&】表示取地址符，但是在 C++ 中还有引用的特性。引用是指对某一变量的别名，因此引用本身并不占存储单元，对引用的操作与直接对变量的操作完全一样。

```
data_type &ref_var = var;
```

声明引用时必须对其初始化，并且之后不能再将该引用作为其它变量的别名。

引用的一个重要作用就是作为函数的参数。C 语言中函数参数传递是按值传递，如果有大块数据作为参数传递的时候，往往采用指针的方式，因为这样可以避免将整块数据全部压栈，可以提高程序的效率。但是 C++ 中传递引用给函数与传递指针的效果是一样的，使用引用传递函数的参数，在内存中并没有产生实参的副本，而是直接对实参操作。

引用

```
#include <iostream>

using namespace std;

void swap(int& num1, int& num2) {
    int temp = num1;
    num1 = num2;
    num2 = temp;
}

int main() {
    int a = 92;
    int b = 28;
    cout << "Before: " << a << " " << b << endl;
    swap(a, b);
```

```
16      cout << "After: " << a << " " << b << endl;
17      return 0;
18  }
```

运行结果

```
Before: 92 28
After: 28 92
```

指针与引用的区别：

- 指针有自己的一块空间，而引用只是一个别名，对引用的操作等效于对原变量的操作。
- 指针的大小为 4 或 8（根据 OS），而引用则是被引用对象的大小。
- 指针可以被初始化为 NULL，而引用必须被初始化且必须是一个已有对象的引用。
- 作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引用的修改都会改变引用所指向的对象。
- 引用在初始化后，便不能再更改引用的目标。
- 指针可以有多级指针，而引用只有一级。
- 指针和引用使用【++】运算符的意义不一样。
- 返回动态内存分配的内存时必须使用指针，引用可能引起内存泄漏。

1.8 函数指针

1.8.1 函数指针

函数指针即指向函数的指针，它最大的作用是把一个函数作为参数传递给另外一个函数。在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。

函数指针

```
#include <iostream>

using namespace std;

int add(int num1, int num2) {
    return num1 + num2;
}

int sub(int num1, int num2) {
    return num1 - num2;
}

int get(int num1, int num2, int (*calculate)(int num1, int num2)) {
    return calculate(num1, num2);
}

int main() {
    cout << get(7, 3, add) << endl;
    cout << get(7, 3, sub) << endl;
    return 0;
}
```

运行结果

```
10
4
```

1.8.2 回调函数（Callback Function）

假设公司要开发一款主打解决国民早餐问题的 APP 明日油条，为了加快开发进度，这款应用由 A 小组和 B 小组协同开发。

其中有一个核心模块由 A 小组开发供 B 小组调用，这个核心模块被写成了一个函数 make_youtiao()。

如果 make_youtiao() 执行地很快并可以立即返回，那么 B 小组只需：

1. 调用 make_youtiao()
2. 等待函数执行完成
3. 执行后续流程 sell() 出售

但是现实中 make_youtiao() 函数需要处理的数据非常庞大，例如 make_youtiao(10000) 不会立刻返回，而是需要等待 10 分钟才能执行完成。显然直接调用的话，需要等待 10 分钟后才能继续执行后续代码，这并不是一种高效的做法。

因此一种更好的做法是调用 make_youtiao() 后不再等待这个函数执行完成，而是让这个函数知道制作完油条后该干什么，例如“制作 10000 个油条，然后卖出去”。

因此 make_youtiao() 需要增加一个参数，除了指定制作油条的数量外，还可以指定制作好后该干什么，这个被 make_youtiao() 调用的函数就叫回调。

回调函数

```cpp
#include <iostream>

using namespace std;

void sell(int n) {
    cout << "selling youtiao ..." << endl;
}

void make_youtiao(int n, void (*sell)(int)) {
    cout << "making youtiao ..." << endl;
    for(int i = 0; i < n; i++);
    sell(n);
}

int main() {
    make_youtiao(10000, sell);
    return 0;
}
```

运行结果

```
making youtiao ...
selling youtiao ...
```

1.9 内存管理

1.9.1 malloc() / free()

C 语言使用 malloc() 分配内存，使用 free() 释放已分配的内存。当 malloc() 分配的内存过多/不够时，可以使用 realloc() 重新分配内存。

malloc() 的功能是向系统申请指定的内存空间（以字节为单位），使用该函数需要包含头文件 stdlib.h。

```
1 void* malloc(size_t size);
```

malloc() 的返回值为 void * 类型，表示一个指向申请到的空间的首地址，是一个无类型的指针，开发者需要自行转换为自己需要的类型。如果 malloc() 申请内存失败，则会返回空指针 NULL。

通过 malloc() 申请来的空间是需要归还给操作系统的，否则程序长时间运行内存会逐渐下降。通过 free() 可以把申请来的空间释放，但是有两点需要注意：

1. 只能释放通过 malloc() 申请得到的空间。
2. 只能通过空间的首地址进行释放。

malloc() / free()

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  int main() {
7      const int n = 10;
8      int *arr = (int *)malloc(sizeof(int) * n);
9      if(!arr) {
10         cerr << "Memory allocation failed." << endl;
```

```
11          return 1;
12      }
13      for(int i = 0; i < n; i++) {
14          arr[i] = i;
15          cout << arr[i] << " ";
16      }
17      cout << endl;
18      free(arr);
19      return 0;
20  }
```

运行结果

```
0 1 2 3 4 5 6 7 8 9
```

1.9.2 new / delete

C++ 中可以通过 new 运算符实现动态内存分配，如果空间内配失败，程序则抛出 bad_alloc 异常。

```
1  T *p = new T;
2  T *p = new T[N];
```

其中 T 为任意类型名，p 是类型为 T * 的指针。

通过 new 动态内存分配所得的空间在使用完后需要使用 delete 运算符释放空间。

```
1  delete p;
2  delete[] p;
```

如果动态分配 new 一个数组，但是却用 delete 释放，而不是 delete[]，会导致动态分配的数据没有被完全释放。如果动态申请的内存空间不再使用后没有及时释放，就会导致内存泄漏（memory leak）。

new / delete

```cpp
#include <iostream>

using namespace std;

int main() {
    int *p1 = new int;
    *p1 = 928;
    cout << "*p1 = " << *p1 << endl;

    const int n = 10;
    int *p2 = new int[n];
    cout << "p2: ";
    for(int i = 0; i < n; i++) {
        p2[i] = i;
        cout << p2[i] << " ";
    }
    cout << endl;

    delete p1;
    delete[] p2;
    return 0;
}
```

运行结果

```
*p1 = 928
p2: 0 1 2 3 4 5 6 7 8 9
```

1.9.3 malloc() / free() 与 new / delete 区别

- new 分配内存按照数据类型，而 malloc() 按照指定指定大小。
- new 返回指定对象的指针，而 malloc() 返回 void *，因此 malloc() 的返回值一般都需要类型转换。

- new 是一个操作符，malloc() 是一个库函数。
- new 分配失败抛出 bad_alloc 异常，而 malloc() 失败返回 NULL。

1.9.4 内存管理

虚拟内存中栈区的内存是由系统自动分配的，不需要程序员对其进行管理，一般用于存储函数的返回地址、参数、局部变量和返回值。而堆区的内存是动态分配的，并且需要手动释放内存。

栈是运行时的单位，堆是存储的单位。栈解决程序运行问题（如何执行），堆解决数据存储的问题（如何存储）。

内存通常包括了栈区（stack）、堆区（heap）、数据区、程序代码区：

- 栈区：由编译器自动分配和释放，存放函数的参数值、局部变量的值等。
- 堆区：一般由程序员分配和释放，若程序员不释放，程序结束后被 OS 回收。
- 数据区：存放全局变量和静态变量，程序结束后由系统释放。
- 程序代码区：存放函数体的二进制代码。

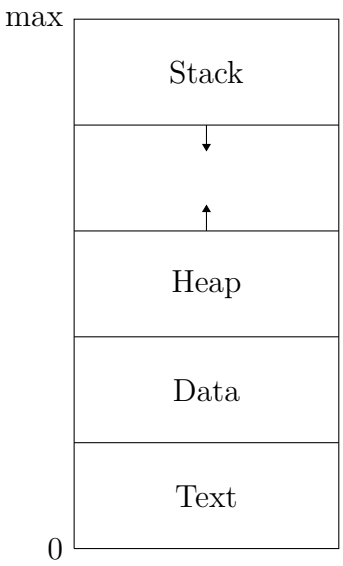

图 1.1: 内存管理

Chapter 2 封装

2.1 面向过程与面向对象

2.1.1 面向过程（Procedure Oriented）

面向过程是一种以过程为中心的编程思想，以什么正在发生为主要目标进行编程，分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用。

C 语言就是一种面向过程的编程语言，但是面向过程的缺陷是数据和函数并不完全独立，使用两个不同的实体表示信息及其操作。

2.1.2 面向对象（Object Oriented）

面向对象是相对于面向过程来讲的，面向对象方法把相关的数据和方法组织为一个整体来看待，从更高的层次来进行系统建模，更贴近事物的自然运行模式。

在面向对象中，把构成问题的事物分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。

Java、C++、Python 等都是面向对象的编程语言，面向对象的优势在于只是用一个实体就能同时表示信息及其操作。

面向对象三大特性：

1. 封装（encapsulation）：数据和代码捆绑，避免外界干扰和不确定性访问。
2. 继承（inheritance）：让某种类型对象获得另一类型对象的属性和方法。
3. 多态（polymorphism）：同一事物表现出不同事物的能力。

2.2 类与对象

2.2.1 类与对象

类（class）表示同一类具有相同特征和行为的对象的集合，类定义了对象的属性和方法。

对象（object）是类的实例，对象拥有属性和方法。

类的设计需要使用关键字 class，类名是一个标识符，遵循大驼峰命名法。类中可以包含属性和方法。其中，属性通过变量表示，又称实例变量；方法用于描述行为，又称实例方法。

通过关键字 new 进行对象的实例化，实例化对象会调用类中的构造函数完成。类是一种引用数据类型，对象的实例化在堆上开辟空间。

类和对象

```
#include <iostream>
#include <string>

using namespace std;

class Person {
public:
    string name;
    int age;

    void eat() {
        cout << "吃饭" << endl;
    }

    void sleep() {
        cout << "睡觉" << endl;
```

```
    }
};

int main() {
    Person person;

    person.name = "小灰";
    person.age = 17;
    cout << "姓名: " << person.name << endl;
    cout << "年龄: " << person.age << endl;

    person.eat();
    person.sleep();
    return 0;
}
```

运行结果

```
姓名: 小灰
年龄: 17
吃饭
睡觉
```

2.3 封装

2.3.1 封装（Encapsulation）

封装是面向对象方法的重要原则，就是把对象的属性和方法结合为一个独立的整体，并尽可能隐藏对象的内部实现细节。

封装可以认为是一个保护屏障，防止该类的数据被外部类随意访问。要访问该类的数据，必须通过严格的接口控制。合适的封装可以让代码更容易理解和维护，也加强了程序的安全性。

实现封装的步骤：

1. 修改属性的可见性来限制对属性的访问，一般限制为 private。
2. 对每个属性提供对外的公共方法访问，也就是提供一对 setter / getter，用于对私有属性的访问。

2.3.2 访问权限

属性和方法的访问权限一般分为 3 种：

1. public：属性和方法在类的内部和外部都可以访问。
2. private：属性和方法只能在类内访问。
3. protected：属性和方法只能在类的内部和其派生类中访问。

2.3.3 this 指针

每一个对象都能通过 this 指针来访问自身的地址，this 指针是所有成员方法的隐含参数，在成员方法内部可以用来指向调用对象。

在类中，属性的名字可以和局部变量的名字相同。此时，如果直接使用名字来访问，优先访问的是局部变量。因此，需要使用 this 指针来访问当前对象的属性。

当需要访问的属性与局部变量没有重名的时候，this 可以省略。

封装

```cpp
#include <iostream>
#include <string>

using namespace std;

class Person {
public:
    void setName(string name) {
        this->name = name;
    }

    string getName() {
        return name;
    }

    void setAge(int age) {
        this->age = age;
    }

    int getAge() {
        return age;
    }

private:
    string name;
    int age;
};

int main() {
```

```
30      Person person;
31
32      person.setName("小灰");
33      person.setAge(17);
34
35      cout << "姓名: " << person.getName() << endl;
36      cout << "年龄: " << person.getAge() << endl;
37      return 0;
38  }
```

运行结果

```
姓名: 小灰
年龄: 17
```

2.4 构造函数与析构函数

2.4.1 构造函数（Constructor）

构造函数也是一个函数，用于实例化对象，在实例化对象的时候调用。一般情况下，使用构造函数是为了在实例化对象的同时，给一些属性进行初始化赋值。

构造函数和普通函数的区别：

1. 构造函数的名字必须和类名一致。
2. 构造函数没有返回值，返回值类型部分不写。

如果一个类中没有构造函数，系统会自动提供一个 public 权限的无参构造函数以便实例化对象。如果一个类中已有构造函数，系统将不再提供任何默认的构造函数。

构造函数

```
#include <iostream>
#include <string>

using namespace std;

class Person {
public:
    Person();
    Person(string name, int age);
    string toString();

private:
    string name;
    int age;
};

// 无参构造函数
```

```cpp
Person::Person() {
    cout << "Person::Person()" << endl;
}

// 有参构造函数
Person::Person(string name, int age) {
    cout << "Person::Person(string, int)" << endl;
    this->name = name;
    this->age = age;
}

string Person::toString() {
    return "姓名：" + name + "，年龄：" + to_string(age);
}

int main() {
    Person p1;
    Person p2("小灰", 17);
    cout << p2.toString() << endl;
    return 0;
}
```

运行结果

```
Person::Person()
Person::Person(string, int)
姓名：小灰，年龄：17
```

2.4.2 初始化列表

与其它函数不同，构造函数还可以有初始化列表。初始化列表以【:】开头，后跟一些列以逗号分割的初始化字段。

初始化列表

```cpp
#include <iostream>
#include <string>

using namespace std;

class Person {
public:
    Person(string name, int age);
    string toString();

private:
    string name;
    int age;
};

Person::Person(string name, int age) : name(name), age(age) {
    cout << "Person::Person(string, int)" << endl;
}

string Person::toString() {
    return "姓名：" + name + "，年龄：" + to_string(age);
}

int main() {
    Person person("小灰", 17);
    cout << person.toString() << endl;
    return 0;
}
```

运行结果

```
Person::Person(string, int)
姓名：小灰，年龄：17
```

有些时候初始化列表是不可或缺的，以下情况必须使用初始化列表：

1. 常量成员：常量只能初始化不能赋值。
2. 引用类型：引用必须在定义时初始化，且不能重新赋值。
3. 没有默认构造函数的类类型：使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化。

2.4.3 析构函数（Destructor）

析构函数与构造函数相反，当对象的生命周期结束时，会自动执行析构函数，用于做清理善后的事情。

析构函数的名称以【 】为前缀，后加类名称，它没有返回值和参数。

析构函数

```
#include <iostream>
#include <string>

using namespace std;

class Person {
public:
    Person(string name, int age);
    ~Person();

private:
    string name;
    int age;
};

Person::Person(string name, int age) : name(name), age(age) {
    cout << "Person::Person(string, int)" << endl;
}

```

```
20 Person::~Person() {
21     cout << "Person::~Person()" << endl;
22 }
23
24 int main() {
25     Person p1("小灰", 17);
26     Person *p2 = new Person("小白", 21);
27     delete p2;
28     return 0;
29 }
```

运行结果

```
Person::Person(string, int)
Person::Person(string, int)
Person::~Person()
Person::~Person()
```

2.4.4 拷贝构造函数（Copy Constructor）

拷贝构造函数是构造函数的一种，它只有一个参数，参数类型为本类的引用。参数可以使 const 引用，也可以是非 const 引用，但是一般使用前者。

如果没有编写拷贝构造函数，编译器会自动生成一个默认的拷贝构造函数。

拷贝构造函数

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 class Person {
```

```
public:
    Person(const Person &p);
    Person(string name, int age);

private:
    string name;
    int age;
};

Person::Person(const Person &p) {
    cout << "Person::Person(const Person &)" << endl;
    this->name = p.name;
    this->age = p.age;
}

Person::Person(string name, int age) {
    cout << "Person::Person(string, int)" << endl;
    this->name = name;
    this->age = age;
}

int main() {
    Person p1("小灰", 17);
    Person p2(p1);
    Person p3 = p1;
    return 0;
}
```

运行结果

```
Person::Person(string, int)
Person::Person(const Person &)
Person::Person(const Person &)
```

拷贝构造函数会在三种情况下被调用：

1. 用一个对象去初始化同类的另一个对象。

2. 函数参数是类的对象。

3. 函数的返回值是类的对象。

2.4.5 浅拷贝 / 深拷贝

当使用浅拷贝（shallow copy）时，仅仅是拷贝指针字面值，如果原来的对象调用析构函数释放掉指针所指向的数据，则会产生空悬指针（dangling pointer），因为所指向的内存空间已经被释放了。

浅拷贝

```
#include <iostream>

using namespace std;

class User {
public:
    User();
    ~User();
    void printDataAddress();

private:
    int *data;
};

User::User() {
    this->data = new int;
}

User::~User() {
    delete data;
    data = nullptr;
}

```

```
24 void User::printDataAddress() {
25     cout << data << endl;
26 }
27
28 int main() {
29     User user1;
30     user1.printDataAddress();
31     User user2(user1);
32     user2.printDataAddress();
33     return 0;
34 }
```

运行结果

```
0x26c2b90
0x26c2b90
```

深拷贝（deep copy）可以解决浅拷贝出现的问题，通过定义一个拷贝构造函数，当被拷贝对象存在动态分配的存储空间时，需要先动态申请一块存储空间，然后逐字节拷贝内容。

深拷贝

```
1  #include <iostream>
2
3  using namespace std;
4
5  class User {
6  public:
7      User();
8      User(const User& user);
9      ~User();
10     void printDataAddress();
11
12 private:
```

```
13      int *data;
14  };
15
16  User::User() {
17      this->data = new int;
18  }
19
20  User::User(const User& user) {
21      this->data = new int;
22      *(this->data) = *(user.data);
23  }
24
25  User::~User() {
26      delete data;
27      data = nullptr;
28  }
29
30  void User::printDataAddress() {
31      cout << data << endl;
32  }
33
34  int main() {
35      User user1;
36      user1.printDataAddress();
37      User user2(user1);
38      user2.printDataAddress();
39      return 0;
40  }
```

运行结果

```
0x6b17b0
0x6b17d0
```

2.5 静态成员

2.5.1 静态成员

类的静态成员在编译时创建并初始化，在该类的任何对象建立之前就已经存在。静态成员不属于任何对象，并且在类中只有一份，为所有此类对象共享。

在静态成员函数的实现中不能直接引用类中的非静态成员，但可以引用类中的静态成员。如果静态成员函数中要引用非静态成员时，需要通过对象来引用。

静态成员

```
#include <iostream>
#include <string>

using namespace std;

class User {
public:
    User(int id, string name) : id(id), name(name) {
        totalUsers++;
    }

    static int getTotalUsers() {
        return totalUsers;
    }

private:
    static int totalUsers;
    int id;
    string name;
};

int User::totalUsers = 0;       // 初始用户数量

```

```
24 int main() {
25     cout << User::getTotalUsers() << endl;
26 
27     for(int i = 0; i < 10; i++) {
28         User user(i, "User-" + to_string(i));
29     }
30 
31     cout << User::getTotalUsers() << endl;
32     return 0;
33 }
```

运行结果

```
0
10
```

2.6 友元

2.6.1 友元函数

封装使得类的数据对外隐藏，但是有些函数不是类的一部分，却又需要频繁访问类的数据成员，这时可以将这些函数定义为该类的友元函数。一个函数可以是多个类的友元函数，只需要在各个类中分别声明。除了友元函数，还有友元类。

友元（friend）的作用是提高程序的运行效率，减少了类型检查和安全性检查等需要的时间开销，但它破坏了类的封装性和隐藏性，使得非成员函数可以访问类的私有成员。

友元函数是可以直接访问类的私有成员的非成员函数。它是定义在类外的普通函数，它不属于任何类，但需要在类的定义中加以声明。

```
1 friend ret_type func_name([param_list]);
```

友元函数

```
1  #include <iostream>
2  #include <cmath>
3
4  using namespace std;
5
6  class Coordinate {
7  public:
8      Coordinate(double x, double y) : x(x), y(y) {};
9
10     friend double distance(Coordinate &c1, Coordinate &c2);
11
12 private:
13     double x;
14     double y;
15 };
16
```

```
17 double distance(Coordinate &c1, Coordinate &c2) {
18     double deltaX = c1.x - c2.x;
19     double deltaY = c1.y - c2.y;
20     return sqrt(deltaX * deltaX + deltaY * deltaY);
21 }
22
23 int main() {
24     Coordinate c1(3, 5);
25     Coordinate c2(4, 6);
26     cout << distance(c1, c2) << endl;
27     return 0;
28 }
```

运行结果

```
1.41421
```

2.6.2 友元类

友元类的所有成员函数都是另一个类的友元函数，可以访问另一个类中的隐藏信息。当一个类想要存取另一个类的私有成员时，可以将该类声明为另一类的友元类。

```
1 friend class class_name;
```

友元有以下需要注意的地方：

1. 友元关系不能被继承。
2. 友元关系是单向的，不具有交换性。如果 A 是 B 的友元，B 不一定是 A 的友元。
3. 友元关系不具有传递性。如果 A 是 B 的友元，C 是 A 的友元，那么 C 不一定是 B 的友元。

2.7 运算符重载

2.7.1 运算符重载

C++ 中预定义的运算符的操作对象只能是基本数据类型，但实际上对于许多用户自定义类型（例如类），也需要类似的运算操作。这时就必须在 C++ 中重新定义这些运算符，赋予已有运算符新的功能，使它能够用于特定类型执行特定的操作。运算符重载的实质是函数重载，它提供了可扩展性。

运算符重载是通过创建运算符函数实现的，运算符函数定义了重载的运算符将要进行的操作。运算符函数的定义与其它函数的定义类似，惟一的区别是运算符函数的函数名是由关键字 operator 和要重载的运算符符号构成。

```
ret_type operator op([param_list]) {
    // code
}
```

运算符重载需要遵循以下规则：

1. 除了【.】、【->】、【sizeof】、【?:】和【#】，其它运算符都可以重载。
2. 重载后的运算符不能改变优先级和结合性，也不能概念运算符的操作数个数及语法结构。
3. 运算符重载是针对新类型数据对实际需要的改造，重载后的运算符应当与原有功能相类似。

2.7.2 二元运算符重载

二元运算符需要两个操作数，例如【+】、【-】、【*】、【/】等。

二元运算符重载

```
#include <iostream>
```

```cpp
#include <string>

using namespace std;

class Complex {
public:
    Complex(int real, int imaginary);
    string getNumber();
    Complex operator+(const Complex& c);

private:
    int real;
    int imaginary;
};

Complex::Complex(int real = 0, int imaginary = 0)
    : real(real), imaginary(imaginary) {}

string Complex::getNumber() {
    return to_string(real) + "+" + to_string(imaginary) + "i";
}

Complex Complex::operator+(const Complex& c) {
    Complex complex;
    complex.real = this->real + c.real;
    complex.imaginary = this->imaginary + c.imaginary;
    return complex;
}

int main() {
    Complex c1(1, 2);
    Complex c2(8, 1);
    Complex result = c1 + c2;
    cout << result.getNumber() << endl;
    return 0;
}
```

运行结果

```
9+3i
```

2.7.3 一元运算符重载

一元运算符只对一个操作数操作，例如【++】、【--】、【-】、【!】等。

一元运算符重载

```
#include <iostream>
#include <string>
#include <iomanip>

using namespace std;

class Time {
public:
    Time(int hour, int minute, int second);
    void display();
    Time operator++();      // 前置++
    Time operator++(int);   // 后置++

private:
    int hour;
    int minute;
    int second;
};

Time::Time(int hour, int minute, int second)
    : hour(hour), minute(minute), second(second) {}

void Time::display() {
    cout << setfill('0')
         << setw(2) << hour << ":"
```

```
26          << setw(2) << minute << ":"
27          << setw(2) << second << endl;
28  }
29
30  // 前置++
31  Time Time::operator++() {
32      second++;
33      if(second == 60) {
34          second %= 60;
35          minute++;
36          if(minute == 60) {
37              minute %= 60;
38              hour++;
39              if(hour == 24) {
40                  hour = 0;
41              }
42          }
43      }
44      return Time(hour, minute, second);
45  }
46
47  // 后置++
48  Time Time::operator++(int) {
49      // 保存原始值
50      Time time(hour, minute, second);
51      second++;
52      if(second == 60) {
53          second %= 60;
54          minute++;
55          if(minute == 60) {
56              minute %= 60;
57              hour++;
58              if(hour == 24) {
59                  hour = 0;
60              }
61          }
62      }
```

```
63      return time;    // 返回原始值
64  }
65
66  int main() {
67      Time time(9, 21, 58);
68      time.display();
69
70      ++time;
71      time.display();
72
73      time++;
74      time.display();
75      return 0;
76  }
```

运行结果

```
09:21:58
09:21:59
09:22:00
```

2.7.4 输入输出运算符重载

C++ 使用流提取运算符【>>】和流插入运算符【<<】进行输入输出，通过运算符重载可以对自定义对象进行输入输出操作。通过把输入输出运算符重载函数声明为类的友元，可以直接调用函数而无需创建对象。

输入输出运算符重载

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   class User {
```

```
6  public:
7      User(int id, string name);
8      friend ostream& operator<<(
9          ostream& out,
10         const User& user);
11     friend istream& operator>>(istream& in, User& user);
12
13 private:
14     int id;
15     string name;
16 };
17
18 User::User(int id = 0, string name = "")
19     : id(id), name(name) {}
20
21 ostream& operator<<(ostream& out, const User& user) {
22     out << "ID: " << to_string(user.id) << ", "
23         << "name: " << user.name;
24     return out;
25 }
26
27 istream& operator>>(istream& in, User& user) {
28     cout << "Enter user ID: ";
29     in >> user.id;
30     cout << "Enter user name: ";
31     in >> user.name;
32     return in;
33 }
34
35 int main() {
36     User user;
37     cin >> user;
38     cout << user;
39     return 0;
40 }
```

运行结果

```
Enter user ID: 1
Enter user name: Terry
ID: 1, name: Terry
```

Chapter 3 继承

3.1 继承

3.1.1 继承（Inheritance）

继承是面向对象的三大特征之一，程序中的继承是类与类之间的特征和行为的一种赠予或获取。两个类之间的继承必须满足“is a”的关系。子类继承自父类，父类也称基类或超类，子类也称派生类。

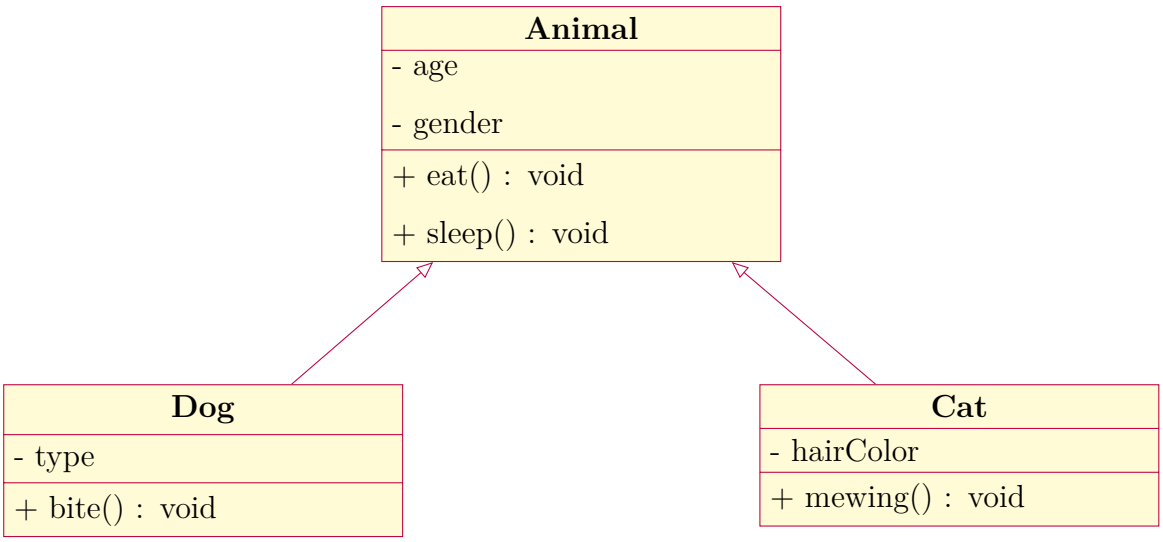

图 3.1: 继承

产生继承关系后，子类可以使用父类中的属性和方法，也可以定义子类独有的属性和方法。

```
1 class subclass : access_modifier superclass {
2     // code
3 };
```

继承时通常使用 public 类型。当一个类 public 继承于父类时，父类的 public 成员也是子类的 public 成员，父类的 protected 成员也是子类的 protected 成员，父类的 private 成员不能被继承。

继承的好处是可以提高代码的复用性、提高代码的拓展性。

继承

animal.h

```h
#ifndef _ANIMAL_H_
#define _ANIMAL_H_

#include <string>

class Animal {
public:
    Animal(std::string name = "", int age = 0);
    void eat();

private:
    std::string name;
    int age;
};

#endif
```

animal.cpp

```cpp
#include "animal.h"
#include <iostream>

using namespace std;

Animal::Animal(string name, int age)
    : name(name), age(age) {}

void Animal::eat() {
    cout << "eating" << endl;
}
```

dog.h

```h
#ifndef _DOG_H
```

```
2  #define _DOG_H_
3
4  #include "animal.h"
5  #include <string>
6
7  class Dog : public Animal {
8  public:
9      Dog(std::string name, int age, std::string type = "");
10     void bite();
11
12 private:
13     std::string type;
14 };
15
16 #endif
```

dog.cpp

```
1  #include "dog.h"
2  #include <iostream>
3
4  using namespace std;
5
6  Dog::Dog(string name, int age, string type)
7      : Animal(name, age), type(type) {}
8
9  void Dog::bite() {
10     cout << "biting" << endl;
11 }
```

test_dog.cpp

```
1  #include <iostream>
2  #include "dog.h"
3
4  using namespace std;
5
6  int main() {
7      Dog dog("狗子", 3, "哈士奇");
```

```
8       dog.eat();
9       dog.bite();
10      return 0;
11  }
```

运行结果

```
eating
biting
```

3.2 多继承

3.2.1 多继承

C++ 支持多继承，即一个子类可以有两个或更多个父类。多继承时通过使用逗号将多个父类隔开，每个父类都可以用不同访问限定符修饰。

当多个父类中有同名的成员时，就会产生命名冲突，因此这时就需要在成员前加上类名和域限定符【::】消除二义性。

多继承

date.h

```h
#ifndef _DATE_H_
#define _DATE_H_

#include <string>

class Date {
public:
    Date(int year = 1970, int month = 1, int day = 1);
    std::string getDate();

private:
    int year;
    int month;
    int day;
};

#endif
```

date.cpp

```cpp
#include "date.h"

using namespace std;

```

```
Date::Date(int year, int month, int day)
    : year(year), month(month), day(day) {}

string Date::getDate() {
    char format[128];
    snprintf(format, sizeof(format),
            "%04d/%02d/%02d", year, month, day);
    string dateStr(format);
    return dateStr;
}
```

time.h

```
#ifndef _TIME_H_
#define _TIME_H_

#include <string>

class Time {
public:
    Time(int hour = 0, int minute = 0, int second = 0);
    std::string getTime();

private:
    int hour;
    int minute;
    int second;
};

#endif
```

time.cpp

```
#include "time.h"

using namespace std;

Time::Time(int hour, int minute, int second)
    : hour(hour), minute(minute), second(second) {}
```

```
7
8  string Time::getTime() {
9      char format[128];
10     snprintf(format, sizeof(format),
11             "%02d:%02d:%02d", hour, minute, second);
12     string timeStr(format);
13     return timeStr;
14 }
```

date_time.h

```
1  #ifndef _DATE_TIME_H_
2  #define _DATE_TIME_H_
3
4  #include "date.h"
5  #include "time.h"
6  #include <string>
7
8  class DateTime : public Date, public Time {
9  public:
10     DateTime(int year = 1970, int month = 1, int day = 1,
11              int hour = 0, int minute = 0, int second = 0);
12     std::string getDateTime();
13
14 private:
15     int year;
16     int month;
17     int day;
18     int hour;
19     int minute;
20     int second;
21 };
22
23 #endif
```

date_time.cpp

```
1  #include "date_time.h"
2
```

```
3  using namespace std;
4
5  DateTime::DateTime(int year, int month, int day,
6          int hour, int minute, int second)
7    : Date(year, month, day),
8      Time(hour, minute, second) {}
9
10 string DateTime::getDateTime() {
11     return getDate() + " " + getTime();
12 }
```

test_date_time.cpp

```
1  #include <iostream>
2  #include "date_time.h"
3
4  using namespace std;
5
6  int main() {
7      DateTime dt1;
8      cout << dt1.getDateTime() << endl;
9      DateTime dt2(2021, 8, 31, 13, 50, 23);
10     cout << dt2.getDateTime() << endl;
11     return 0;
12 }
```

运行结果

```
1970/01/01 00:00:00
2021/08/31 13:50:23
```

3.3 向上转型与向下转型

3.3.1 向上转型 / 向下转型

对象由子类类型转型为父类类型，即是向上转型。向上转型是一种隐式转换，一定会转型成功。向上转型后的对象，只能访问父类中定义的成员。

由父类类型转型转型为子类类型，即是向下转型。向下转型是不安全的，可能会导致数据的丢失，原因是父类的指针或引用中可能不包含子类成员的内存。

向上转型

animal.h

```h
#ifndef _ANIMAL_H_
#define _ANIMAL_H_

#include <string>

class Animal {
public:
    Animal(std::string name = "");
    std::string getName();

private:
    std::string name;
};

#endif
```

animal.cpp

```cpp
#include "animal.h"

using namespace std;

Animal::Animal(string name) : name(name) {}

```

```
7  string Animal::getName() {
8      return name;
9  }
```

dog.h

```
1  #ifndef _DOG_H_
2  #define _DOG_H_
3
4  #include "animal.h"
5  #include <string>
6
7  class Dog : public Animal {
8  public:
9      Dog(std::string name, std::string type = "");
10     std::string getType();
11
12 private:
13     std::string type;
14 };
15
16 #endif
```

dog.cpp

```
1  #include "dog.h"
2
3  using namespace std;
4
5  Dog::Dog(string name, string type)
6      : Animal(name), type(type) {}
7
8  string Dog::getType() {
9      return type;
10 }
```

test_dog.cpp

```
1  #include <iostream>
2  #include "animal.h"
```

```cpp
#include "dog.h"

using namespace std;

int main() {
    Dog dog("狗子", "哈士奇");
    cout << "dog: " << dog.getName()
         << ", " << dog.getType() << endl;

    Animal animal = (Animal)dog;
    cout << "animal: " << animal.getName() << endl;
    return 0;
}
```

运行结果

```
dog: 狗子, 哈士奇
animal: 狗子
```

Chapter 4 多态

4.1 多态

4.1.1 多态（Polymorphism）

多态是同一个行为具有多个不同表现形式或形态的能力。例如可以把一只哈士奇，当成它的父类去看待，因此哈士奇是一只狗、一个动物或一个生物。

通过父类引用指向子类对象，例如 Animal animal = new Dog()，从而产生多种形态。父类引用仅能访问父类所声明的属性和方法，不能访问子类独有的属性和方法。

通过父类引用指向子类对象，从而产生多种形态。父类引用仅能访问父类所声明的属性和方法，不能访问子类独有的属性和方法。

在一对有继承关系的类中都有一个方法，其方法名、参数列表、返回值均相同，通过调用方法实现不同类对象完成不同的事件。

构成多态需要满足三个条件：

1. 必须存在继承关系。
2. 继承关系中必须有同名的虚函数。
3. 存在基类类型的指针或引用，通过该指针或引用调用虚函数。

4.2 虚函数

4.2.1 虚函数

虚函数是定义在基类中的函数，子类必须对其进行重写/覆盖（override），虚函数需要在类的成员函数前面加上 virtual 关键字。

重写/覆盖是指子类中存在重新定义的函数，其函数名、参数列表、返回值类型都与父类中被重写的函数一致。被重写的函数必须是虚函数。

子类若重写了父类的函数，那么子类将会隐藏其父类中被重写的函数。但是子类通过强制类型转换成父类后可以重新调用父类中被重写的函数。

虚函数

programmer.h

```
1  #ifndef _PROGRAMMER_H_
2  #define _PROGRAMMER_H_
3
4  #include <string>
5
6  class Programmer {
7  public:
8      Programmer(std::string title = "programmer");
9      virtual void work();
10
11 private:
12     std::string title;
13 };
14
15 #endif
```

programmer.cpp

```
1  #include "programmer.h"
2  #include <iostream>
3
```

```
4  using namespace std;
5
6  Programmer::Programmer(string title) : title(title) {}
7
8  void Programmer::work() {
9      cout << "programming" << endl;
10 }
```

java_programmer.h

```
1  #ifndef _JAVA_PROGRAMMER_H_
2  #define _JAVA_PROGRAMMER_H_
3
4  #include "programmer.h"
5  #include <string>
6
7  class JavaProgrammer : public Programmer {
8  public:
9      JavaProgrammer(std::string title = "Java Programmer");
10     virtual void work() override;
11 };
12
13 #endif
```

java_programmer.cpp

```
1  #include "java_programmer.h"
2  #include <iostream>
3
4  using namespace std;
5
6  JavaProgrammer::JavaProgrammer(string title)
7      : Programmer(title) {}
8
9  void JavaProgrammer::work() {
10     cout << "Android Development" << endl;
11 }
```

test_programmer.cpp

```cpp
#include <iostream>
#include "programmer.h"
#include "java_programmer.h"

using namespace std;

int main() {
    JavaProgrammer javaProgrammer;
    javaProgrammer.work();
    Programmer programmer = (Programmer)javaProgrammer;
    programmer.work();
    return 0;
}
```

运行结果

```
Android Development
programming
```

4.3 纯虚函数

4.3.1 纯虚函数

在虚函数后加上【= 0】后可以让这个函数变成纯虚函数，包含纯虚函数的类叫做抽象类（abstract class）或接口类（interface）。

抽象类不能被用于实例化对象，只是提供了所有的子类共有的部分。例如在动物园中，存在的都是动物具体的子类对象，并不存在动物对象，所以动物类不应该被独立创建成对象。

抽象类的作用是可以被子类继承，提供共性的属性和方法。父类提供的方法很难满足子类不同的需求，如果不定义该方法，则表示所有的子类都不具有该行为。如果定义该方法，所有的子类都在重写，那么这个方法在父类中是没有必要实现的，显得多余。

被 virtual 关键字修饰的方法称为纯虚函数。纯虚函数只有声明，没有实现。纯虚函数只能包含在抽象类中。产生继承关系后，子类必须重写父类中所有的纯虚函数，否则子类还是抽象类。

纯虚函数

shape.h

```
1 #ifndef _SHAPE_H_
2 #define _SHAPE_H_
3
4 class Shape {
5 public:
6     virtual double getArea() = 0;
7 };
8
9 #endif
```

rectangle.h

```
1 #ifndef _RECTANGLE_H_
```

```
2  #define _RECTANGLE_H_
3
4  #include "shape.h"
5
6  class Rectangle : public Shape {
7  public:
8      Rectangle(double length = 0, double width = 0);
9      virtual double getArea() override;
10
11 private:
12     double length;
13     double width;
14 };
15
16 #endif
```

rectangle.cpp

```
1  #include "rectangle.h"
2
3  Rectangle::Rectangle(double length, double width)
4      : length(length), width(width) {}
5
6  double Rectangle::getArea() {
7      return length * width;
8  }
```

circle.h

```
1  #ifndef _CIRCLE_H_
2  #define _CIRCLE_H_
3
4  #include "shape.h"
5
6  class Circle : public Shape {
7  public:
8      Circle(double raidus = 0);
9      virtual double getArea() override;
10
```

```
11 private:
12     double radius;
13 };
14
15 #endif
```

circle.cpp

```
1  #include "circle.h"
2
3  Circle::Circle(double radius) : radius(radius) {}
4
5  double Circle::getArea() {
6      return 3.14159 * radius * radius;
7  }
```

test_shape.cpp

```
1  #include <iostream>
2  #include "rectangle.h"
3  #include "circle.h"
4
5  using namespace std;
6
7  int main() {
8      Rectangle rectangle(7, 5);
9      Circle circle(6);
10     cout << "rectangle: " << rectangle.getArea() << endl;
11     cout << "circle: " << circle.getArea() << endl;
12     return 0;
13 }
```

运行结果

```
rectangle: 35
circle: 113.097
```

4.3.2 接口（Interface）

在面向对象中会使用抽象类为外部提供一个通用的、标准化的接口。

宏观上来讲，接口是一种标准。例如常见的 USB 接口，电脑通过 USB 接口连接各种外设设备，每一个接口不用关心连接的外设设备是什么，只要这个外设设备实现了 USB 的标准，就可以连接到电脑上。

从程序上来讲，接口代表了某种能力和约定。当父类的方法无法满足子类需求时，可实现接口扩充子类的能力，接口中方法的定义代表能力的具体要求。

使用接口可以进行对行为的约束和规则的制定，接口表示一组能力，那么一个类可以接受多种能力的约束。因此一个类可以实现多个接口，实现多个接口的时候，必须要把每一个接口中的方法都实现。

接口

language.h

```
#ifndef _LANGUAGE_H_
#define _LANGUAGE_H_

class Language {
public:
    virtual void translate() = 0;
};

#endif
```

english.h

```
#ifndef _ENGLISH_H_
#define _ENGLISH_H_

#include "language.h"
#include <string>

class English : public Language {
```

```
public:
    English(std::string content = "");
    virtual void translate() override;

private:
    std::string content;
};

#endif
```

english.cpp

```
#include "english.h"
#include <iostream>
#include <string>

using namespace std;

English::English(string content) : content(content) {}

void English::translate() {
    cout << content << endl;
}
```

chinese.h

```
#ifndef _CHINESE_H_
#define _CHINESE_H_

#include "language.h"
#include <string>

class Chinese : public Language {
public:
    Chinese(std::string content = "");
    virtual void translate() override;

private:
    std::string content;
```

```
};

#endif
```

chinese.cpp

```
#include "chinese.h"
#include <iostream>
#include <string>

using namespace std;

Chinese::Chinese(string content) : content(content) {}

void Chinese::translate() {
    cout << content << endl;
}
```

test_language.cpp

```
#include <iostream>
#include "language.h"
#include "english.h"
#include "chinese.h"

using namespace std;

void show(Language& language) {
    language.translate();
}

int main() {
    English english("Hello!");
    show(english);

    Chinese chinese("你好！ ");
    show(chinese);
    return 0;
}
```

运行结果

```
Hello!
你好
```

Chapter 5 异常

5.1 异常

5.1.1 异常（Exception）

异常就是程序在运行过程中出现的非正常的情况。异常本身是一个类，产生异常就是创建异常对象并抛出一个异常对象。Java 处理异常的方法是中断处理。

C++ 异常处理涉及到三个关键字：

1. throw：当问题出现时，程序会抛出一个异常。
2. try：放置可能抛出异常的代码。
3. catch：捕获并处理异常。

除以 0

```cpp
#include <iostream>

using namespace std;

```

```
5  int divide(int num1, int num2) {
6      if(num2 == 0) {
7          throw "division by zero";
8      }
9      return num1 / num2;
10 }
11
12 int main() {
13     try {
14         int result = divide(5, 0);
15         cout << result << endl;
16     } catch(const char *msg) {
17         cerr << msg << endl;
18     }
19
20     return 0;
21 }
```

运行结果

```
division by zero
```

普通的异常会导致程序无法完成编译，这样的异常被称为非运行时异常（non-runtime exception），但是由于异常是发生在编译时期的，因此常常称为编译时异常。在运行中如果遇到了异常，会导致程序执行的强制停止，这样的异常被称为运行时异常。

5.2 异常类

C++ 提供了一系列标准的异常，定义在 <exception> 中。

异常	描述
std::exception	所有标准 C++ 异常的父类
std::bad_alloc	通过 new 抛出
std::bad_cast	通过 dynamic_cast 抛出
std::bad_exception	处理无法预期的异常
std::bad_typeid	通过 typeid 抛出
std::logic_error	逻辑错误
std::domain_error	使用了无效的定义域
std::invalid_argument	使用了无效的参数
std::length_error	创建过长的 std::string
std::out_of_range	访问定义外的元素
std::runtime_error	运行时错误
std::overflow_error	发生上溢
std::underflow_error	发生下溢
std::range_error	存储超出范围的值

表 5.1: 异常类

what() 是异常类提供的一个公共方法，它已被所有子异常类重载。

bad_alloc

```
1 #include <iostream>
2 #include <exception>
3
4 using namespace std;
5
6 int main() {
7     try {
8         char *p = new char[0xffffffff];
9         delete p;
```

```
10      } catch(bad_alloc &e) {
11          cerr << e.what() << endl;
12      }
13      return 0;
14  }
```

运行结果

```
std::bad_alloc
```

5.3 自定义异常

5.3.1 自定义异常

系统中提供了很多的异常类型，但是异常类型提供地再多，也无法满足所有的需求。当需要的异常类型系统没有提供的时候，此时就需要自定义异常了。通过继承和重载 exception 类可以定义新的异常。

自定义异常

```
#include <iostream>
#include <string>
#include <exception>

using namespace std;

class AgeException : public exception {
public:
    AgeException(string msg) : msg(msg) {}

    virtual const char* what() const noexcept override {
        return msg.c_str();
    }

private:
    string msg;
};

int main() {
    try {
        int age;
        cout << "Enter age: ";
        cin >> age;
        if(age < 0 || age > 130) {
            throw AgeException("invalid age");
        }
```

```
27      } catch(AgeException& e) {
28          cout << e.what() << endl;
29      }
30
31      return 0;
32  }
```

运行结果

```
Enter age: -1
invalid age
```

Chapter 6 I/O 库

6.1 标准 I/O

6.1.1 标准 I/O

C++ 不直接处理输入输出，而是通过标准库中的一组类来处理 I/O。输入流 istream 提供输入，输出流 ostream 提供输出。

标准输入输出有以下特点：

1. cin 是 istream 的对象，从标准输入读取数据。
2. cout 是 ostream 的对象，向标准输出写数据。
3. cerr 是 ostream 的对象，用于输出错误信息，写到标准错误。
4. 【>>】运算符从 istream 对象读取输入。
5. 【<<】运算符从 ostream 对象写输出。
6. getline() 从给定的 ‘istream‘读取一行数据，存入 string 对象。

6.1.2 I/O 格式化

每个 iostream 对象维护一个格式状态来控制 I/O 的细节，如进制、精度、宽度等。操纵符改变流的格式状态时，通常改变后的状态对所有后续 I/O 都生效。

标准库定义了一组操纵符用来修改流的格式状态：

操纵符	功能
boolalpha	将 true 和 false 输出为字符串
noboolalpha	将 true 和 false 输出为 1 和 0
showbase	对整型值输出表示进制的前缀
noshowbase	不生成表示进制的前缀
showpoint	浮点数总是显示小数点
noshowpoint	只有浮点数包含小数部分才显示小数点
showpos	非负数显示【+】
noshowpos	非负数不显示【+】
uppercase	在十六进制中打印 0X，在科学计数法中打印 E
nouppercase	在十六进制中打印 0x，在科学计数法中打印 e
dec	整型显示为十进制
hex	整型显示为十六进制
oct	整型显示为八进制
left	在值的左侧添加填充字符
right	在值的右侧添加填充字符
internal	在符号和值之间添加填充字符
fixed	浮点数显示为定点十进制
scientific	浮点数显示为科学计数法
unitbuf	每次输出操作后刷新缓冲区
nounitbuf	恢复正常的缓冲区刷新方式
skipws	输入运算符跳过空白符
noskipws	输入运算符不跳过空白符
flush	刷新 ostream 缓冲区
ends	插入空字符，然后刷新 ostream 缓冲区
endl	插入换行符，然后刷新 ostream 缓冲区
setfill(ch)	用 ch 填充空白
setprecision(n)	将浮点精度设置为 n
setw(n)	读或写值的宽度为 n 个字符
setbase(n)	将整数输出为 n 进制

表 6.1: 操纵符

格式化输出

```cpp
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

int main() {
    cout << "布尔: ";
    cout << boolalpha << true << " " << false << endl;
    cout << "------------------------------" << endl;

    cout << "十进制: ";
    cout << dec << 20 << " " << 1024 << endl;

    cout << "十六进制: ";
    cout << showbase << hex
        << 20 << " "<< 1024
        << noshowbase << endl;

    cout << "八进制: ";
    cout << oct << 20 << " " << 1024 << dec << endl;
    cout << "------------------------------" << endl;

    cout << "科学计数法: ";
    cout << scientific
        << 100 * sqrt(2)
        << defaultfloat << endl;
    cout << "------------------------------" << endl;

    cout << "默认输出浮点数: ";
    cout << 10.0 << endl;

    cout << "浮点数打印小数点: ";
    cout << showpoint << 10.0 << noshowpoint << endl;
    cout << "------------------------------" << endl;
```

```
36
37      cout << "精度: ";
38      cout << setprecision(3) << fixed << sqrt(2) << endl;
39      cout << "------------------------------" << endl;
40
41      cout << "宽度填充: ";
42      cout << setfill('0') << setw(4) << 2021 << "/"
43           << setw(2) << 9 << "/"
44           << setw(2) << 2 << endl;
45      return 0;
46  }
```

运行结果

```
布尔: true false
------------------------------
十进制: 20 1024
十六进制: 0x14 0x400
八进制: 24 2000
------------------------------
科学计数法: 1.414214e+02
------------------------------
默认输出浮点数: 10
浮点数打印小数点: 10.0000
------------------------------
精度: 1.414
------------------------------
宽度填充: 2021/09/02
```

6.2 文件 I/O

6.2.1 文件 I/O

程序不仅要从控制台进行 I/O，还需要读写文件和字符串。

标准库的 I/O 类型在 3 个头文件中：

1. <iostream> 定义了读写流的基本类型。
2. <fstream> 定义了读写文件的类型。
3. <sstream> 定义了读写 string 对象的类型。

<fstream> 中定义了 3 个 I/O 类来读写文件：

1. ifstream 从给定文件读数据。
2. ofstream 向给定文件写数据。
3. fstream 可读写文件。

6.2.2 文件打开模式

每个流都有一个关联的文件模式，在打开文件时可以指定文件模式。

打开模式	作用
ios::in	以读方式打开
ios::out	以写方式打开
ios::app	以追加方式打开
ios::ate	打开文件定位到文件末尾
ios::trunc	如果文件存在，其内容将被截断，即把文件长度设为 0

表 6.2: 文件打开模式

文件 I/O

```cpp
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    string name;
    int id;
    cout << "Enter name: ";
    cin >> name;
    cout << "Enter id: ";
    cin >> id;

    ofstream out("info.txt");
    out << name << " " << id << endl;
    out.close();

    ifstream in("info.txt");
    in >> name >> id;
    in.close();

    cout << "name = " << name << ", id = " << id << endl;
    return 0;
}
```

运行结果

```
Enter name: Terry
Enter id: 979489
name = Terry, id = 979489
```

6.3 string 流

6.3.1 string 流

<sstream> 定义了 3 个类来支持内存 IO：

1. istringstream 从 string 读数据。
2. ostringstream 向 string 写数据。
3. stringstream 可读写 string。

string 流

```
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main() {
    string line;
    cout << "convert a string to Python list format: ";
    getline(cin, line);

    ostringstream out;
    istringstream in(line);
    string token;

    out << "[";
    while(in >> token) {
        out << token << ", ";
    }
    out << "\b\b]";
    cout << out.str() << endl;
    return 0;
}
```

运行结果

```
convert a string to Python list format: This is a test
[This, is, a, test]
```

Chapter 7 STL 标准模板库

7.1 模板

7.1.1 泛型编程（Generic Programming）

面向对象编程（OOP）和泛型编程（GP）都能处理在编写程序时类型未知的情况，OOP 能处理运行时获取类型的情况，GP 能处理编译期可获取类型的情况。

模板是泛型编程的基础，泛型编程就是以一种独立于任何特定类型的方式编写代码。C++ 标准库的容器、迭代器、算法都是泛型编程的例子。

7.1.2 函数模板

通过定义一个通用的函数模板可以处理参数为多种类型的情形，而不是为每个类型都定义一个重载。模板定义使用 template 关键字，后跟模板参数列表。模板参数表示函数或类定义中用到的类型，使用模板时需要隐式或显式提供模板实参，将其绑定到模板参数。

函数模板

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  template <typename T>
7  inline T getMax(const T& val1, const T& val2) {
8      return val1 > val2 ? val1 : val2;
9  }
10
```

```
11 int main() {
12     int iVal1 = 28;
13     int iVal2 = 92;
14     cout << getMax(iVal1, iVal2) << endl;
15 
16     double dVal1 = 3.14;
17     double dVal2 = 3.71;
18     cout << getMax(dVal1, dVal2) << endl;
19 
20     string sVal1 = "hello";
21     string sVal2 = "world";
22     cout << getMax(sVal1, sVal2) << endl;
23     return 0;
24 }
```

运行结果

```
92
3.71
world
```

函数模板仅仅是函数的规范，本身并不会占用内存。当编译器遇到对模板函数的调用时，才会在内存中创建该函数的实例。

7.1.3 类模板

类模板用来生成类的蓝图，与函数模板不同的是，类模板在实例化时编译器无法为类模板推导模板参数类型，而是必须在模板名后用【<>】提供实参。根据显式提供的模板实参列表，编译器使用这些模板参数来实例化特定的类。

编译器从类模板实例化一个类时，会重写模板，将模板参数的每个实例替换为给定的模板实参。因此类模板的每个实例都是独立的类，使用不同模板实参实例化出的类之间没有关联，也没有特殊的访问权限。

类模板

```cpp
#include <iostream>
#include <sstream>
#include <algorithm>

using namespace std;

template<class T>
class SortedArray {
public:
    SortedArray(int capacity = 1);
    SortedArray(T *arr, int capacity);
    ~SortedArray();

    string data();
    void add(T val);
    void remove(T val);

private:
    T *arr;
    int len;
    int capacity;
    void resize(int size);
};

template<class T>
SortedArray<T>::SortedArray(int capacity) {
    this->len = 0;
    this->capacity = capacity;
    this->arr = new T[capacity];
}

template<class T>
SortedArray<T>::SortedArray(T *arr, int len) {
    this->len = len;
    this->capacity = len;
```

```
36      this->arr = new T[len];
37      for(int i = 0; i < len; i++) {
38          this->arr[i] = arr[i];
39      }
40  }
41
42  template<class T>
43  SortedArray<T>::~SortedArray() {
44      delete arr;
45  }
46
47  template<class T>
48  string SortedArray<T>::data() {
49      if(len == 0) {
50          return "[]";
51      }
52
53      sort(this->arr, this->arr + len);
54      ostringstream out;
55      out << "[";
56      for(int i = 0; i < len; i++) {
57          out << arr[i] << ", ";
58      }
59      out << "\b\b]";
60      return out.str();
61  }
62
63  template<class T>
64  void SortedArray<T>::resize(int size) {
65      T *temp = new T[size];
66      for(int i = 0; i < len; i++) {
67          temp[i] = arr[i];
68      }
69      delete arr;
70      arr = temp;
71  }
72
```

```
73  template<class T>
74  void SortedArray<T>::add(T val) {
75      if(len == capacity) {
76          capacity *= 2;
77          resize(capacity);
78      }
79      arr[len++] = val;
80  }
81
82  template<class T>
83  void SortedArray<T>::remove(T val) {
84      for(int i = 0; i < len; i++) {
85          if(arr[i] == val) {
86              arr[i] = arr[len-1];
87              len--;
88              if(len <= capacity / 2) {
89                  capacity /= 2;
90                  resize(capacity);
91              }
92              break;
93          }
94      }
95  }
96
97  int main() {
98      int arr[] = {7, 7, 3, 9, 7, 1, 3};
99      int n = sizeof(arr) / sizeof(arr[0]);
100
101     SortedArray<int> sortedArray(arr, n);
102     cout << sortedArray.data() << endl;
103
104     sortedArray.add(28);
105     sortedArray.add(12);
106     cout << sortedArray.data() << endl;
107
108     sortedArray.remove(7);
109     sortedArray.remove(9);
```

```
110     cout << sortedArray.data() << endl;
111
112     return 0;
113 }
```

运行结果

```
[1, 3, 3, 7, 7, 7, 9]
[1, 3, 3, 7, 7, 7, 9, 12, 28]
[1, 3, 3, 7, 7, 12, 28]
```

7.2 容器

7.2.1 容器（Container）

容器是特定类型对象的集合，容器分为顺序容器和关联容器：

- 顺序容器：元素的顺序与其加入容器的位置对应。
- 关联容器：元素的顺序由其关联的关键字决定，关联容器分为有序关联容器和无序关联容器。

所有容器类共享公有接口，不同容器按不同方式扩展。

C++ 新标准容器的性能比旧版本快很多，其性能与最精心优化过的同类数据结构一样好。现代 C++ 程序应该使用标准库容器，而不是更原始的数据结构。

7.2.2 顺序容器

每个容器都定义于一个头文件中，文件名与容器名相同。容器都定义为模板类，顺序容器几乎可以保存任意类型的元素，还可以在容器中保存容器。

容器	描述
array	固定大小数组，支持快速随机访问，不能增删元素
vector	可变大小数组，支持快速随机访问，非尾部位置增删较慢
string	专门用于保存字符，随机访问快，在尾部增删速度快
deque	双端队列，支持快速随机访问，在头尾位置增删速度很快
list	双向链表，支持双向顺序访问，在任何位置增删都很快
forward_list	单向链表，只支持单向顺序访问，在任何位置增删都很快

表 7.1: 顺序容器

array 和内置数组一样大小固定，但操作更安全。除固定大小的 array 外，其它容器都提供高效灵活的内存管理，可以添加、删除、扩展和收缩容器的大小。

vector 和 string 将元素存储在连续空间中，故通过下标的随机访问很快。在尾部添加元素很快，但中间和头部插入或删除很慢。添加元素可能造成空间的重新分配和元素拷贝。

deque 支持快速随机访问，在两端插入或删除很快，但在中间插入或删除元素很慢。

list 和 forward_list 的设计目的是让任何位置的插入或删除都快速高效且不需重新分配内存，但是不支持随机访问，为访问一个元素需要遍历整个链表。

7.2.3 迭代器（Iterator）

迭代器比下标访问更通用，所有标准库容器都支持迭代器，但只有几种支持下标。迭代器提供了对容器对象的间接访问，类似于指针。begin() 返回指向首元素的迭代器，end() 返回指向尾元素下一位置（尾后）的迭代器。如果容器为空，则 begin() 和 end() 返回的都是尾后迭代器。任何可能改变容器容量的操作都会使容器的迭代器失效。

容器	**描述**
iterator	容器的迭代器
begin()	返回指向首元素的迭代器
end()	返回尾后迭代器
const_iterator	只读迭代器
cbegin()	返回指向首元素的只读迭代器
cend()	返回尾后只读迭代器
reverse_iterator	按逆序寻址元素的迭代器
rbegin()	返回指向尾元素的逆序迭代器
rend()	返回首前逆序迭代器
const_reverse_iterator	只读逆序迭代器
crbegin()	返回指向尾元素的只读逆序迭代器
crend()	返回首前只读逆序迭代器

表 7.2: 迭代器

迭代器可以进行算术运算，将迭代器与整数相加减可以得到向前或向后若干位置的迭代器。使用关系运算符【<】、【<=】、【>】、【>=】和【==】可以对迭代器所指位置比较大小。将两个迭代器相减，结果是两个迭代器的距离。

迭代器

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string s = "hello world";

    string::iterator iter = s.begin();
    cout << "[";
    while(iter != s.end()) {
        cout << *iter << ", ";
        iter++;
    }
    cout << "\b\b]" << endl;

    return 0;
}
```

运行结果

```
[h, e, l, l, o,  , w, o, r, l, d]
```

7.3 STL 数组

7.3.1 array

array 容器是 C++11 标准中新增的序列容器，它在普通数组的基础上，添加了一些成员函数和全局函数。在使用上，它比普通数组更安全，且效率并没有因此变差。和其它容器不同，array 的大小是固定的，无法动态的扩展或收缩。与内置数组不同的是，array 允许做整个容器的拷贝和赋值，要求两个 array 大小和元素类型都一样。

array 以类模板的形式定义在 <array> 头文件，array 具有固定大小，其大小也是类型的一部分，定义时模板参数包含元素类型和大小。

成员函数	功能
size()	返回容器中当前元素的数量
max_size()	返回容器可容纳元素的最大数量
empty()	判断容器是否为空
at(n)	返回容器中第 n 个元素的引用
front()	返回容器中第一个元素的直接引用
back()	返回容器中最后一个元素的直接应用
data()	返回一个指向容器首个元素的指针
fill(val)	将 val 赋值给容器中的每个元素
arr1.swap(arr2)	交换相同长度和类型的 arr1 和 arr2 中所有元素

表 7.3: array 成员函数

array

```
1 #include <iostream>
2 #include <array>
3
4 using namespace std;
5
6 int main() {
7     array<int, 10> arr = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
8       cout << "size = " << arr.size() << endl;
9       array<int, 10>::iterator begin = arr.begin();
10      array<int, 10>::iterator end = arr.end();
11      while(begin != end) {
12          cout << *begin << " ";
13          begin++;
14      }
15      cout << endl;
16      return 0;
17  }
```

运行结果

```
size = 10
0 1 2 3 4 5 6 7 8 9
```

7.3.2 vector

vector 表示对象的集合，由于 vector 容纳其它的对象，所以是一种容器。使用 vector 需要包含头文件 <vector>。vector 是一个类模板，模板可以看作编译器生成类或函数的一份说明。

初始化	功能
vector<T> v	创建一个空的 vector
vector<T> v2(v1)	用 v1 中所有元素的副本创建 v2
vector<T> v2 = v1	等价于 v2(v1)
vector<T> v(n, val)	v 中包含了 n 个值为 val 的元素
vector<T> v(n)	v 中包含了 n 个值为默认初始化的元素
vector<T> va, b, c, ...	用列表元素初始化 v
vector<T> v = a, b, c, ...	等价于 va, b, c, ...
vector<T> v (begin, end)	根据迭代器范围 [begin, end) 复制到 vector 中

表 7.4: vecor 初始化

vector 初始化

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

template <typename T>
ostream& operator<<(ostream& out, const vector<T>& v) {
    if(!v.empty()) {
        out << "[";
        copy(v.begin(), v.end(), ostream_iterator<T>(out, ", "));
        out << "\b\b]";
    }
    return out;
}

int main() {
    vector<int> v1(10);          // 有10个元素，都是0
    vector<int> v2{10};          // 有1个元素，值是10
    vector<int> v3(10, 1);       // 有10个元素，都是1
    vector<int> v4{10, 1};       // 有2个元素，10和1
    vector<string> v5{"hello"}; // 有1个元素，是字符串"hello"

    cout << "v1 = " << v1 << endl;
    cout << "v2 = " << v2 << endl;
    cout << "v3 = " << v3 << endl;
    cout << "v4 = " << v4 << endl;
    cout << "v5 = " << v5 << endl;
    return 0;
}
```

运行结果

```
v1 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
v2 = [10]
v3 = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
v4 = [10, 1]
v5 = [hello]
```

7.3.3 vector 操作

操作	功能
v.empty()	判断 vector 是否为空
v.size()	返回 vector 元素个数
v[n]	返回 vector 中第 n 个元素的引用
v1 = v2	用 v2 中的元素拷贝替换 v1 中的元素
v1 == v2、v1 != v2	v1 和 v2 相等当且仅当元素个数和对应元素都相同
v.push_back(val)	向 vector 尾部添加一个元素
v.insert(iter, val)	向迭代器指向元素前添加一个元素
v.pop_back()	删除 vector 最后一个元素
v.erase(iter)	删除迭代器指向元素
v.erase(begin, end)	删除迭代器返回 [begin, end) 范围元素
v.clear()	清空 vector
v.swap(vector)	交换两个同类型 vector 数据
v.assign(n, val)	设置 vector 中前 n 个元素值为 val

表 7.5: vector 操作

vector 不能使用下标添加元素，否则会造成缓冲区溢出，确保下标合法的一种有效手段就是尽可能使用 for-each 循环。如果循环体内部包含向 vector 添加元素的语句，则不能使用 for-each 循环。

vector

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> v;
    for(int i = 0; i < 10; i++) {
        v.push_back(i * i);
    }

    for(int& item : v) {
        cout << item << " ";
    }
    cout << endl;
    return 0;
}
```

运行结果

```
0 1 4 9 16 25 36 49 64 81
```

7.4 STL 字符串

7.4.1 string

string 是标准库中的类型，表示可变长字符序列，使用需要包含头文件 <string>。

string 的初始化分为：

1. 直接初始化：使用括号初始化，调用构造函数。
2. 拷贝初始化：使用赋值初始化，调用重载的赋值运算符。

string 初始化

```cpp
#include <iostream>
#include <string>

using namespace std;

int main() {
    string s1;              // 默认初始化，为空字符串
    string s2(s1);          // 直接初始化，s2是s1的副本
    string s3 = s1;         // 拷贝初始化，s3是s1的副本，等价s3(s1)
    string s4("hello");     // 直接初始化，初始化为字面值常量
    string s5 = "hello";    // 拷贝初始化，初始化为字面值常量
    string s6(10, 'x');     // 直接初始化，初始化为10个字符'x'

    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;
    cout << "s3 = " << s3 << endl;
    cout << "s4 = " << s4 << endl;
    cout << "s5 = " << s4 << endl;
    cout << "s6 = " << s4 << endl;

    return 0;
}
```

运行结果

```
s1 =
s2 =
s3 =
s4 = hello
s5 = hello
s6 = hello
```

7.4.2 string 操作

操作	功能
out « s	将 s 写到输出流 out 中
in » s	从输入流中读取字符串赋给 s，字符串以空白符分割
getline(in, s)	从输入流中读取一行赋给 s
s.empty()	判断 s 是否为空
s.size()	返回 s 中字符个数
s[n]	返回 s 中第 n 个字符的引用
s1 + s2	返回 s1 和 s2 连接后的结果
s1 = s2	用 s2 的副本替换 s1
s1 == s2、s1 != s2	判断 s1 和 s2 是否相等
<、<=、>、>=	字典序比较，对大小写敏感
s1.append(s2)	尾部插入
s1.insert(pos, s2)	在第 pos 个位置插入 s2
s.erase(pos, n)	从第 pos 个位置删除 n 个字符
s1.replace(pos, n, s2)	从第 pos 个位置开始替换 n 个字符为 s2
s.substr(pos, n)	返回一个从 pos 开始的 n 个字符的拷贝
s1.find(s2)	查找 s1 中 s2 第一次出现的位置
s1.rfind(s2)	查找 s1 中 s2 最后一次出现的位置

表 7.6: string 操作

string

```cpp
#include <iostream>
#include <string>

using namespace std;

int main() {
    string s("Hello");

    s.append("world");          // Helloworld
    s.insert(s.size(), "!");    // Helloworld!

    s.replace(1, 4, "i");       // Hiworld!
    s.erase(2, 5);              // Hi!
    s.insert(2, " C++");        // Hi C++!

    cout << s << endl;

    cout << s.substr(3, 3) << endl;
    cout << s.substr(3) << endl;
    cout << s.find("C++") << endl;

    return 0;
}
```

运行结果

```
Hi C++!
C++
C++!
3
```

7.5 STL 链表

7.5.1 list

list 双向链表通过指针连成逻辑意义上的线性表，由于 list 中结点并不要求在一段连续内存中，因此 list 不支持快速随机存取，迭代器只能通过【++】或【--】移动。

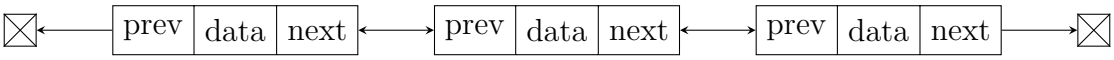

图 7.1: 双向链表

操作	功能
list<T> lst	创建空的 list
list<T> lst(n)	创建含有 n 个元素的 list
list<T> lst1(lst2)	使用 lst2 初始化 lst1
lst.size()	返回 list 元素个数
lst.clear()	删除所有元素
lst.empty()	判断 list 是否为空
lst.front()	返回第一个元素
lst.back()	返回最后一个元素
lst.insert()	插入一个元素
lst.erase()	删除一个元素
lst.push_front()	在头部添加一个元素
lst.push_back()	在尾部添加一个元素
lst.pop_front()	删除第一个元素
lst.pop_back()	删除最后一个元素
lst.remove()	删除元素
lst.reverse()	反转 list
lst.sort()	排序
lst.unique()	去除相邻的重复元素
lst.merge()	合并两个有序 list

表 7.7: list 操作

其中，lst.unique() 并不是把重复的元素删除，而是全部放到数组尾部，返回去重后的尾地址。unique() 中不自带 sort()，因此需要先使用 sort() 进行排序。

list

```cpp
#include <iostream>
#include <list>

using namespace std;

void printList(list<int> lst) {
    for(list<int>::iterator iter = lst.begin();
                            iter != lst.end();
                            iter++) {
        cout << *iter << " ";
    }
    cout << endl;
}

int main() {
    list<int> lst;

    lst.push_back(11);       // [11]
    lst.push_front(22);      // [22, 11]
    cout << lst.front() << endl;    // 22
    cout << lst.back() << endl;     // 11

    lst.insert(++lst.begin(), 3);   // [22, 3, 11]
    lst.insert(--lst.end(), 2);     // [22, 3, 2, 11]
    lst.push_back(2);               // [22, 3, 2, 11, 2]
    printList(lst);

    lst.pop_front();                // [3, 2, 11, 2]
    lst.sort();                     // [2, 2, 3, 11]
    lst.unique();                   // [2, 3, 11]
    printList(lst);
```

```
32
33      lst.sort();                         // [2, 3, 11]
34      printList(lst);
35
36      list<int> lst2{1, 2, 8};
37      lst.merge(lst2);                    // [1, 2, 2, 3, 8, 11]
38      printList(lst);
39
40      return 0;
41  }
```

运行结果

```
22
11
22 3 2 11 2
2 3 11
2 3 11
1 2 2 3 8 11
```

7.5.2 forward_list

forward_list 和 list 的区别在于前者是单向链表，每个元素内部只有一个链接指向下一个元素，因此在存储方面 list 会消耗更多的空间。

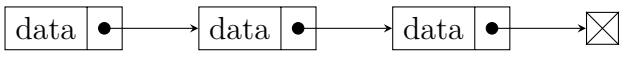

图 7.2: 单向链表

forward_list 不支持反向迭代器，并且没有指向尾元素的迭代器，因此不提供 back()、push_back()、pop_back() 等操作。

7.6 容器适配器

7.6.1 stack

栈，又名堆栈，是一种运算受限的线性数据结构，栈只能在表尾进行插入和删除操作。

栈中的元素只能先进后出（FILO, First In Last Out）。最早进入栈的元素所存放的位置叫作栈底（bottom），最后进入栈的元素存放的位置叫作栈顶（top）。

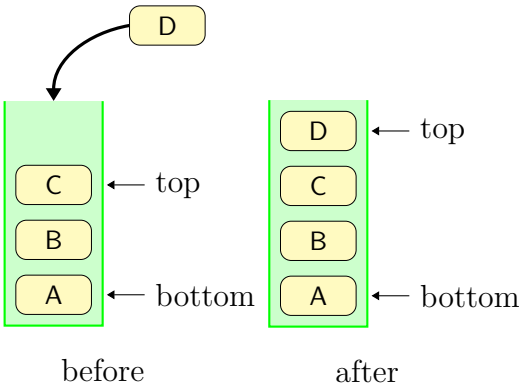

图 7.3: 栈

向一个栈插入新元素的操作称为入栈 push（或进栈、压栈），从一个栈删除元素的操作称为出栈 pop（或退栈、弹栈）。入栈操作就是把新元素放入栈中，只允许从栈顶一侧放入元素，新元素的位置将会成为新的栈顶。出栈操作就是把新元素从栈中弹出，只有栈顶元素才允许出栈，出栈元素的前一个元素将会成为新的栈顶。

操作	**功能**
empty()	判断栈是否为空
size()	返回栈中元素个数
push()	入栈，调用底层容器的 push_back() 实现
pop()	出栈
top()	返回栈顶元素的引用

表 7.8: stack 操作

stack

```cpp
#include <iostream>
#include <stack>

using namespace std;

int main() {
    stack<int> s;
    s.push(1);
    s.push(2);
    s.push(3);
    cout << s.top() << endl;

    while(!s.empty()) {
        cout << s.top() << endl;
        s.pop();
    }
    return 0;
}
```

运行结果

```
3
3
2
1
```

7.6.2 deque

deque 双端队列是一种同时具有队列和栈的性质的数据结构，双端队列可以从其两端插入和删除元素。

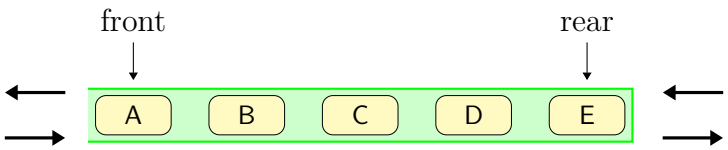

图 7.4: 双端队列

操作	功能
empty()	判断 deque 是否为空
size()	返回 deque 中元素个数
front()	返回首元素引用
back()	返回尾元素引用
push_front()	在头部添加一个元素
push_back()	在尾部添加一个元素
pop_front()	在头部删除一个元素
pop_back()	在尾部删除一个元素
clear()	清空 deque

表 7.9: deque 操作

deque

```
#include <iostream>
#include <deque>

using namespace std;

int main() {
    deque<int> deq;

    deq.push_front(1);
    deq.push_front(2);
    deq.push_back(3);
    deq.push_back(4);
    cout << deq.front() << endl;
    cout << deq.back() << endl;
```

```
15
16      deq.pop_back();
17      deq.pop_front();
18      cout << deq.front() << endl;
19      cout << deq.back() << endl;
20
21      return 0;
22  }
```

运行结果

```
2
4
1
3
```

7.6.3 priority_queue

普通的队列是一种先进先出（FIFO, First In First Out）的数据结构，元素在队尾添加，在队头删除。

在优先队列 priority_queue 中，元素被赋予优先级，当访问元素时，具有最高优先级的元素最先被访问。使用 priority_queue 需要包含头文件 <queue>。

操作	功能
empty()	判断队列是否为空
size()	返回队列中元素个数
top()	访问队头
push()	插入元素
pop()	弹出队头

表 7.10: priority_queue 操作

priority_deque

```cpp
#include <iostream>
#include <queue>

using namespace std;

int main() {
    priority_queue<int> pq;
    pq.push(9);
    pq.push(2);
    pq.push(8);

    while(!pq.empty()) {
        cout << pq.top() << endl;
        pq.pop();
    }
    return 0;
}
```

运行结果

```
9
8
2
```

7.7 关联容器

7.7.1 关联容器

顺序容器的元素是按照在容器中的位置来保存和访问的，关联容器的元素按照关键字来保存和访问。关联容器支持高效地关键字查询和访问。所有关联容器都支持通用容器操作，但不支持顺序容器特有的操作，例如 push_front() 或 push_back()。

set 和 map 是两种关联容器，set 中的元素只包含关键字，而 map 中的元素是键值对（key-value pair）。

关联容器	描述	头文件
set	只保存关键字的容器	<set>
multiset	关键字可以重复出现的 set	<set>
unordered_set	用哈希函数组织的 set	<unordered_set>
unordered_multiset	哈希组织的 set，关键字可重复	<unordered_set>
map	保存键值对的容器	<map>
multimap	关键字可重复出现的 map	<map>
unordered_map	用哈希函数组织的 map	<unordered_map>
unordered_multimap	哈希组织的 map，关键字可重复	<unordered_map>

表 7.11: 关联容器

set 是关键字的集合，其底层实现使用的是红黑树，当想要查找一个值是否存在时可以使用。set 是模板，使用时必须在模板参数中指定元素类型。

map 是模板，使用时必须在模板参数中指定 key 和 value 的类型。map 常称为关联数组或字典，但是其下标不必是整数，而是通过关键字来查找值。

map 的元素都是 pair 类型，pair 也是模板，定义在 <utility> 中，一个 pair 保存两个 public 的数据成员，分别叫 first 和 second。

关键词提取

summary.txt

```
Internet of Things allows billions of physical objects to
be connected to collect and exchange data for offering various
applications, such as environmental monitoring, infrastructure
management, and home automation. On the other hand, IoT has
unsupported features that are critical for some IoT applications,
including smart traffic lights, home energy management and
augmented reality. To support these features, fog computing is
integrated into IoT to extend computing, storage and networking
resources to the network edge. Unfortunately, it is confronted
with various security and privacy risks, which raise serious
concerns towards users.
```

excludes.txt

```
the a an is this
that of at in on for
and it with to we I
into which these those are
be as has have or
```

STL_set_map.cpp

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <set>
#include <map>
#include <cctype>
#include <algorithm>

using namespace std;

string getSummary(string filename) {
    ifstream in(filename);
    string summary;
    string line;
```

```
    while (getline(in, line)) {
        summary += line;
    }
    in.close();
    return summary;
}

set<string> getExcludes(string filename) {
    ifstream in(filename);
    set<string> excludes;
    string token;
    while (in >> token) {
        excludes.insert(token);
    }
    in.close();
    return excludes;
}

int main() {
    string summary = getSummary("summary.txt");
    vector<string> tokens;

    istringstream in(summary);
    string token;
    while (in >> token) {
        // eliminate the trailing punctuation
        if (!isalpha(token.back())) {
            token.pop_back();
        }

        // convert to lower case
        transform(
            token.begin(), token.end(),
            token.begin(), ::tolower);

        tokens.push_back(token);
    }
```

```
54
55      set<string> excludes = getExcludes("excludes.txt");
56      map<string, int> keywords;
57
58      for (string token : tokens) {
59          // not in excludes set
60          if (excludes.find(token) == excludes.end()) {
61              keywords[token]++;
62          }
63      }
64
65      for (auto& p : keywords) {
66          // print keywords that appear more than once
67          if (p.second > 1) {
68              cout << p.first << ": " << p.second << endl;
69          }
70      }
71
72      return 0;
73  }
```

运行结果

```
applications: 2
computing: 2
features: 2
home: 2
iot: 3
management: 2
various: 2
```

7.8 泛型算法

7.8.1 泛型算法（Generic Algorithm）

标准库未给容器添加大量功能，而是提供一组独立于容器的泛型算法。它们实现了一些经典算法的公共接口，可用于不同类型的容器和元素。标准库算法不直接操作容器，而是遍历两个迭代器指定的元素范围。指针类似于内置数组上的迭代器，故泛型算法也可操作内置数组和指针。

大多数算法定义在 <algorithm> 中，另外一组数值算法定义在 <numeric> 中。

大多标准库算法都对一个范围内的元素操作，这个范围称为输入范围，接受输入范围的算法总是用前两个参数来表示输入范围。多数算法遍历输入范围的方式相似，但使用元素的方法不同，如是否读、是否写、是否重排等。

7.8.2 只读算法

只读算法只读取输入范围的元素，但不改变它们。使用只读算法，最好用 cbegin() 和 cend()。

find() 的作用是将范围中每一个元素与给定值比较，返回第一个等于给定值的元素的迭代器，如果没有匹配则返回该范围的尾后迭代器。find() 会调用给定值类型的【==】运算符来比较。

```
1 template <class InputIterator, class T>
2 InputIterator find(
3     InputIterator first, InputIterator last, const T& val
4 );
```

count() 的作用是将范围中每一个元素与给定值比较，返回给定值在范围中出现的次数。

accumulate() 定义于 <numeric>，其作用是对范围中元素求和，再加上给定值，返回求值结果。accumulate() 会调用给定值类型的【+】运算符来求和。

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);
```

equal() 的作用是确定两个序列的值是否相同，若果所有元素都相等时返回 true，否则 false。函数接受有 3 个参数，前两个是第一个序列的输入范围，第三个参数是第二个范围的首迭代器。equal() 会调用【==】运算符来比较，元素类型不必严格一致。

```
template <class InputIterator1, class InputIterator2>
bool equal(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2
);
```

只读算法

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

int main() {
    vector<int> v1 = {40, 9, 20, 34, 7, 34, 85, 9};
    vector<int> v2 = {40, 9, 20, 34, 7, 34, 85, 9};
    int key = 34;

    vector<int>::iterator iter = find(
        v1.begin(), v1.end(), key
    );
    if(iter != v1.end()) {
        cout << "key found in vector: " << *iter << endl;
    } else {
        cout << "key not found" << endl;
```

```
20      }
21
22      cout << key << " appears "
23           << count(v1.begin(), v1.end(), key) << endl;
24      cout << "sum = "
25           << accumulate(v1.begin(), v1.end(), 0) << endl;
26      cout << "v1 == v2? " << boolalpha
27           << equal(v1.begin(), v1.end(), v2.begin()) << endl;
28
29      return 0;
30  }
```

运行结果

```
key found in vector: 34
34 appears 2
sum = 238
v1 == v2? true
```

7.8.3 写容器元素算法

写容器元素算法可对序列中元素重新赋值，要求原序列大小不小于要写入的元素数目。

fill() 的作用是用给定值填满输入范围，函数接受 3 个参数，前 2 个是输入范围，第 3 个是给定值。

```
1  template <class ForwardIterator, class T>
2  void fill(
3      ForwardIterator first, ForwardIterator last, const T& val
4  );
```

copy() 的作用是将输入范围的值拷贝到目标序列，返回目标序列的尾后迭代器。函数接受 3 个参数，前 2 个是输入范围，第 3 个是目标序列的起始位置。

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(
    InputIterator first, InputIterator last,
    OutputIterator result
);
```

replace() 的作用是将序列中所有等于给定值的元素换为另一个值，函数接受 4 个参数，前 2 个是输入范围，后 2 个分别是要搜索的值和新值。

```
template <class ForwardIterator, class T>
void replace(
    ForwardIterator first, ForwardIterator last,
    const T& old_value, const T& new_value
);
```

写容器元素算法

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

ostream& operator<<(ostream& out, vector<int>& v) {
    for(vector<int>::iterator iter = v.begin();
    iter != v.end();
    iter++
    ) {
        out << *iter << " ";
    }
    return out;
}

int main() {
    vector<int> v1 = {9, 2, 8, 8, 2, 1, 0, 1, 2};
```

```
19      vector<int> v2(10);
20
21      fill(v2.begin(), v2.end(), 0);
22      cout << v2 << endl;
23
24      copy(v1.begin(), v1.end(), v2.begin());
25      cout << v2 << endl;
26
27      replace(v2.begin(), v2.end(), 1, 7);
28      cout << v2 << endl;
29      return 0;
30  }
```

运行结果

```
0 0 0 0 0 0 0 0 0 0
9 2 8 8 2 1 0 1 2 0
9 2 8 8 2 7 0 7 2 0
```

7.8.4 重排容器元素算法

重排容器元素算法可对容器中元素重新排列顺序。

sort() 的作用是重排输入序列的元素使其有序，函数接受 2 个参数表示输入范围，函数会调用序列元素类型的【<】运算符。

标准库允许在执行算法时用自定义操作代替默认算符，而不需要在类型中重载。

```
1  template <class RandomAccessIterator>
2  void sort(
3      RandomAccessIterator first, RandomAccessIterator last
4  );
5
6  template <class RandomAccessIterator, class Compare>
```

```
void sort(
    RandomAccessIterator first, RandomAccessIterator last,
    Compare comp
);
```

谓词（predicate）是一个可调用的表达式，其返回值（true / false）可用作条件。按照参数的数量分为一元谓词和二元谓词。接受谓词的算法用该谓词代替默认的算符来操作元素。

unique() 的作用是重排输入序列，消除相邻重复项，返回消除后的无相邻重复值的尾后迭代器。unique() 不真正删除元素，只是将后面的不重复值前移来覆盖前面的重复值，真正删除元素需要使用容器操作。

```
template <class ForwardIterator>
ForwardIterator unique(
    ForwardIterator first, ForwardIterator last
);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(
    ForwardIterator first, ForwardIterator last,
    BinaryPredicate pred
);
```

重排容器元素算法

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

ostream& operator<<(ostream& out, vector<string>& v) {
    for(vector<string>::iterator iter = v.begin();
                                 iter != v.end();
```

```
                                iter++) {
        out << *iter << " ";
    }
    return out;
}

bool isLonger(const string& s1, const string& s2) {
    return s1.size() > s2.size();
}

int main() {
    vector<string> v = {
        "C++", "Java", "Python",
        "C++", "C", "JavaScript",
        "Golang", "C++"
    };

    sort(v.begin(), v.end());
    cout << v << endl;

    sort(v.begin(), v.end(), isLonger);
    cout << v << endl;

    vector<string>::iterator endUnique = unique(
    v.begin(), v.end()
);
    v.erase(endUnique, v.end());
    cout << v << endl;
    return 0;
}
```

运行结果

```
C C++ C++ C++ Golang Java JavaScript Python
JavaScript Golang Python Java C++ C++ C++ C
JavaScript Golang Python Java C++ C
```