

CPP

Do or Not Do ,There Is No Try!

Edited by yyllll

术语

arguments---实参 left brace\right brace---左右大括号 identifiers---标识符
parameters---形参 access specifiers---访问说明符 encapsulated---封装
member-initializer list---构造函数成员初始值列表 scope---域 statement---
语句 pseudocode---伪代码 solid circle---实心圆 dotted line---虚线
indented---缩进 Incrementing---递增 Truncation---截断 round---四舍五入
arithmetic overflow---数值溢出 operand---操作数 quotient---商 explicit
conversion---显式转换 implicit conversion---隐式转换 promotion---提升
unary operator---一元运算符 whole number---非负整数 Polymorphism---
多态 evaluate---计算 Divide-and-Conquer---分治法 vector---向量

1. C++编程的介绍

1.1 第一个C++程序

```
1 // Fig. 2.1
2 // Text-printing program.
3 #include <iostream> // enables program to output data to
4 // the screen
5 // function main begins program execution
6 int main() {
7     std::cout << "Welcome to C++!\n"; // display message
8     return 0; // indicate that program ended successfully
9 }
```

每个程序都应该以注释开头用以解释程序的目的

第三行是预处理指令，是在程序被编译之前给预处理器的信息。这一行通知预处理器在程序中包括输入/输出流头文件<iostream>。此标头是一个包含信息的文件，编译器在编译任何将数据输出到屏幕或从键盘输入数据时使用使用C++的输入/输出流。关于<iostream>的更多细节会在后面的章节讨论。

```
1 | int main() {
```

main后面的括号表示main是一个称为函数的程序构建块。每个程序都必须有一个main函数。函数名前面的关键词指明函数的返回类型，函数名后面的括号，是书写形参之处，形参列表可以为空，大括号里的部分称为函数体。

```
1 | std::cout << "Welcome to C++!\n";
```

在编程语言中，命名空间是一种特殊的作用域，它包含了处于该作用域中的所有标示符，而且其本身也是由标示符表示的。命名空间的使用目的是为了将逻辑相关的标示符限定在一起，组成相应的命名空间，可使整个系统更加模块化，最重要的是它可以防止命名冲突。就好比在两个函数或类中定义相同名字的对象一样，利用作用域标示符限定该对象是哪个类里定义的。

使用using std::cout 事先声明： `cout<<"Hello!"<<std::endl;`//分别引入，需要用哪个引用哪个，保证程序中名称的唯一性

使用using namespace std声明： `cout<<"Hello!"<<endl;`//**引入名字空间的所有内容，不推荐这样写**

在输出语句的上下文中，<<运算符被称为流插入操作符，操作人员执行此程序时，运算符右侧的值，即右侧操作数，将插入输出流中。请注意，<<运算符指向数据所在的位置。字符串文字面量通常按照它们出现在双引号之间的方式打印。但是，这些字符（\n）没有打印在屏幕上。反斜杠（\）被称为转义符。它表示要输出一个“特殊”字符。常见的转义字符如下图：

Escape sequence	Description
\n	Newline. Position the screen cursor to the beginning of the next line.
\t	Horizontal tab. Move the screen cursor to the next tab stop.
\r	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
\a	Alert. Sound the system bell.
\\\	Backslash. Used to print a backslash character.
\'	Single quote. Used to print a single-quote character.
\"	Double quote. Used to print a double-quote character.

const对象一旦创建后其值就不能改变，所以const对象必须初始化。

1.2 初始化问题

可以使用大括号初始化，也可以使用等号和小括号初始化。

- 初始化的概念：创建变量时赋予它一个值（不同于赋值的概念）
- 类的构造函数控制其对象的初始化过程，无论何时只要类的对象被创建就会执行构造函数
- 如果对象未被用户指定初始值，那么这些变量会被执行默认初始化，默认值取决于变量类型和定义变量的位置
- 无论何时只要类的对象被创建就会执行构造函数，通过显式调用构造函数进行初始化被称为显式初始化，否则叫做隐式初始化
- 使用等号（=）初始化一个类变量执行的是拷贝初始化，编译器会把等号右侧的初始值拷贝到新创建的对象中去，不使用等号则执行的是直接初始化
- 传统C++中列表初始化仅能用于普通数组和POD类型，C++11新标准将列表初始化应用于所有对象的初始化（但是内置类型习惯于用等号初始化，类类型习惯用构造函数圆括号显式初始化，vector、map和set等容器类习惯用列表初始化）

初始化不等于赋值

初始化的含义是创建变量时赋予其一个初始值，而赋值的含义是把对象的当前值擦去，并用一个新值替代它。C++定义了初始化的好几种不同形式，例如我们定义一个int变量并初始化为0，有如下4种方式：

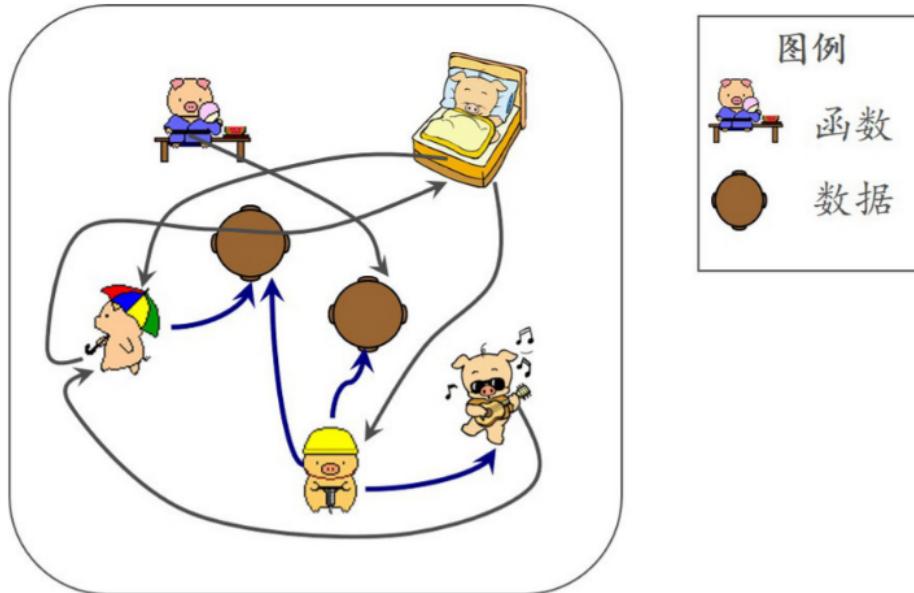
```
1 int i = 0;
2 int i = {0}; //C++11
3 int i{0}; //C++11
4 int i(0);
```

变量名不能和关键字重复。有效的标识符包括一系列字符（字母、数字和下划线）组成，并且不以数字开头。在C++中，区分字母的大小写，因此，A1和a1不是一回事。

变量的声明必须在变量使用的位置之前。

1.3 C++编程范式(C++ programming paradigm)

1. Structural Programming (结构化编程)

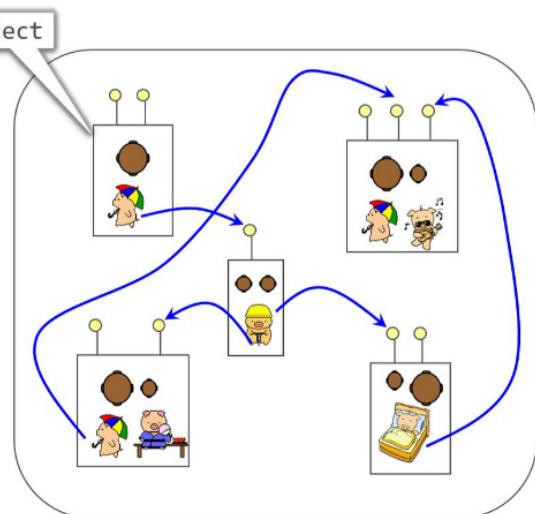


2. Object-Oriented Programming (OOP, 面向对象编程)

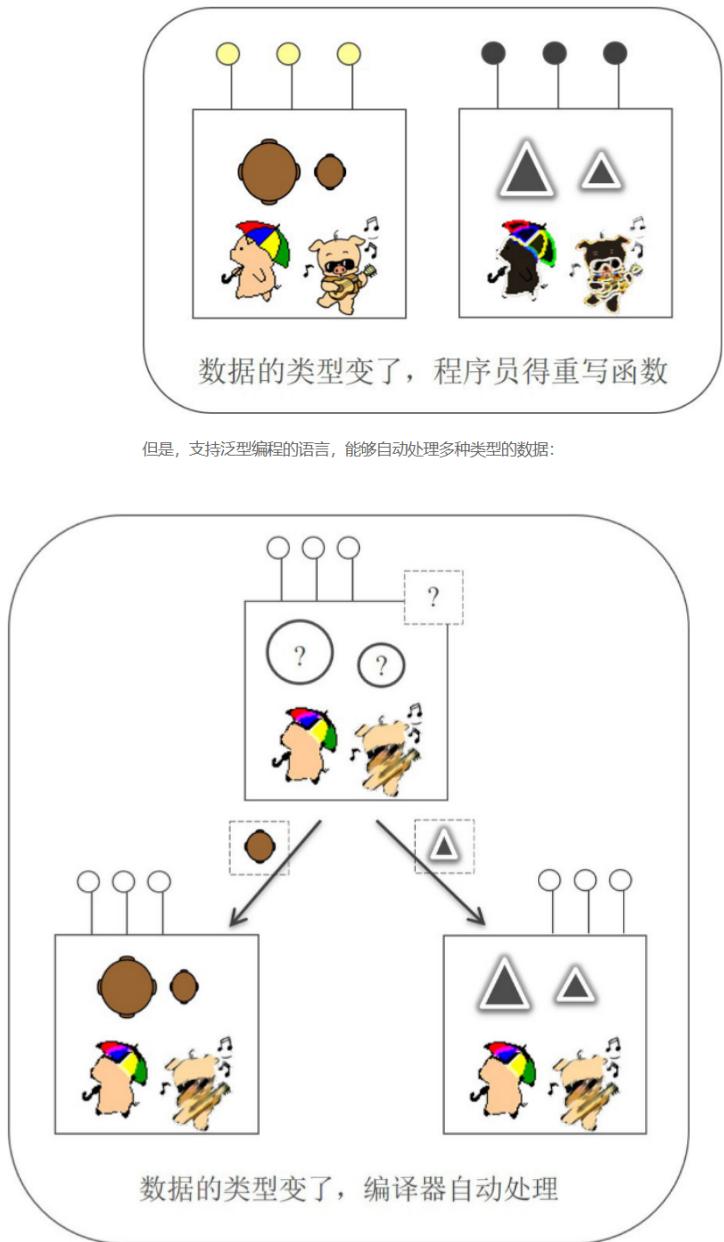
Advantages of OO

OOP can make programs easy to develop and easy to maintain.

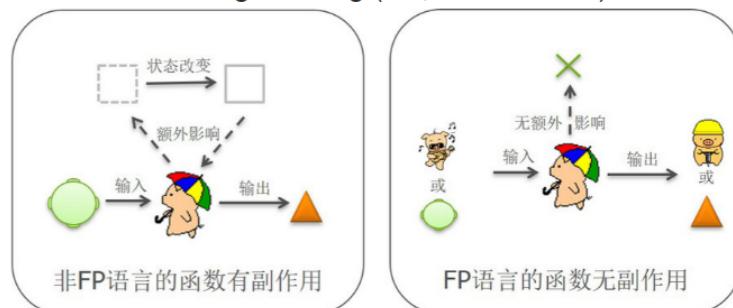
面向对象编程使得开发和维护程序变得更容易。



3. Generic Programming (GP, 泛型编程)



4. Functional Programming (FP, 函数式编程)



在计算机科学中，一个函数的副作用就是该函数除了返回计算结果之外，还对其他部分(如外部变量、数据结构、系统状态等)进行了修改或产生了其他影响。如果一个函数没有副作用，那么给定相同的输入，它总会产生相同的输出，而且不会改变系统的状态。例如下列的加法函数

```
1 def add(x,y):  
2     return x+y  
3 //没有副作用。无论调用多少次，只要输入相同，总是会返回相同的结果，而  
且不会改变系统的状态。
```

```
1 counter=0;  
2 def increment_counter(x):  
3     global counter  
4     counter+=x  
5     return counter  
6 //有副作用，改变了全局变量counter的值
```

函数式编程鼓励使用无副作用的函数，因为这样的函数更容易理解和调试，也更容易进行并行和分布式计算。使得代码更加模块化。

1.4 命名空间/名字空间(name space)

[详解c++的命名空间namespace - 知乎 \(zhihu.com\)](#)

[命名空间\(C++\) | Microsoft Learn](#)

在c++中，名称（name）可以是符号常量、变量、函数、结构、枚举、类和对象等等。工程越大，名称互相冲突性的可能性越大。另外使用多个厂商的类库时，也可能导致名称冲突。为了避免，在大规模程序的设计中，以及在程序员使用各种各样的C++库时，这些标识符的命名发生冲突，标准C++引入关键字 namespace（命名空间/名字空间/名称空间），可以更好地控制标识符的作用域。

命名空间是一个声明性区域，为其内部的标识符（类型、函数和变量等的名称）提供一个范围。命名空间用于将代码组织到逻辑组中，还可用于避免名称冲突，尤其是在基本代码包括多个库时。命名空间范围内的所有标识符彼此可见，而没有任何限制。命名空间之外的标识符可通过使用每个标识符的完全限定名（例如 `std::vector<std::string> vec;`）来访问成员，也可通过单个标识符的 using 声明 (`using std::string`) 或命名空间中所有标识符的 `using namespace std;` 来访问成员。头文件中的代码应始终使用完全限定的命名空间名称。

不要使用如下的命名空间写法：

```
1 using namespace std;
```

可以使用如下形式的写法：

```
1 | using std::cout;
2 | using std::endl;
```

1.5 声明与定义

声明是引入标识符并描述其类型，无论是什么类型，对象还是函数。编译器需要该声明，以便识别在别处使用该标识符。

```
1 | extern int bar;
2 | extern int g(int,int);
3 | double f(int,double);
4 | class foo;
```

定义是实例化这个标识符。链接器需要定义，以便将对标识符的引用链接到标识符所表示的实体。

```
1 | int bar;
2 | int g(int lhs,int rhs){
3 |     return lhs*rhs
4 | };
5 | class foo{
6 |     //do something
7 | };
```

两者的区别：

1. 定义有时可以取代声明，反之不可以
2. 标识符可以被声明多次，但只能定义一次
3. 定义通常伴随着编译器为标识符分配内存

2. 类、对象、成员函数的介绍

2.1 类的基本概念

在上一节中，讨论了类，对象，数据成员(属性)，成员函数(行为)。有日期对象、时间对象、音频对象、视频对象、汽车对象、人对象等。几乎任何名词都可以在属性 (如名称、颜色和大小) 和行为 (如计算、移动和通信) 方面合理地表示为软件对象。

可以将汽车比喻为类，汽车可以完成的任务就是成员函数。在C++中，我们经常创建一个称为类的程序单元来容纳一组函数执行类的任务——这些任务被称为类的成员函数。例如代表银行帐户的类可能包含用于存款的成员函数给一个账户，另一个从账户提款，第三个查询账户的当前余额是。一个类类似于一辆汽车的工程图纸，哪个房子设计了加速踏板、制动踏板、方向盘等。

就像有人必须根据工程图纸制造汽车一样驾驶汽车时，必须先从类中构建一个对象，然后程序才能执行任务类的成员函数定义的。这样做的过程称为实例化。然后，一个对象被称为其类的实例。

就像汽车的工程图纸可以多次重复使用来制造许多汽车一样，你可以多次重用一个类来构建许多对象。在构建新类和程序时重用现有类可以节省时间和精力。重用还可以帮助您构建更可靠、更有效的系统，因为现有的类和组件通常都经过了广泛的测试、调试和性能调优。正如可互换部件的概念对工业革命至关重要一样，可重用类对对象技术推动的软件革命也至关重要。

当你驾驶汽车时，踩下油门会向汽车发送执行任务的信息——也就是说，走得更快。类似地，您向对象发送消息。每个消息都实现为成员函数调用，告诉对象的成员函数执行其任务。例如，程序可能会调用特定银行帐户对象的存款成员函数来增加帐户余额。

汽车除了具有完成任务的能力外，还具有一些属性，如颜色、车门数量、油箱中的汽油量、当前速度和行驶总里程记录（即里程表读数）。与它的功能一样，汽车的属性也作为其设计的一部分在其工程图中表示（例如，包括里程表和燃油表）。当你驾驶一辆真正的汽车时，这些属性会随汽车一起携带。每辆车都有自己的特点。例如，每辆车都知道自己油箱里有多少油，但不知道其他车油箱里有多少油。

类似地，一个对象在程序中使用时也会附带一些属性。这些属性被指定为对象类的一部分。例如，银行帐户对象具有余额属性，表示帐户中的金额。每个银行帐户对象都知道它所代表的帐户中的余额，但不知道银行中其他帐户的余额。属性由类的数据成员指定。

类将属性和成员函数封装（即包装）到从创建的对象中这些类——对象的属性和成员函数是密切相关的。对象可以相互通信，但通常不允许它们知道其他对象是如何实现的——实现细节隐藏在对象本身中。正如我们将看到的，这种信息隐藏对于良好的软件工程至关重要。

一个新的对象类可以通过继承快速方便地创建——类吸收了现有类的特性，可能会对其进行自定义并添加独特的特点。在我们的汽车类比中，一个“敞篷”类的物体当然是更普通的“汽车”的一个对象，但更具体地说，车顶可以升高或降低。

类包含：

1.由变量定义的数据域

2.由函数定义的行为

代表类型的名字必须首字母大写。

2.2 对象的构成

对象是类的实例，具有唯一的标识、状态和行为：

1.对象状态由数据域(也称为属性)及其当前值构成

2.对象的行为由一组函数定义

在你创造一个类的对象之前，不可以调用类中的成员函数。

2.3 类：数据成员、set和get函数

2.3.1 类的定义

```
1 // Fig. 3.2: Account.h
2 // Account class that contains a name data member
3 // and member functions to set and get its value.
4 #include <string> // enable program to use C++ string data
5 // type
6
7 class Account {
8 public:
9     // member function that sets the account name in the
10    // object
11    void setName(std::string accountName) {
12        name = accountName; // store the account name
13    }
14
15    // member function that retrieves the account name from
16    // the object
17    std::string getName() const {
18        return name; // return name's value to this
19        // function's caller
20    }
21 }
```

```
17 private:
18     std::string name; // data member containing account
19 };// end class Account
```

上述代码即为Account类的定义。

2.3.2 关键词class和class body

1.在类的定义中，结束的大括号后面要写上分号。忘记类定义末尾的分号是语法错误。为了较好的可复用性，可以将类的定义单独写为一个.h文件。

2.在C++中，需要对类、对象和函数等命名，若名称中间没有空格和标点，除第一个单词外后面的单词首字母均大写。称为驼峰式命名。

如果第一个单词首字母大写，称之为 `upper camel case` (`camelCase`, 大驼峰式)，例如 `"GetUserName"`。

如果第一个单词首字母小写，称之为 `lower camel case` (`camelCase`, 小驼峰式)，例如 `"getUserName"`。

2.3.3 类数据成员

数据成员存在于：

1.在程序调用对象的成员函数之前

2.在成员函数执行之时

3.在成员函数完成执行之后

数据成员的声明在类的定义内，但在类的成员函数体之外声明。如上Account类的定义。按照惯例，一般将数据成员的定义放在class body的最后。

2.3.4 函数形参

在函数的形参中，每个形参必须指明类型。当一个函数有多个形参时，每个形参之间用逗号隔开。实参的个数和顺序必须和形参的个数顺序相同。函数的形参是局部变量。

实参和形参的类型必须一致，但不必相同

2.4 使用构造函数初始化对象

2.4.1 构造函数

构造函数是特殊的成员函数，必须和类的名字一样。C++在每个对象建立时，都要调用构造函数。像成员函数一样，构造函数也可以有形参，对应的实参值帮助初始化对象的数据成员。每个类都可以定义一个构造函数，该构造函数为该类的对象指定自定义初始化。

构造函数的特点：

- 1.在创建对象时，自动的进行初始化工作。
- 2.构造函数在声明时，前面不能加返回类型，void也不可以。
- 3.构造函数与类同名。
- 4.没有返回值。
- 5.可以重载构造函数。
- 6.构造函数可以不带参数。

默认构造函数：调用时可以不需要实参的构造函数。即参数表为空的构造函数或全部参数都有默认值的构造函数。

```
1  clock(){  
2      //class body  
3  } //默认构造函数，无参数，函数体可能为空  
4  clock(int n=0,int x=1,double y=0){  
5      //class body  
6  } //默认构造函数，默认参数，函数体可能为空
```

上述两种形式的构造函数如果同时在类中出现，将会产生编译错误。

默认构造函数的角色：

- 1.若内嵌对象成员没有被显式的初始化，该内嵌对象的无参构造函数会被自动调用。若内嵌对象没有无参构造函数，则编译器会报错。

```
1 class X{  
2     private:  
3         Circle c1; //c1即为内嵌对象  
4     public:  
5         X() {}  
6     };
```

2.也可以在初始化列表中手工构造对象。参见2.7节。

若类的数据域是一个对象类型(且它没有无参构造函数), 则该对象初始化可以放到类的构造函数初始化列表中。

创建对象:

```
1 Circle circle1; //正确, 但是不推荐这么写  
2 Circle circle(); //错误, 编译器会认为这是一个函数声明  
3 Circle circle3{}; //正确, 推荐写法。这里明确显式用空初始化列表初始化circle3对象(调用Circle的默认构造函数)
```

类可以不声明构造函数, 此情况下, 编译器会提供一个带有空函数体的无参构造函数

1.一个类可以有多个构造函数

```
1 class Student  
2 {  
3     Student() {}; //默认构造函数 default constructor  
4     Student(int age) {}; //非默认构造函数  
5     Student(int age, bool sex) {}; //非默认构造函数  
6 };  
7  
8 int main()  
9 {  
10    Student stu1; //调用默认构造函数  
11    Student stu2(10); //调用带一个整形参数的构造函数  
12    Student stu3(10, true); //调用两个参数的那个构造函数  
13  
14    return 0;  
15 }
```

2.编译器合成的默认构造函数

如果我们一个构造函数都没有定义，编译器也会为我们合成一个默认构造函数（default constructor）。

```
1 class Student
2 {
3 };
4
5 int main()
6 {
7     Student stu1; //调用编译器合成的默认构造函数（虽然我们没有看见那个函数在哪，编译器真是太热心了）
8
9     return 0;
10 }
```

3. 禁止编译器合成默认构造函数

只要我们自己定义了构造函数，编译器就不会再多管闲事替我们热心的合成一个默认的构造函数了。

下面的类已经没有默认构造函数了，因为我们定义了非默认构造函数，编译器不再替我们合成，而我们自己也没有定义默认构造函数。

```
1 class Student
2 {
3     Student(int age) {};//非默认构造函数
4     Student(int age, bool sex) {};//非默认构造函数
5 };
6
7 int main()
8 {
9     Student stu2(10); //调用带一个整形参数的构造函数
10    Student stu3(10, true); //调用两个参数的那个构造函数
11
12    return 0;
13 }
```

这样的类也有一个不便之处，那就是像下面这样定义对象会报错：

```
est_11_29 > demo.cpp > main()
1 #include <iostream>
2 using namespace std;
3
4 class Student{
5     Student(int age) {};//非默认构造函数
6     Student(int age, bool sex) {};//重载了构造函数
7 };
8
9 int main(){
10     Student stu1;
11 }
```

如果要解决这个问题，就只能像一开始那样，自己显式的定义一个默认构造函数。

如果类中已经定义构造函数，默认情况下编译器就不再隐含生成默认构造函数。如果此时依然希望编译器隐含生成默认构造函数，可以使用"=default"

```
1 class Clock{
2 public:
3     Clock()=default;//指示编译器提供默认构造函数
4     Clock(int n,intx,int y);//构造函数
5 private:
6     int hour,minute,second;
7 }
```

```
1 | Clock(){}  
|
```

使用上述形式的构造函数也可以满足需求，但是上述形式的构造函数在每次生成对象时都会调用，使用default只要在需要的时候才会调用。

2.4.2 对象访问运算符

使用点运算符访问对象中的数据和函数。

```
1 | Circle circle1;
2 | circle.radius=10;
3 | int area =circle1.getArea();  
|
```

2.4.3 explicit关键词

指定单个参数的构造函数应声明为explicit

2.4.4 在对象创立时初始化对象

```
1 #include <iostream>
2 #include "Account.h"
3
4 using namespace std;
5
6 int main(){
7     Account account1{"Jane Green"};
8     Account account2{"John Blue"};
9
10    cout<<"account1 name is: "<<account1.getName()<<endl;
11    cout<<"account2 name is: "<<account2.getName()<<endl;
12 }
```

在任何没有明确定义构造函数的类中，编译器都会提供一个没有参数的默认构造函数。默认构造函数不会初始化类的基本类型数据成员，但会为另一个类的对象的每个数据成员调用默认构造函数。例如在如下的代码中：

```
1 class Account{
2 public:
3     void setName(std::string accountName){
4         name=accountName;
5     }
6     std::string getName() const{
7         return name;
8     }
9 private:
10    std::string name;
11 }
```

上述例中的类的默认构造函数调用了string类的默认构造函数初始化名为name的数据成员为空的字符串。

一个未被初始化的基本类型变量包含一个未定义的值

如果你定义了一个传统的（custom）的构造函数在类中，编译器不会为类生成一个默认构造函数。在这种情况下，就不可以使用如下的语句创造一个Account对象，除非你定义的构造函数有一个空的形参列表。后续我们会知道，在C++11的标准中，允许你强迫编译器创造一个默认的构造函数即使你已经定义了非默认函数。除非类的数据成员的默认初始化是可接受的，否则通常应该提供一个自定

义构造函数，以确保在创建类的每个新对象时，数据成员都以有意义的值进行了正确的初始化。

```
1 | Account myAccount;
```

2.5 类的示例

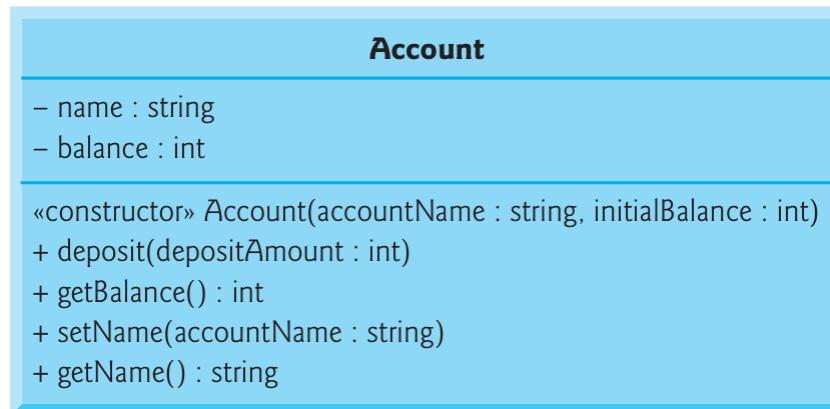
balance(余额)

2.5.1 UML类图

```
1 | #include <string>
2 | class Account {
3 | public:
4 |     // constructor initializes data member name with
5 |     // parameter accountname
6 |     Account(std::string accountName, int initialBalance)
7 |         :name{accountName} {
8 |         if (initialBalance>0) {
9 |             balance=initialBalance;
10 |         }
11 |         void deposit(int depositAmount) {
12 |             if(depositAmount>0) {
13 |                 balance=balance+depositAmount;
14 |             }
15 |         }
16 |
17 |         int getBalance() const{
18 |             return balance;
19 |         }
20 |         void setName(std::string accountName) {
21 |             name=accountName;
22 |         }
23 |
24 |         std::string getName() const{
25 |             return name;
26 |         }
27 |     private:
28 |         std::string name;
29 |         int balance{0};
30 |     };
```

在上述代码中，即使变量balance没有在任何一个成员函数中声明，所有的成员函数仍然可以使用balance，我们可以在这些成员函数中使用balance因为它是同一个类定义中的数据成员。

上诉类的UML类图为：



2.6 匿名对象

有时候需要创建一个只用一次的对象，此时，无需给对象命名，这种对象叫做匿名对象。匿名对象用在成员拷贝，如下例：

```
1 | classname(); //无参构造函数
2 | classname(arguments); //有参构造函数
```

```
1 | #include <iostream>
2 | using namespace std;
3 |
4 | class Circle{
5 | public:
6 |     double radius;
7 |     Circle(){
8 |         radius=1;
9 |     }
10 |
11 |     Circle(double newRadius){
12 |         radius=newRadius;
13 |     }
14 |
15 |     double getArea(){
16 |         return radius*radius*radius;
17 |     }
18 | }
```

```

18 };
19
20 int main(){
21     Circle circle1,circle2;
22     circle1=Circle();
23     circle2=Circle(5);
24
25     cout<<"area is"<<circle1.getAera()<<endl;
26     cout<<"area is"<<circle2.getAera()<<endl;
27     cout<<"area is"<<Circle().getAera()<<endl;
28     cout<<"area is"<<Circle(5).getAera()<<endl;
29 }

```

```

PS D:\vsc_project\vsCode_Cpp> & 'c:\Users\22364\.vscode\extensions\ms-vscode.cppTools-1.18.5-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-Ifftny4x.qdd' '--stdout=Microsoft-MIEngine-Out-z14mlbm2.tlt' '--stderr=Microsoft-MIEngine-Error-g5qdyo3d.ziq' '--pid=Microsoft-MIEngine-Pid-dlapzeby.sew' '--dbgExe=D:\MINGW\mingw64\bin\gdb.exe' '--interpreter=mi'
area is1
area is125
area is125
PS D:\vsc_project\vsCode_Cpp>

```

2.7 构造函数初始化列表

在构造函数中用初始化列表初始化数据域。

```

1 className (parameterList)
2     :dataField1{value1},dataField2{value2} //这一行就是初始化列
3     表
4 {
5     //do something
6 }

```

对于基础类型，下列两种写法效果相同。

```

1 Circle::Circle():radius{1}{}
2
3

```

```

1 Circle::Circle(){
2     radius=1;
3 }

```

使用构造函数初始化列表的原因：

因为在类的数据域中可能存在一个对象类型，此对象被称为对象中的对象，或者内嵌对象。

内嵌对象必须在被嵌对象的构造函数体执行完成前就构造完成。因此使用构造函数初始化列表

考虑下列代码：

```
1 class Time{  
2     //code omitted  
3 };  
4  
5 class Action{  
6     public:  
7         Action(int hour,int minute,int second){  
8             time=Time(hour,minute,second)  
9         }  
10  
11     private:  
12         Time time;  
13     };
```

在Action类中，构造函数的函数体中，对time对象执行赋值操作(使用Time创建的匿名对象)。也就是说在构造函数的函数体执行之前，time对象就要存在了，不存在的话就无法进行赋值操作。那么time对象在何处创建并初始化的呢？应该在函数体执行之前构造并初始化完成。即使用构造函数的成员初始化列表进行time的初始化。

若类的数据域是一个对象类型(且它没有无参构造函数)，则该对象初始化必须放在初始化列表内。

2.8 不可变对象

不可变对象，即对象创建后，其内容不可改变，除非通过成员拷贝

不可变类：不可变对象所属的类

让类变为不可变类的方法：

- 1.所有数据域均设置为private属性
- 2.没有更改器函数
- 3.也没有能够返回可变数据域对象的引用或指针的访问器

多线程编程和函数式编程会用到不可变类。

2.9 类的前向引用声明

类应该先声明，后引用

如果需要在某个类的声明之前，引用该类，则应进行前向引用声明

前向引用声明只为程序引入一个标识符，但具体声明在其他地方

2.10 常量(Constant)/const关键字

1. 常量是程序中的一块数据，这个数据一旦声明后就不能修改了。

如果这块数据有一个名字，这个名字就叫做命名常量。比如const int A=42,A就是命名常量

如果这个常量从字面上看就可以知道它的值，那它就叫做“字面值常量”，例如上例中的42即为字面值常量。

2. 符号常量(包括枚举值)必须全部大写并用下划线分隔单词

例如：MAX_ITERATIONS,COLOR_RED,PI

通过下列方式定义一个常量：

```
1 | const datatype CONSTANTNAME = VALUE; //const和数据类型可以互相  
  | 交换
```

```
1 | const double PI=3.14159;  
2 | const int SIZE=3;  
3 | int const X=5;  
4 | const int C=3;  
5 | const char C='k';  
6 | const char* STR="hello";
```

上述代码中，int const和const int等价。

PI、SIZE等就叫做命名常量或者符号常量

2.10.1 常量表达式(constant expressions)

1. 常量表达式是编译期可以计算值的一个表达式。

例如，C++数组的大小要求是编译期的一个常量(原生数组以及array)

```
1 int n=1;
2 n++;
3 array<int,n> a1;//error!n不是一个常量表达式
4 const int x=2;
5 array<int,x> a2;//正确！n是一个常量表达式
```

2. const修饰的对象未必是编译期常量

```
1 const int rcn=n;//rcn是一个运行期常量，编译器在编译期不知道它的值
2 rcn=++n;//
3 array<int,rcn> a3;//
4
```

2.10.2 constexpr:编译期常量表达式说明符

constexpr说明符声明可在编译时计算函数或变量的值

```
1 constexpr int max(int a , int b) { // c++11 引入 constexpr
2     if (a > b)
3         return a; // c++14才允许constexpr函数中有分支循环等
4     else
5         return b;
6 }
7
8 int main() {
9     int m = 1;
10    const int rcm = m++; // rcm是运行期常量
11    const int cm = 4; // 编译期常量，等价于: constexpr
12    int cm = 4;
13    int a1[ max(m , rcm)]; // 错误: m & rcm 不是编译期常量
14    std::array<char , max(cm , 5)> a2; // OK: cm 和 5 是编
15    译期常量
16 }
```

constexpr函数可以接受非常量实参，但此时的结果不再是一个常量表达式。

2.10.3 const vs constexpr

1. 主要区别

const: 告知程序员，const修饰的内容是不会被修改的。主要目的是帮程序员避免bug。

```
1 char* s1 = "Hello"; // C语言允许, 但C++编译出错
2
3 *s1 = 'h';           // C语言中, 语法正确, 但运行时会出错
4
5 const char* s2 = "world"; // C++ 要求加const
6
7 *s2 = 'w';           // C++编译器报错
```

constexpr：用在所有被要求使用“constant expression”的地方（就是constexpr修饰的东西可以在编译期计算得到值），主要目的是让编译器能够优化代码提升性能。

2 constexpr：初学者只需了解其含义即可

constexpr用法有非常多的细节（cppreference.com 列出了30多个条目）

C++14、C++17、C++20 对它都有细节修改

2.10.4 字面值(字面量)

C++中字面值常量是一类特殊的常量，它们没有名字，只能用它们的值来称呼，因此得名“字面值常量”。常见的字面值常量包括以下几类：

- 整型字面值常量：1,2,3,4,5等等
- 浮点型字面值常量：1.1,2.2,3.3等等
- 布尔类型字面值常量：true, false
- 字符字面值常量：'a','b','c','d'等等
- 字符串字面值常量："abc","def"等等

其中只有字符串字面值常量存储在全局区，可以取地址，其他的字面值常量都放在寄存器上，不能取内存地址。

比起字面值常量，使用const等定义的常量有一个可以称呼的名字，如 `const int a=2;` 名字就是 `a`

2.10.5 C++14 字符串字面量

1. C++11 Raw String literals (C++11“原始/生”字符串字面量)

从名字上就可以看出，这种“Raw String literals”应该长得很原始。那么这个原始该怎么体现出来呢？我们通过语法和简单示例就能看出来。

语法：

```
1 | R"delimiter( raw_characters )delimiter"
```

```
1 | #include <iostream>
2 | const char* s1 = R"(Hello
3 | World)";
4 |
5 | // s1效果与下面的s2和s3相同
6 | const char* s2 = "Hello\nWorld";
7 | const char* s3 = R"NoUse(Hello
8 | World)NoUse";
9 |
10 int main(){
11     std::cout << s1 << std::endl;
12     std::cout << s2 << std::endl;
13     std::cout << s3 << std::endl;
14 }
```

从例子中看出，“Raw String literals”在程序中写成什么样子，输出之后就是什么样子。我们不需要为“Raw String literals”中的换行、双引号等特殊字符进行转义。

2. C++14: String Literals (C++14的字符串字面量)

C++14将运算符 ""s 进行了重载，赋予了它新的含义，使得用这种运算符括起来的字符串字面量，自动变成了一个 std::string 类型的对象。

```
1 | auto hello = "Hello!"s;           // hello is of
2 | std::string type
3 |
4 | auto hello = std::string{"Hello!"}; // equals to the above
5 |
6 | auto hello = "Hello!";           // hello is of const
7 | char* type
```

例子：

```
1 #include <string>
2 #include <iostream>
3
4 int main() {
5     using namespace std::string_literals;
6     std::string s1 = "abc\0\0def";
7     std::string s2 = "abc\0\0def"s;
8     std::cout << "s1: " << s1.size() << " \\" " << s1 <<
9     "\\\n";
10    std::cout << "s2: " << s2.size() << " \\" " << s2 <<
11    "\\\n";
12 }
```

上面例子的一个可能输出结果如下：

```
D:\Code\cppProject\test240412b\cmake-build-debug\test240412b.exe
s1: 3 "abc"
s2: 8 "abc  def"

进程已结束，退出代码为 0
```

2.11 用指针访问对象成员(对象指针)

对象指针可以指向新的对象名。箭头运算符->,使用指针访问对象成员。

```
1 Circle circle;
2 Circle* pCircle=&circle1;
3
4 cout<<(*pCircle).radius<<endl;
5 cout<<(*pCircle).getArea<<endl;
6
7 (*pCircle).radius=5.5;
8
9 cout<<pCircle->radius<<endl;
10 cout<<pCircle->getArea()<<endl;
```

2.12 C++成员的就地初始化

非静态成员可以就地初始化。

在类中进行数组的就地初始化，编译器不能进行数组大小的自动推断。

```
1 class X{  
2     int a[] {1,4,5}; //错误!  
3     int a1[3] {1,4,5}; //正确!  
4 };
```

2.13 类的成员的初始化次序

执行次序：就地初始化->构造函数初始化列表->在构造函数体中为成员赋值

初始化优先级：在构造函数体中为成员赋值->构造函数初始化列表->就地初始化

若一个成员同时就地初始化和构造函数列表初始化，则就地初始化语句被忽略不执行。

2.14 断言(Assertion)与静态断言

断言是一条检测假设成立与否的语句。如果假设成立，断言不会产生作用，如果假设不成立，断言会终止程序的运行。

1.1. assert :C语言的宏(Macro)，运行时检测。

用法：

```
1 //包含头文件 <cassert> 以调试模式编译程序  
2  
3 assert( bool_expr ); // bool_expr 为假则中断程序
```

```
1 std::array a{ 1, 2, 3 }; //C++17 类型参数推导  
2  
3 for (size_t i = 0; i <= a.size(); i++) {  
4     assert(i < 3); //断言：i必须小于3，否则失败  
5     std::cout << a[ i ];  
6     std::cout << (i == a.size() ? "" : " " );  
7 }
```

1.2. assert()依赖于NDEBUG 宏

NDEBUG这个宏是C/C++标准规定的，所有编译器都有对它的支持。

(1) 调试(Debug)模式编译时，编译器不会定义NDEBUG，所以assert()宏起作用。

(2) 发行(Release)模式编译时，编译器自动定义宏NDEBUG，使assert不起作用

如果要强制使得assert()生效或者使得assert()不生效，只要手动#define NDEBUG 或者 #undef NDEBUG即可。

1.3. assert 帮助调试解决逻辑bug (部分替代“断点/单步调试”)

```
1 #undef NDEBUG // 强制以debug模式使用<cassert>
2
3 int main() {
4     int i;
5     std::cout << "Enter an int: ";
6     std::cin >> i;
7     assert((i > 0) && "i must be positive"); //&&是一个非空指
8     针，一定是真的。
9 }
```

上面示例的第6行代码中，若assert中断了程序则表明程序出bug了！程序员要重编代码解决这个bug，而不是把assert()放在那里当成正常程序的一部分

2.C++11: static_assert (C++11的静态断言)

2.1. static_assert (bool_constexpr, message)//C++17起， message可选

其中两个参数解释如下：

(1) bool_constexpr: 编译期常量表达式，可转换为bool 类型

(2) message: 字符串字面量，是断言失败时显示的警告信息。自C++17起，message是可选的

2.2. 作用：编译时断言检查

// 下面的语句能够确保该程序在32位的平台上编译进行。

// 如果该程序在64位平台上编译，就会报错

```
1 static_assert(sizeof(void *) == 4, "64-bit code generation
  is not supported.');
```

2.3. static_assert的用途

常用在模版编程中，对写库的作者用处大

在static_assert的第一个参数 bool_constexpr 中不能有变量表达式

3.何时使用断言

若某些状况是你预期中的，那么用错误处理；若某些状况永不该发生，用断言

```
1 int n{ 1 } , m{ 0 };
2 std::cin >> n;
3 assert((n != 0) && "Divisor cannot be zero!"); // 不合适
4 int q = m / n;
5
6
7 int n{ 1 } , m{ 0 };
8 do {           // 这是修补bug的代码
9     std::cin >> n; // 断言失败后，要解决这个bug
10 } while (n == 0); // 在这里编写修复bug的代码
11 assert((n != 0) && "Divisor cannot be zero!");
12 int q = m / n;
```

2.15 不可变对象和类

不可变对象：对象创建后，其内容不可改变，除非通过成员拷贝，对于编写多线程程序很有帮助。

不可变类：不可变对象所属的类。

如何让一个类成为不可变类：

- 1.所有数据域均设置为private属性
- 2.没有更改器函数
- 3.没有能够返回可变数据域对象的引用或指针的访问器

3.控制语句、运算符及bool类型

3.1 算法

任何计算问题都可以通过按特定顺序执行一系列操作来解决。用以下方法解决问题的程序：

1.要执行的操作

2.这些操作的执行顺序

就叫做算法。指定程序中语句(操作)执行的顺序称为程序控制。

3.3 伪代码(Pseudocode)

使用伪代码，不必担心C++中的细枝末节。伪代码并不在计算机上执行，一个精心准备的伪代码程序可以很容易地转换成相应的C++程序。在我们的伪代码，我们通常不包括变量声明，但是一些程序员选择列出变量并说明它们的用途。

3.4 控制结构

它们使您能够指定要执行的下一个语句不一定是按顺序执行的下一个语句。这就是所谓的控制权转移(transfer of control)。所有的程序都可以用三种结构表示：顺序结构、选择结构和循环结构。C++提供了四种循环语句，while, do...while, for和range-based for。C++的关键词如下：

C++ Keywords				
<i>Keywords common to the C and C++ programming languages</i>				
asm	auto	break	case	char
const	continue	default	do	double
else	enum	extern	float	for
goto	if	inline	int	long
register	return	short	signed	sizeof
static	struct	switch	typedef	union
unsigned	void	volatile	while	
<i>C++-only keywords</i>				
and	and_eq	bitand	bitor	bool
catch	class	compl	const_cast	delete
dynamic_cast	explicit	export	false	friend
mutable	namespace	new	not	not_eq
operator	or	or_eq	private	protected
public	reinterpret_cast	static_cast	template	this

C++ Keywords

throw	true	try	typeid	typename
using	virtual	wchar_t	xor	xor_eq
<i>C++11 keywords</i>				
alignas	alignof	char16_t	char32_t	constexpr
decltype	noexcept	nullptr	static_assert	thread_local

关键词必须小写，并且不能用作标识符。

3.5 带有初始化器的if和switch语句(C++17)

3.5.1 带有初始化器的if语句

不带初始化器的if语句：

```
1 int foo(int arg){  
2     return arg;  
3 }  
4 int main(){  
5     auto x=foo(42);  
6     if(x>40){  
7         //do something with x  
8     }  
9     else{  
10         //do something with x  
11     }  
12     //auto x=3;  
13 }
```

带初始化器的if语句：

```
1 int foo(int arg){  
2     return arg;  
3 }  
4 int main(){  
5     //auto x=foo(42);  
6     if(auto x=foo(42);x>40){  
7         //do something with x  
8     }  
9     else{  
10        //do something with x  
11    }  
12    auto x=3;//名字x可以重用  
13 }
```

为何使用带有初始化器的if语句：

- 1.本应限制于if块的变量，侵入了周边的作用域
- 2.若编译器确知变量作用域限于if块，则可以更好地优化代码

3.5.2 带有初始化器的switch语句

语法：

```
1 switch (initializer;variable);
```

示例：

```
1 switch(int i=rand()%100;i){  
2     case 1:  
3         //do something  
4     default:  
5         std::cout<<i<<endl;  
6         break;  
7 }
```

3.6 if...else

悬挂else问题(Dangling-else Problem)。

在整个文本中，我们总是将控制语句正文括在大括号中，从而避免了一个被称为悬挂else问题的逻辑错误。**else的匹配：不管你if—else是如何嵌套使用，对不对齐，你看的顺不顺眼，我else只会和离得最近的if去匹配**，所以要养成良好的习惯，即在if和else后都加上大括号。

3.6.1 三元运算符

?是C++唯一的三元运算符。可以在代码中代替if...else让代码更加简洁。例如：

```
1 | cout<<(studentGrade>=60? "passed":"failed");
```

若? 前的语句为真，则执行：前的内容，反之执行：后的内容。由于此运算符的优先级较低，所以一般将整个运算符放在括号内，在if...else不能执行功能时用来自代替。冒号两侧不光可以输出字符串，还可以执行操作，如下：

```
1 | grade>=60?cout<<"passed":"failed";
```

3.7 嵌套控制语句(narrowing conversion)

使用列表初始化防止narrowing conversion

A narrowing conversion changes a value to a data type that might not be able to hold some of the possible values. For example, a fractional value is rounded when it is converted to an integral type, and a numeric type being converted to Boolean is reduced to either True or False.

在此之前，初始化值可以写为

```
1 | int x=12.7
```

此种情况下，C++会浮点部分截断，即narrowing conversion，实际上赋给x的值是12。许多编译器会给出警告，但仍然会允许编译。但是使用列表初始化，例如：

```
1 | int x{12.7};  
2 | //or  
3 | int x={12.7};
```

就会产生编译错误，帮助你预防潜在的逻辑错误。

3.8 复合赋值运算符

Assignment operator	Sample expression	Explanation	Assigns
<i>Assume: int c = 3, d = 5, e = 4, f = 6, g = 12;</i>			
<code>+=</code>	<code>c += 7</code>	<code>c = c + 7</code>	10 to c
<code>-=</code>	<code>d -= 4</code>	<code>d = d - 4</code>	1 to d
<code>*=</code>	<code>e *= 5</code>	<code>e = e * 5</code>	20 to e
<code>/=</code>	<code>f /= 3</code>	<code>f = f / 3</code>	2 to f
<code>%=</code>	<code>g %= 9</code>	<code>g = g % 9</code>	3 to g

3.9 递增和递减运算符

即++ 和 --

Operator	Operator name	Sample expression	Explanation
<code>++</code>	prefix increment	<code>++a</code>	Increment a by 1, then use the new value of a in the expression in which a resides.
<code>++</code>	postfix increment	<code>a++</code>	Use the current value of a in the expression in which a resides, then increment a by 1.
<code>--</code>	prefix decrement	<code>--b</code>	Decrement b by 1, then use the new value of b in the expression in which b resides.
<code>--</code>	postfix decrement	<code>b--</code>	Use the current value of b in the expression in which b resides, then decrement b by 1.

3.10 运算符优先级以及关联

Operators	Associativity	Type
<code>:: ()</code>	left to right <i>[See Fig. 2.10's caution regarding grouping parentheses.]</i>	primary
<code>++ -- static_cast<type>()</code>	left to right	postfix
<code>++ -- + -</code>	right to left	unary (prefix)
<code>* / %</code>	left to right	multiplicative
<code>+ -</code>	left to right	additive
<code><< >></code>	left to right	insertion/extraction
<code>< <= > >=</code>	left to right	relational
<code>== !=</code>	left to right	equality
<code>? :</code>	right to left	conditional
<code>= += -= *= /= %=</code>	right to left	assignment

3.11 布尔数据类型

C++语言在其标准化过程中引入了 `bool`、`true` 和 `false` 关键字，增加了原生数据类型来支持布尔数据。布尔数据类型主要与条件语句相关。

例如：

```
1 | bool isMybook;  
2 | bool isRunning={false};  
3 | bool isBoy();
```

布尔数据类型与整型的转换：

整数0和布尔`false`互相转化

整数1和布尔`true`互相转化

任意非0整数和布尔`true`相互转化

❖ 关系运算得到布尔值

```
int a=0, b={1}; //C++11  
3 == a  
b < a  
3.2 >= b
```

Relational Operator:
`==, !=, <=, >=, <>`

❖ Example (示例)

```
if (3 == a) {  
    // blah blah  
}
```

❖ 逻辑运算得到布尔值

```
int a={0}, b{1}; //C++11  
a && b  
b || 18  
!a
```

Logical Operator:
`&&, ||, !`

❖ Example (示例)

```
while (!a) {  
    // blah blah  
}
```

编码规范：

布尔变量/函数的命名应该使用前缀"is"

```
1 | isMale, isOpen, isVisible
```

4.控制语句：逻辑运算符

4.1 for循环

for循环的基本结构为：

```
for (initialization; loopContinuationCondition; increment) {
    statement
}
```

两个分号是必不可少的，用while表示for可以写为：

```
initialization;

while (loopContinuationCondition) {
    statement
    increment;
}
```

如果在for循环中，第一部分被省略，C++会假定判断条件始终为真，循环一直进行。

程序经常在循环体中显示控制变量值或在计算中使用它，但这种使用不是必需的。控制变量通常用于控制迭代，尽管控制变量的值可以在for循环的主体中更改，但应避免这样做是因为这种做法可能会导致细微的错误。如果程序必须修改循环体中控制变量的值，请使用while而不是for。

setw(int n)是c++中在输出操作中使用的字段宽度设置，设置输出的域宽，n表示字段宽度。**只对紧接着的输出有效**，紧接着的输出结束后又变回默认的域宽。当后面紧跟着的输出字段长度小于n的时候，在该字段前面用空格补齐；当输出字段长度大于n时，全部整体输出。头文件为#include

```
1 #include <iostream>
2 #include <iomanip>
3
4 using namespace std;
5
6 int main()
7 {
8     // 开头设置宽度为 4，后面的 runoob 字符长度大于 4，所以不起
9     // 作用
10    cout << setw(4) << "runoob" << endl;
11    // 中间位置设置宽度为 4，后面的 runoob 字符长度大于 4，所以
12    // 不起作用
13    cout << "runoob" << setw(4) << "runoob" << endl;
14    // 开头设置间距为 14，后面 runoob 字符数为6，前面补充 8 个空
15    // 格
16    cout << setw(14) << "runoob" << endl;
17    // 中间位置设置间距为 14，后面 runoob 字符数为6，前面补充 8
18    // 个空格
19    cout << "runoob" << setw(14) << "runoob" << endl;
```

```
16     return 0;  
17 }
```

representational error---表征错误

不要使用double (或float) 类型的变量来执行精确的货币计算。浮点数的不精确性可能导致错误。

clunky---笨重

类型long long和其他C++的整数类型的范围在不同的平台会有所不同。

类成员函数是类的一个成员，它可以操作类的任意对象，可以访问对象中的所有成员。

C++中的abs()函数返回参数的绝对值。abs()方法在C++语言中，最早的C98版本中，只对double、float、long double类型生效，不支持int类型，作用是求数据的绝对值。从C++11开始，增加了对int整型数据类型的支持。

size函数用来获取字符串长度。

**请注意，类的任何成员函数可以直接调用任何其他成员函数来对类的同一对象执行操作

Banker's Rounding:

传统的“四舍五入”会让数量非常大的数据计算之后偏大，因为两端距离相等的时候始终选择绝对值大的，而“四舍六入五成双”的Banker's Rounding规则会让数据两端取舍的概率均等，因为对于不同的数值，偶数可能在左边，也可能在右边，计算之后的数据不会明显偏大或偏小。Banker's rounding就是对于0.5的情况，向最近的偶数舍入，例如0.5舍入为0，3.5和4.5都舍入为4。

4.2 switch

If no match occurs and the switch does not contain a default case, program control simply continues with the first statement after the switch.

如果没有匹配，并且不包含默认情况，程序控制只需继续执行switch后的第一条语句。

尽管case和default可以以任意顺序出现在switch中，但还是习惯于将default放在最后，当default在最后时，break就不是必须的。

在case后，除了可以使用整数，还可以使用字符常量(以单引号包裹)，还可以使用枚举常量(enum)。

注意1：switch语句中表达式类型只能是整型或者字符型。

注意2：case里如果没有break，那么程序会一直向下执行。

总结：与if语句比，对于多条件判断时，switch的结构清晰，执行效率高，缺点是switch不可以判断区间。

在case中，可以用x ... y 表示范围在[x,y]的值，两边都是闭区间。

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5
6     int month;
7     cin >> month;
8     if(month < 1 || month > 12) //优先判断是否合法月份
9         cout << "不合法" << endl;
10    else{
11        switch(month){ //根据月份判断
12            case 3 ... 5: //连续的值
13                cout << "春季" << endl; break;
14            case 6 ... 8:
15                cout << "夏季" << endl; break;
16            case 9 ... 11:
17                cout << "秋季" << endl; break;
18            default:
19                cout << "冬季" << endl;
20        }
21    }
22    return 0;
23 }
```

4.3 break

break语句在while、for、do...while或switch中执行时，会导致立即退出该语句——执行从循环语句之后的第一个语句开始。

4.4 continue

continue语句在while、for或do...while中执行时，跳过循环体中的其余语句，继续循环的下一次迭代。

4.5 逻辑操作符(与或非)

C++'s logical operators enable you to form more complex conditions by combining simple conditions. 逻辑操作符包括; &&(与)、 ||(或)、 ! (非)。

`boolalpha` 的作用是使bool型变量按照 `false`、`true` 的格式输出。如不使用该标识符，那么结果会按照 1、0 的格式输出。当使用`boolalpha`后，以后的bool类型结果都将以`true`或`false`形式输出，除非使用`noboolalpha`取消 `boolalpha`流的格式标志。

Lvalues can also be used as rvalues on the right side of an assignment, but not vice versa.

5. 函数和递归

inline functions ---内联函数 function template---函数模板

5.1 C++中的程序构件

Function	Description	Example
<code>ceil(x)</code>	rounds x to the smallest integer not less than x	<code>ceil(9.2)</code> is 10.0 <code>ceil(-9.8)</code> is -9.0
<code>cos(x)</code>	trigonometric cosine of x (x in radians)	<code>cos(0.0)</code> is 1.0
<code>exp(x)</code>	exponential function e^x	<code>exp(1.0)</code> is 2.718282 <code>exp(2.0)</code> is 7.389056

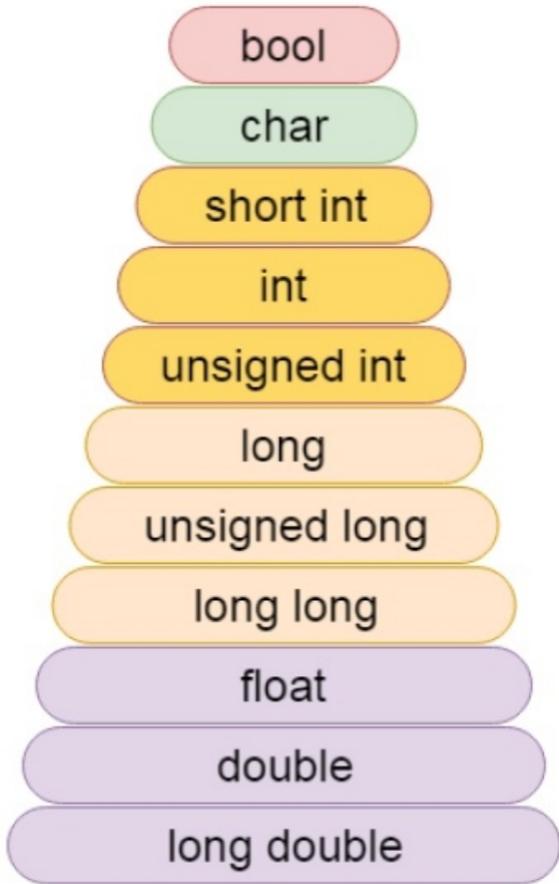
<code>fabs(x)</code>	absolute value of x	<code>fabs(5.1)</code> is 5.1 <code>fabs(0.0)</code> is 0.0 <code>fabs(-8.76)</code> is 8.76
<code>floor(x)</code>	rounds x to the largest integer not greater than x	<code>floor(9.2)</code> is 9.0 <code>floor(-9.8)</code> is -10.0
<code>fmod(x, y)</code>	remainder of x/y as a floating-point number	<code>fmod(2.6, 1.2)</code> is 0.2
<code>log(x)</code>	natural logarithm of x (base e)	<code>log(2.718282)</code> is 1.0 <code>log(7.389056)</code> is 2.0
<code>log10(x)</code>	logarithm of x (base 10)	<code>log10(10.0)</code> is 1.0 <code>log10(100.0)</code> is 2.0
<code>pow(x, y)</code>	x raised to power y (x^y)	<code>pow(2, 7)</code> is 128 <code>pow(9, .5)</code> is 3
<code>sin(x)</code>	trigonometric sine of x (x in radians)	<code>sin(0.0)</code> is 0
<code>sqrt(x)</code>	square root of x (where x is a nonnegative value)	<code>sqrt(9.0)</code> is 3.0
<code>tan(x)</code>	trigonometric tangent of x (x in radians)	<code>tan(0.0)</code> is 0

function prototype---函数原型

5.1.1 函数原型

A function prototype is a declaration of a function that tells the compiler the function's name, its return type and the types of its parameters.

参数强制转换是编译器可以将参数从一种类型隐式转换为另一种类型的一种技术。它遵循论据提升规则。如果一个参数是较低的数据类型，则可以将其转换为较高的数据类型，但反之则不成立。原因是如果将一个较高的数据类型转换为较低的数据类型，则可能会丢失一些数据。



5.2 C++库标头

Standard Library header	Explanation
<code><iostream></code>	Contains function prototypes for the C++ standard input and output functions, introduced in Chapter 2, and is covered in more detail in Chapter 13, Stream Input/Output: A Deeper Look.
<code><iomanip></code>	Contains function prototypes for stream manipulators that format streams of data. This header is first used in Section 4.10 and is discussed in more detail in Chapter 13, Stream Input/Output: A Deeper Look.
<code><cmath></code>	Contains function prototypes for math library functions (Section 6.3).
<code><cstdlib></code>	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Section 6.7; Chapter 11, Operator Overloading; Class <code>string</code> ; Chapter 17, Exception Handling: A Deeper Look; Chapter 22, Bits, Characters, C Strings and <code>structs</code> ; and Appendix F, C Legacy Code Topics.
<code><ctime></code>	Contains function prototypes and types for manipulating the time and date. This header is used in Section 6.7.
<code><array>, <vector>, <list>, <forward_list>, <deque>, <queue>, <stack>, <map>, <unordered_map>, <unordered_set>, <set>, <bitset></code>	These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The <code><vector></code> header is first introduced in Chapter 7, Class Templates <code>array</code> and <code>vector</code> ; Catching Exceptions. We discuss all these headers in Chapter 15, Standard Library Containers and Iterators. <code><array></code> , <code><forward_list></code> , <code><unordered_map></code> and <code><unordered_set></code> were all introduced in C++11.
<code><cctype></code>	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. These topics are discussed in Chapter 22, Bits, Characters, C Strings and <code>structs</code> .
<code><cstring></code>	Contains function prototypes for C-style string-processing functions. This header is used in Chapter 10, Operator Overloading; Class <code>string</code> .
<code><typeinfo></code>	Contains classes for runtime type identification (determining data types at execution time). This header is discussed in Section 12.9.
<code><exception>, <stdexcept></code>	These headers contain classes that are used for exception handling (discussed in Chapter 17, Exception Handling: A Deeper Look).

Standard Library header	Explanation
<code><memory></code>	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 17, Exception Handling: A Deeper Look.
<code><fstream></code>	Contains function prototypes for functions that perform input from and output to files on disk (discussed in Chapter 14, File Processing).
<code><string></code>	Contains the definition of class <code>string</code> from the C++ Standard Library (discussed in Chapter 21, Class <code>string</code> and String Stream Processing).
<code><sstream></code>	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 21, Class <code>string</code> and String Stream Processing).
<code><functional></code>	Contains classes and functions used by C++ Standard Library algorithms. This header is used in Chapter 15.
<code><iostream></code>	Contains classes for accessing data in the C++ Standard Library containers. This header is used in Chapter 15.
<code><algorithm></code>	Contains functions for manipulating data in C++ Standard Library containers. This header is used in Chapter 15.
<code><cassert></code>	Contains macros for adding diagnostics that aid program debugging. This header is used in Appendix E, Preprocessor.
<code><cfloat></code>	Contains the floating-point size limits of the system.
<code><climits></code>	Contains the integral size limits of the system.
<code><cstdio></code>	Contains function prototypes for the C-style standard input/output library functions.
<code><locale></code>	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).
<code><limits></code>	Contains classes for defining the numerical data type limits on each computer platform—this is C++’s version of <code><climits></code> and <code><cfloat></code> .
<code><utility></code>	Contains classes and functions that are used by many C++ Standard Library headers.

macro---宏

5.3 随机数的产生

随机数的产生可以使用`i=rand();`

`rand`函数产生一个0到`RAND_MAX`之间的无符号整数，如果`rand`真的产生了一个任意整数，那么在范围内的整数的出现概率是相同的。头文件为 `<cstdlib >`

为了产生范围在0-5之内的随机整数，我们使用：

```
1 | rand()%6;
```

由于要模拟骰子的点数，所以在后面+1.

C++14引入了一个新的语法特性，即数字分隔符（Digit Separators，也被称为"单下划线"）。这个特性的引入，主要是为了提高代码的可读性。在处理大量数字，特别是长数字串时，数字分隔符可以帮助我们更清晰地看到数字的大小和单位。例如，我们可以将一个长整数 1000000000 写成 1'000'000'000

rand实际上产生的是伪随机数(pseudorandom numbers), Repeatedly calling rand produces a sequence of numbers that appears to be random. However, the sequence repeats itself each time the program executes.

srand函数的头文件为:

需要一个无符号整数实参以及seeds rand函数来创造一组随机数, 且每次执行产生的一组随机数都是不同的。

rand()函数和srand()函数必须配套使用

```
1 #include <iostream>
2 #include <iomanip>
3 #include <cstdlib>
4 using namespace std;
5
6 int main() {
7     unsigned int seed{0};
8     cout<<"enter seed: ";
9     cin>>seed;
10    srand(seed);
11
12    for(unsigned int counter{1};counter<=10;++counter){
13        cout<<setw(10)<<(1+rand()%6);
14        if(counter%5==0){
15            cout<<endl;
16        }
17    }
18 }
```

还可以使用时间作为种子(seed), 为了不必每次都要输入seed, 我们可以使用如下代码:

```
1 srand(static_cast<unsigned int>(time(0)));
```

time函数返回自从1970年1月1号到当前时间的秒数, 返回的值类型为time_t, 头文件为, 由于srand需要无符号整数作为实参, 所以采用static_cast将之转化为合适的类型。

5.3.1 放缩及平移随机数

```
type variableName{shiftingValue + rand() % scalingFactor};
```

where the shiftingValue is equal to the first number in the desired range of consecutive integers and the scalingFactor is equal to the width of the desired range of consecutive integers

5.4 枚举类型(enum)

关键字enum class，后面为类型名和一组表示整数常量的标识符。例如下述代码：

```
1 | enum class Status{CONTINUE, WON, LOST};
```

CONTINUE代表0， WON代表1， LOST代表2

一个枚举类中的标识符必须是唯一的，但是不同的枚举常数可以有相同的整数值。用户自定义类型Status的变量只能赋值在枚举中声明的三个值中的一个。

按照惯例，你应该把一个enum class 名字的第一个字母大写。

另一个流行的scoped enumeration是：

```
1 | enum class Months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

第一个值被设置为1，剩下的值就会从1开始递增，任何一个枚举常数都可以在枚举定义中被赋予一个整数值，并且后续的枚举常数每个都比列表中的前一个常数高一个值1，直到下一个显式设置。

标识符表示的默认为整数(int)，但是也可以指定不同的类型。

```
1 | enum class Status : unsigned int {CONTINUE, WON, LOST};
```

5.5 C++11中的随机数

使用rand函数，不具备很好的统计特性，仍然可以被预测到，在C++11中，引入了头文件<random>

在本节中，我们将使用默认的随机数生成引擎--default_random_engine和均匀分布的uniform_int_distribution，将伪随机整数均匀分布在指定的取值范围内。默认范围是从0到你的平台上的一个int的最大值。

class template---类模板

```
1 #include <iostream>
2 #include <iomanip>
3 #include <random>
4 #include <ctime>
5 using namespace std;
6
7 int main() {
8     default_random_engine engine{static_cast<unsigned int>
9     (time(nullptr))};
10    uniform_int_distribution<unsigned int> randomInt{1,
11    6};
12
13    for(unsigned int counter{1}; counter<=10; ++counter){
14        cout<<setw(10)<<randomInt(engine);
15
16        if(counter%5==0){
17            cout<<endl;
18        }
19    }
20 }
```

5.6 作用域规则

当块是嵌套的，并且外块中的标识符与内块中的标识符具有相同的名字时，外块中的标识符被“隐藏”，直到内块终止--内块“看到”自己的局部变量的值而不是封闭块的相同命名变量的值。

局部变量也可以被声明为静态的。这样的变量也有块范围，但与其他局部变量不同的是，一个静态的局部变量在函数返回到它的调用者时保留了它的值。下次调用该函数时，**静态局部变量包含该函数上次执行完毕时的值**。

所有数值类型的静态局部变量默认初始化为零。下面的语句声明静态局部变量计数，并将其初始化为0：

```
1 | static unsigned int count;
```

An identifier declared outside any function or class has **global namespace scope**.

全局变量是通过在任何类或函数定义之外放置变量声明来创建的。这些变量在程序执行的整个过程中保持其值。

将变量声明为全局而非局部，当一个不需要访问变量的函数意外或恶意修改变量时，就会出现意想不到的副作用。这是最小特权原则的另一个例子- -除了真正的全局资源，如cin和cout，全局变量应该避免。一般来说，变量应该在其需要访问的最窄范围内进行声明。仅在特定函数中使用的变量应声明为该函数中的局部变量而不是全局变量。

函数原型中的参数，其作用域始于左括号，终于右括号

```
1 | double area(double radius); //radius的作用域仅在于括号内，不能用  
于程序正文以及其他地方
```

类的成员具有类作用域，其范围包括类体和非内联成员函数的函数体。

如果在类作用域以外访问类的成员，要通过类名(访问静态成员)，或者该类的对象名、对象引用、对象指针(访问非静态成员)。

5.7 函数调用栈和激活记录

5.7.1 栈

pushing---进栈 popping---出栈

栈是后人先出结构(last-in, first-out (LIFO) data structures)，即最后一个进栈的会最先出栈。

函数调用栈(function-call stack)，有时也称为程序执行栈，和数据结构意义上的“栈”不同。

栈帧(Stack Frames)

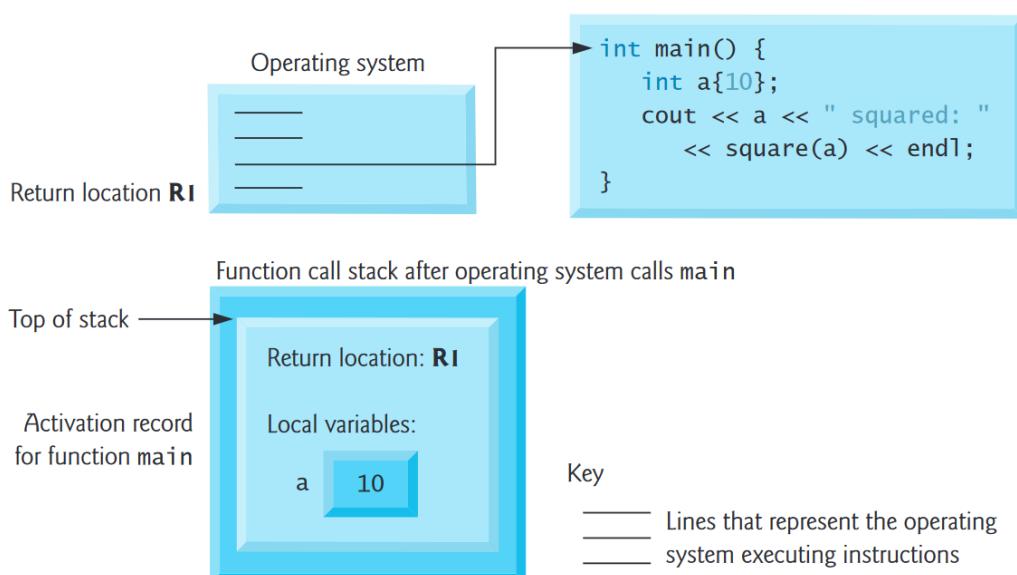
每次函数调用其他函数，一条记录就会放入栈中，这条记录，就叫做栈帧或者激活记录(activation record)，包含被调用函数为了返回调用函数所需要的返回地址。如果被调用的函数返回而不是在返回之前调用另一个函数，则弹出函数调用的堆栈帧，控制转移到弹出堆栈帧中的返回地址。If the called function returns instead of calling another function before returning, the stack frame for the function call is popped, and control transfers to the return address in the popped stack frame.

调用栈的美妙之处在于，每个被调用的函数总是在调用栈的顶端找到它需要返回给它的调用者的信息。并且，如果一个函数对另一个函数进行了调用，新函数调用的栈帧就被简单地推到了调用栈上。这样，新被调用函数返回其调用者所需的返回地址就位于栈的顶端。

函数在执行过程中需要存在非静态的局部变量。如果函数调用其他函数，它们需要保持活动状态。但是当一个被调用函数返回到它的调用者时，被调用函数的非静态局部变量需要“离开”。被调用函数的堆栈框架是为被调用函数的非静态局部变量预留内存的绝佳场所。只要被调用的函数是活动的，该栈帧就存在。当该函数返回而不再需要它的非静态局部变量时，它的堆栈框架从堆栈中弹出，并且这些非静态局部变量不再存在。

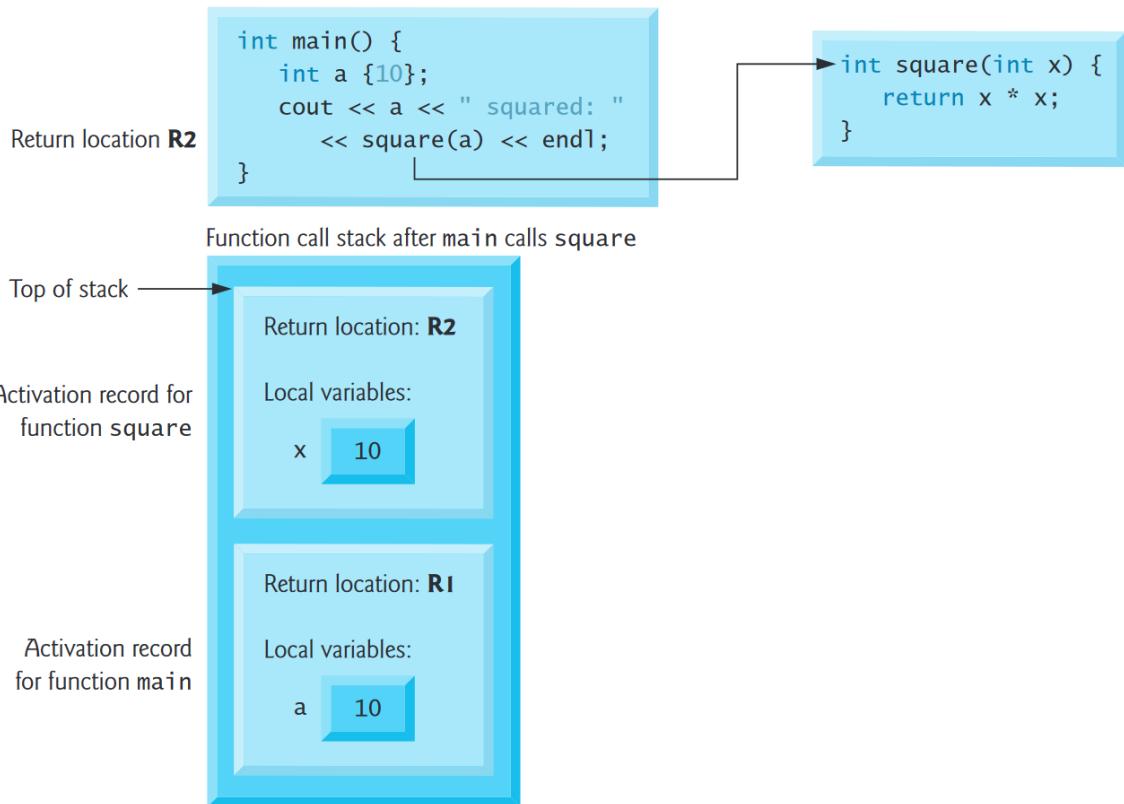
stack overflow(栈溢出)

Step 1: Operating system calls main to execute application

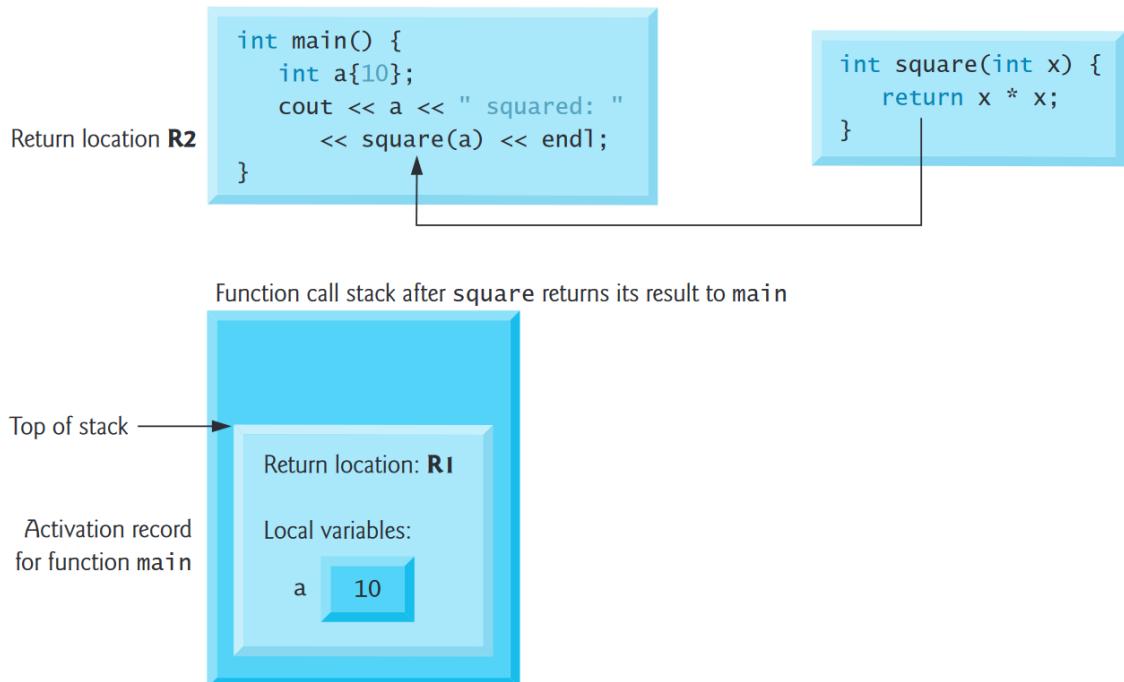


```
1 #include <iostream>
2 using namespace std;
3
4 int square(int);
5
6 int main(){
7     int a[10];
8     cout << a << " squared: " << square(a) << endl;
9 }
10
11 int square(int x){
12     return x*x;
13 }
```

Step 2: main calls function square to perform calculation



Step 3: square returns its result to main



5.8 内联函数(inline functions)

从软件工程的观点来看，将程序作为一组函数来实现是很好的，但是函数调用涉及执行时间开销。C++提供内联函数来帮助减少函数调用开销。

将限定符置于函数定义中函数的返回类型之前，建议编译器在函数调用的每个地方(适当的时候)生成函数体代码的副本，以避免函数调用。这往往会使程序变大。编译器可以忽略内联限定符，一般对除最小函数外的所有函数都这样做。可重复使用的内联函数通常放置在头文件中，因此它们的定义可以包含在每个使用它们的源文件中。

如果一个函数是内联的，那么在编译时，编译器会把该函数的代码副本放置在每个调用该函数的地方。对内联函数进行任何修改，都需要重新编译函数的所有客户端，因为编译器需要重新更换一次所有的代码，否则将会继续使用旧的函数。如果想把一个函数定义为内联函数，则需要在函数名前面放置关键字 **inline**，在调用函数之前需要对函数进行定义。

内联函数的声明和定义一般不分开(也可以分开，具体参见如下链接)[c++ - Is possible to separate declaration and definition of inline functions? - Stack Overflow](#)

对内联函数不能进行异常接口说明

内联函数的定义必须出现在内联函数第一次被调用之前。

如果已定义的函数多于一行，编译器会忽略 **inline 限定符。**

inline 只适合函数体内代码简单的函数使用，不能包含复杂的结构控制语句例如 **while**、**switch**，并且内联函数本身不能是直接递归函数(自己内部还调用自己的函数)。

以下情况不宜使用内联：

- (1) 如果函数体内的代码比较长，使用内联将导致内存消耗代价较高。
- (2) 如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。
- (3) 内联函数不能递归。

```
1 #include <iostream>
2 using namespace std;
3
4 inline double cube(const double side){
5     return side*side*side;
6 }
7
8 int main(){
9     double sideValue;
```

```
10     cout<<"Enter the side length of your cube: ";
11     cin >> sidevalue;
12
13     cout<<"Volume of cube with side "
14     <<sidevalue<<" is "<<cube(sidevalue)<<endl;
15 }
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

当函数在类的声明中实现，它自动成为内联函数。

```
1 class A{
2 public:
3     A()=default;
4     double f1(){
5         //do something
6     }
7     double f2();
8 };
9
10 double A::f2(){
11     //do something
12 }
13 //f1就变成了内联函数
14 //f2不是内联函数
```

5.9 函数默认参数

某些函数有这样一种形参，在函数的很多次调用中它们都被赋予一个相同的值，我们把这个反复出现的值称为函数的默认实参。

当程序在函数调用时，省略形参的默认实参，编译器会重写函数调用并向那个实参插入默认值。

```
1 #include <iostream>
2 using namespace std;
3
4 unsigned int boxvolume(unsigned int length=1,unsigned int
5 width=1,unsigned int height=1);
```

```
5
6 int main() {
7     cout<<"The default box volume is: "<<boxVolume();
8
9     cout<<"\n\nThe volume of a box with length 10,\n"
10    <<"width 1 and height 1 is: "<<boxVolume(10);
11
12    cout<<"\n\nThe volume of a box with length 10,\n"
13    <<"width 5 and height 2 is: "<<boxVolume(10,5,2)
14    <<endl;
15
16 unsigned int boxVolume(unsigned int length,unsigned int
17 width,unsigned int height){
18     return length*width*height;
19 }
```

在上述代码中，函数的原型指明三个形参都有默认值1。第一次调用boxVolume函数，没有指定实参，所以使用的是三个默认值1。第二次调用只传递了一个length实参，所以width和height仍然使用的默认实参1。最后一次调用，传入了三个实参，就不要默认实参了。

任何显式传递给函数的实参均从左至右赋值给函数的形参。

默认实参必须是函数形参列表最右边的实参。当调用的函数有两个或者多个默认实参，如果一个遗漏的实参不是最右边的实参，那么所有的实参都必须被省略。默认实参必须在函数名称首次出现时指明---典型的，在函数原型中。如果函数原型遗漏了，那么默认实参应该在函数头中指明。默认值可以是任意表达式，包括常量，全局变量或者函数调用。默认实参也可以在内联函数中使用。

```
1 定义时应当注意：参数列表中默认值参数应后置
2 void t1(int x,int y=0,int z);//错误
3 void t2(int x=0,int y=0,int z);//错误
4
5 void t3(int x,int y=0,int z=0);//正确
6 void t4(int x=0,int y=0,int z=0);//正确
7
8 调用时应当注意：参数列表中实参应前置
9 t3(1, ,7);//错误
10 t4( , ,6);//错误
11
12 t3(1);//正确，y,z使用默认值
13 t4(1,2);//正确，z使用默认值
```

函数重载时，不允许重定义默认参数。

```
1 #include <iostream>
2 using namespace std;
3
4 void printArea(double radius=1.0){
5     cout<<radius*2<<endl;
6 }
7
8 void printArea(int radius=2){
9     cout<<radius*radius<<endl;
10 }
11
12 int main() {
13
14     printArea();
15     printArea(4);
16
17     return 0;
18 }
19
```

```
===== [ 构建 | test240107 | Debug ] =====
"D:\CLION\CLion 2023.2.2\bin\cmake\win\x64\bin\cmake.exe" --build D:\Code\cppProject\test240407\cmake-build-debug
[1/2] Building CXX object CMakeFiles/test240107.dir/main.cpp.o
FAILED: CMakeFiles/test240107.dir/main.cpp.o
"D:\CLION\CLion 2023.2.2\bin\mingw\bin\g++.exe" -g -fdiagnostics-color=always -MD -MT CMakeFiles/test240107.dir/main.cpp.o -c D:/Code/cppProject/test240407/main.cpp -o D:/Code/cppProject/test240407/CMakeFiles/test240107.dir/main.cpp.o
D:/Code/cppProject/test240407/main.cpp: In function 'int main()':
D:/Code/cppProject/test240407/main.cpp:14:14: error: call of overloaded 'printArea()' is ambiguous
  14 |     printArea();
     |     ~~~~~~^~~
D:/Code/cppProject/test240407/main.cpp:4:6: note: candidate: 'void printArea(double)'
   4 | void printArea(double radius=1.0){
     |     ~~~~~~^~~
D:/Code/cppProject/test240407/main.cpp:8:6: note: candidate: 'void printArea(int)'
   8 | void printArea(int radius=2){
     |     ~~~~~~^~~
ninja: build stopped: subcommand failed.
```

5.10 一元作用域解析运算符(::)

当一个同名的局部变量在作用域时，C++提供一元作用域解析运算符(::)来访问全局变量。一元作用域解析运算符不能用于访问外部块中同名的局部变量。如果一个全局变量的名字与一个局部变量的名字在范围上不相同，则可以直接访问一个全局变量，而不需要一元作用域解析运算符。

```
1 #include <iostream>
2 using namespace std;
3 int number{7};
4 int main() {
5     double number{10.5};
6     cout << "Local double value of number = "
7     << number
8     << "\nGlobal int value of number = "
9     << ::number << endl;
10 }
```

```
:
D:\Code\cppProject\test240407\cmake-build-debug\test240107.exe
Local double value of number = 10.5
Global int value of number = 7
```

进程已结束，退出代码为 0

第二个作用是在类外定义函数，例如：

```
1 #include<iostream>
2 using namespace std;
3
4 class A
5 {
6 public:
```

```

7     // Only declaration
8     void fun();
9
10    };
11
12 // Definition outside class using ::
13 void A::fun()
14 {
15     cout << "fun() called";
16 }
17
18 int main()
19 {
20     A a;
21     a.fun();
22     return 0;
23 }
```

第三个作用是访问类的静态变量，参照本文档的8.14节

第四个作用是：**如果有多个继承**：如果两个祖先类中存在相同的变量名，则可以使用作用域运算符进行区分。例如：

```

1 // Use of scope resolution operator in multiple
2 // inheritance.
3 #include<iostream>
4 using namespace std;
5
6 class A
7 {
8 protected:
9     int x;
10 public:
11     A() { x = 10; }
12 };
13
14 class B
15 {
16 protected:
17     int x;
18 public:
19     B() { x = 20; }
```

```
20
21 class C: public A, public B
22 {
23 public:
24     void fun()
25     {
26         cout << "A's x is " << A::x;
27         cout << "\nB's x is " << B::x;
28     }
29 }
30
31 int main()
32 {
33     C c;
34     c.fun();
35     return 0;
36 }
```

第五个作用是：**对于命名空间**：如果两个命名空间中都存在一个具有相同名称的类，则可以将名称空间名称与作用域解析运算符一起使用，以引用该类而不会发生任何冲突

```
1 #include<iostream>
2 int main(){
3     std::cout << "Hello" << std::endl;
4 }
```

在这里，cout和endl属于std命名空间。具体参照本文的8.14.5

第六个作用是：**在另一个类中引用一个类**：如果另一个类中存在一个类，我们可以使用嵌套类使用作用域运算符来引用嵌套的类。

```
1 #include<iostream>
2 using namespace std;
3
4 class outside
5 {
6 public:
7     int x;
8     class inside
9     {
10 public:
```

```
11         int x;
12         static int y;
13         int foo();
14
15     };
16 };
17 int outside::inside::y = 5;
18
19 int main(){
20     outside A;
21     outside::inside B;
22
23 }
```

5.11 重载函数(Overloading Functions)

C++允许多个相同名称的函数被定义，只要它们具有不同的特征。这就叫做函数重载。

C++通过验证引用的实参的数量、数据类型和顺序来选择合适的函数调用。函数重载被用来生成多个执行相似任务、具有相同名字的函数，但数据类型不同。

```
1 #include <iostream>
2 using namespace std;
3
4 int square(int x){
5     cout<<"square of integer "<<x<<"is";
6     return x*x;
7 }
8
9 double square(double y){
10    cout<<"square of double "<<y<<"is";
11    return y*y;
12 }
13
14 int main(){
15     cout<<square(7);
16     cout<<endl;
17     cout<<square(7.5);
18     cout<<endl;
```

重载函数通过他们的特征来区分，特征(signatures)是函数的名字和形参类型(按顺序)的结合。编译器需要为C++中的所有函数，在符号表中生成唯一的标识符，来区分不同的函数。而对于同名不同参的函数，编译器在进行 `name mangling` 操作时，会通过函数名和其参数类型生成唯一标识符，来支持函数重载，以实现类型安全的联结属性(type-safe linkage)。类型安全的联结属性保证了合适的重载函数被调用，并且实参的类型和形参的类型一致。

注意：`name mangling` 后得到的函数标识符与返回值类型是无关的，因此函数重载与返回值类型无关。

- 函数的参数个数、参数类型、参数顺序不同三者中满足其中一个，就是函数重载了
- 如果只有函数返回值不同，不是函数重载；返回值不同，参数也不同的时候，可以作为函数重载

函数的重载的规则：

- 函数名称必须相同。
- 参数列表必须不同（个数不同、类型不同、参数排列顺序不同等）。
- 函数的返回类型可以相同也可以不相同。
- **仅仅返回类型不同不足以成为函数的重载。**
- 如果重载函数有默认参数，调用函数时，可能导致匹配失败
- `const`不能作为函数重载的特征

```
1 // Fig. 6.21: fig06_21.cpp
2 // Name mangling to enable type-safe linkage.
3
4 // function square for int values
5 int square(int x) {
6     return x * x;
7 }
8
9 // function square for double values
10 double square(double y) {
11     return y * y;
12 }
13
14 // function that receives arguments of types
15 // int, float, char and int&
16 void nothing1(int a, float b, char c, int& d) { }
17
18 // function that receives arguments of types
19 // char, int, float& and double&
20 int nothing2(char a, int b, float& c, double& d) {
21     return 0;
22 }
23
24 int main() { }
```

```
__Z6squarei
__Z6squared
__Z8nothing1ifcRi
__Z8nothing2ciRfRd
main
```

上述代码就显示了编译器以汇编语言生成的函数名(mangled function names)。对于GNU C++ 每个mangled name(main函数除外)都以两个下划线开始，后面跟字母Z、一个数字和函数的名字，Z后面的数字指明函数名包括多少个字符。例如square函数的形参为int类型，就在square后加i。在nothing1中，形参类型分别为int、float、char和int &，所以nothing后加ifcRi。

重载函数可以有不同的返回类型，但是如果返回类型不同，他们必须有不同的形参列表。

5.12 函数模板(function templates)

重载函数通常用于在不同数据类型上执行涉及不同程序逻辑的类似操作。如果每种数据类型的程序逻辑和操作完全相同，使用函数模板可以更紧凑、方便地执行重载操作。编写了以一个函数的模板定义，C++根据对该函数调用提供的参数类型，自动生成单独的函数模板特殊化(function template specializations)，以妥善处理每种类型的调用。因此，定义一个函数模板本质上是定义了一整族重载函数。

C++提供了模板(template)编程的概念。所谓模板，实际上是建立一个通用函数或类，其类内部的类型和函数的形参类型不具体指定，用一个虚拟的类型来代表。这种通用的方式称为模板。

在 C++ 中，模板分为函数模板和类模板两种。函数模板是用于生成函数的，类模板则是用于生成类的。

函数模板的写法如下：

```
1 template <typename 类型参数1, typename 类型参数2, ...>
2 返回值类型 模板名(形参表)
3 {
4     函数体
5 }
```

函数模板看上去就像一个函数。

例如对于交换多种类型的值的函数，可以编写函数模板如下：

```
1 template <typename T>
2 void Swap(T & x, T & y)
3 {
4     T tmp = x;
5     x = y;
6     y = tmp;
7 }
```

上述代码形参列表中的形参不能改成不带引用的形式：

```
1 void Swap(T x, T y)
```

上述形式的代码不能实现两个数值的交换，因为在将实参传递给形参后，是形参进行的数值交换，main函数体内的实参并没有进行数值交换。所以要使用引用。

类模板的声明

◦ 类模板

```
template<模板参数表>
class类名
{类成员声明}
```

◦ 如果需要在类模板以外定义其成员函数，则要采用以下的形式：

```
template <模板参数表>
类型名 类名<模板参数标识符列表>::函数名 (参数表)
```

同一个类型参数只能替换为同一种类型。编译器在编译到调用函数模板的语句时，会根据实参的类型判断该如何替换模板中的类型参数。

所有的模板定义都以关键词template开始(第一行)，后面跟模板形参列表。每个模板形参列表内的形参都在前面加上关键词typename或者class(在此种情形下，两者等价)。template parameters有三种类型：

- 1) Type parameters (类型参数) :
- 2) Nontype parameters (非类型参数) :
- 3) template template parameters (双重模板参数)

类型参数是基本类型或用户自定义类型的占位符。占位符就是先占用一个固定的位置，等着你再往里面添加内容的符号，广泛用于计算机中各类文档的编辑。格式占位符(%)是在C/C++语言中格式输入函数，如 scanf、printf 等函数中使用。其意义就是起到格式占位的意思，表示在该位置有输入或者输出。

这些占位符，例如T，被用来指定函数形参的类型(例如第二行)和指定函数的返回类型(第二行)，还有在函数体内声明变量(第三行)。一个函数模板的定义就像函数的定义一样，只是使用类型参数作为实际数据类型的占位符。

```
1 template<typename T>
2 T maximum(T value1, T value2, T value3){
3     T maximumValue{value1};
4
5     if(value2>maximumValue){
6         maximumValue=value2;
7     }
8
9     if(value3>maximumValue){
10        maximumValue=value3;
11    }
12
13     return maximumValue;
14 }
```

函数模板在第一行声明了单个类型参数T作为函数maximum待测数据类型的占位符。对于特定的模板定义，类型参数的名称必须在模板形参列表中唯一。当编译器在程序源代码中检测到maximum调用时，在整个模板定义中将maximum调用中的实参类型替换成T，C++创建了一个完整的函数，用于确定指定类型的3个值中的最大值--这3个值必须具有相同的类型，因为在本例中我们只使用了一个类型参数。然后对新创建的函数进行编译--模板是代码生成的一种手段。

```

1 #include <iostream>
2 #include "maximum.h"
3 using namespace std;
4
5 int main(){
6     cout<<"input three integer values: ";
7     int int1,int2,int3;
8     cin>>int1>>int2>int3;
9     cout<<"the maximum integer value is: "
10    <<maximum(int1,int2,int3);
11
12    cout << "\n\nInput three double values: ";
13    double double1, double2, double3;
14    cin >> double1 >> double2 >> double3;
15    cout << "The maximum double value is: "
16    <<maximum(double1, double2, double3);
17
18    cout<<"\n\nInput three characters: ";
19    char char1, char2, char3;
20    cin >> char1 >> char2 >> char3;
21    cout << "The maximum character value is: "
22    << maximum(char1, char2, char3)<< endl;
23 }

```

为type int创建的函数模板特殊化将T的每个出现替换为int如下：

```

int maximum(int value1, int value2, int value3) {
    int maximumValue{value1}; // assume value1 is maximum
    // determine whether value2 is greater than maximumValue
    if (value2 > maximumValue) {
        maximumValue = value2;
    }
    // determine whether value3 is greater than maximumValue
    if (value3 > maximumValue) {
        maximumValue = value3;
    }
    return maximumValue;
}

```

函数模板不是函数，写了函数模板，但不在任何地方使用它(也不显式实例化)，则编译器不会为该函数模板生成任何代码。函数模板实例化分为隐式实例化和显式实例化。

5.12.1 隐式实例化

仍以 swap 为例，我们在main 中调用 swap(a,b)时，就发生了隐式实例化。当函数模板被调用，且在之前没有显式实例化时，即发生函数模板的隐式实例化。如果模板实参能从调用的语境中推导，则不需要提供。效率较低。

```
1 //template2.cpp #include
2 template void print(const T &r) {
3     std::cout << r << std::endl;
4 }
5 int main() {
6     // 隐式实例化print(int) print(1);
7     // 实例化 print(char) print<>('c');
8     // 仍然是隐式实例化，我们希望编译器生成print(double) print(1);
9     return 0;
10 }
```

5.13 递归(recursion)

对于一些问题，函数调用自身是有用的。递归函数是一个可以直接或间接调用自身的函数(通过另一个函数)。C++标准文件指出，在程序中不应该调用main函数，也不应该递归调用main函数。它的唯一目的是作为程序执行的起点。该函数只知道如何求解最简单的情况(simplest case)，或所谓的基本情况(base case)。如果以base case调用该函数，该函数返回一个结果。如果函数被更复杂的问题调用，它通常将问题分为两个概念部分--函数知道如何做的部分和它不知道如何做的部分。为了使递归可行，后一部分问题必须与原问题相似，不过是略微简单或更小的版本。这个新问题看起来像原来的问题，所以函数调用自身的一个副本在更小的问题上--这被称为递归调用，也被称为递归步骤。递归步骤通常包括关键字return，因为它的结果会与函数知道如何解决的部分问题相结合，形成结果传回原调用者，可能是main。

递归步骤在执行时，对函数的原始调用仍然是"开放的"，即还没有执行完毕。递归步骤可以导致更多的递归调用，因为函数不断地将每个新的子问题划分为两个概念部分。为了使递归最终终止，每次函数调用自身与原问题稍微简单的版本，这个越来越小的问题序列最终必须收敛于base case。此时，函数识别出base case并返回一个结果到函数的前一个副本，然后一系列的返回沿着这个行进行，直到原始的调用最终返回最终的结果给main。

5.13.1阶乘

```
1 #include <iostream>
2 #include <iomanip>
3
4 using namespace std;
5 unsigned long factorial(unsigned long);
6
7 int main() {
8     for(unsigned int counter{0}; counter<=10; ++counter){
9         cout<<setw(2)<<counter<<"!="<<factorial(counter)
10    <<endl;
11 }
12
13 unsigned long factorial(unsigned long number){
14     if(number<=1){
15         return 1;
16     }
17     else{
18         return number * factorial(number-1);
19     }
20 }
```

factorial函数的形参类型为unsigned long, 返回结果的类型为unsigned long, 这是unsigned long int的简写。C++标准规定unsigned long int 的长度至少要和int一样。一般的, unsigned long int在计算机内存储为四个字节。

5.14 使用递归的例子：斐波那契数列

C++只规定了4种操作符的求值顺序--&&, ||, 逗号(,)和 (?) 。

5.15 递归VS.迭代

1.迭代和递归都是基于一个控制语句：迭代使用一个迭代语句；递归使用了一个选择语句。

2.迭代和递归都涉及迭代：迭代显式地使用一个迭代语句；递归通过重复的函数调用实现迭代。

3.迭代和递归各自涉及一个终止测试：当循环继续条件失败时，迭代终止；递归在识别base case时终止。

4.counter控制的迭代和递归每次逐渐接近终止：迭代修改counter，直到counter假设一个值使得循环连续条件失败；递归产生原始问题的更简单的版本，直到达到base case。

5.迭代和递归都可以无限地发生：如果循环延续性测试从未变得错误，则随着迭代的进行会出现一个无限的循环；如果递归步骤在每次递归调用过程中没有以收敛于base case的方式减少问题，则会发生无限递归。

```
1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 unsigned long factorial(unsigned int);
6
7 int main() {
8     for(unsigned int counter{0}; counter<=10; ++counter){
9         cout<<setw(2)<<counter<<"!"<<factorial(counter)
10        <<endl;
11    }
12 }
13
14 unsigned long factorial(unsigned int number){
15     unsigned long result{1};
16
17     for(unsigned int i{number}; i>=1; --i){
18         result*=i;
19     }
20     return result;
21 }
```

上述代码使用迭代的方法实现了斐波那契数列。

5.15.1 递归的负面效果

递归具有负面效应。它反复调用函数的机制，进而调用函数的开销。这在处理器时间和内存空间上都可能是昂贵的。每一次递归调用都会导致函数变量的另一个副本被创建；这会消耗相当大的内存。迭代通常发生在函数内部，因此省略了重复函数调用和额外内存分配的开销。

仍然选择递归的原因是**任何可以递归求解的问题也可以迭代求解(非递归)**。当递归方法更自然地反映问题并产生更易于理解和调试的程序时，通常选择递归方法。选择递归解的另一个原因是，当递归为解时，迭代解可能不明显。

5.16 round函数

使用round函数解决四舍五入问题比较简单。

在C++中，round()函数是[标准库](#)中的一个函数，用于对浮点数进行四舍五入。

函数原型为：

```
1 | double round(double x);
```

参数x是一个双精度浮点数。

round()函数将返回最接近参数x的整数。如果x正好在两个整数中间，则向远离零的整数方向取整。例如：

```
1 | #include <iostream>
2 | #include <cmath>
3 |
4 | int main() {
5 |     double num1 = 2.5;
6 |     double num2 = 2.3;
7 |     std::cout << "round(2.3) is: " << round(num1) <<
8 |     std::endl; // 输出 "round(2.3) is: 2"
9 |     std::cout << "round(2.3) is: " << round(num2) <<
10 |    std::endl; // 输出 "round(-2.3) is: -2"
11 |    return 0;
12 | }
```

```
D:\clionproject\test_11_17\cmake-build-debug\test_11_17.exe
round(2.3) is: 3
round(-2.3) is: 2
```

进程已结束，退出代码为 0

5.17 函数的参数

1.在函数被调用时才分配形参的存储单元

2.实参可以是常量、变量或表达式、

3.实参类型必须与形参相符或可以隐式转换为形参类型

4.值传递是传递参数值，即单向传递

5.引用传递可以实现双向传递

6. 常引用(const)作参数可以保障实参数数据的安全

当函数执行完毕，在函数体内定义的变量，全部都被释放了。引用在定义时必须初始化，但是在函数的形参中使用引用并不会进行初始化。这是因为引用在定义时必须初始化，是在为引用分配内存时，必须进行初始化，但函数的形参，系统并不会为他分配内存，所以就不需要初始化。

5.17.1 可变数量形参

使用模板类initializer_list可以向函数传递同类型不定个数参数，例如：

```
1 void log_info(initializer_list<string> lst) {  
2     for(auto &info: lst){  
3         cout<<info<< ' ';  
4     }  
5     cout<<endl;  
6 }  
7 log_info({"hello", "world", "!"});
```

5.18 对象作为函数参数和返回值

5.18.1 作为参数

对象作为函数参数，可以按值传递，也可以按引用传递。

```
1 //按值传递  
2 void print(Circle c){  
3     //do something  
4 }  
5  
6 int main(){  
7     Circle myCircle{5.0};  
8     print(myCircle);  
9 }
```

```
1 //按值传递
2 void print(Circle& c){
3     //do something
4 }
5
6 int main(){
7     Circle myCircle{5.0};
8     print(myCircle);
9 }
```

```
1 //按值传递
2 void print(Circle* c){
3     //do something
4 }
5
6 int main(){
7     Circle myCircle{5.0};
8     print(&myCircle);
9 }
```

5.18.2 作为函数返回值

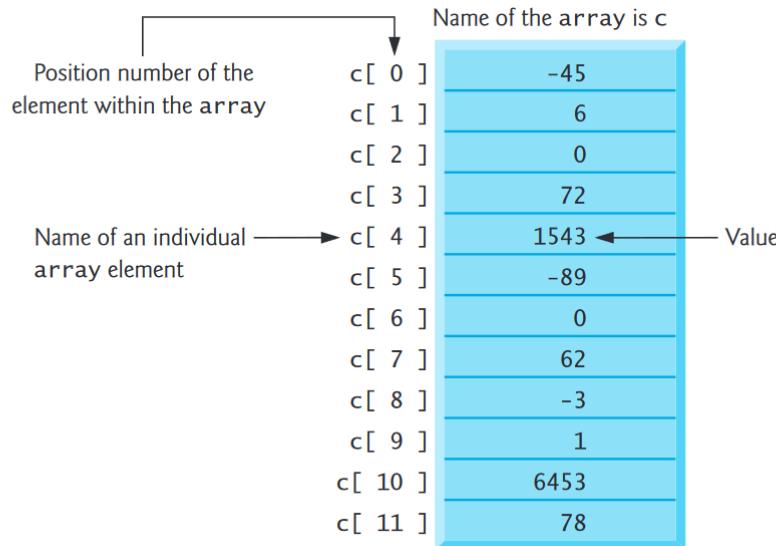
形式如下：

```
1 Object f(函数形参){
2     //do something
3     return Object(args);
4 }
5
6 // main(){
7     Object o=f;
8 }
9
```

6. 类-数组(array)和string

6.1 数组

数组是一组连续的内存位置，它们都具有相同的类型。为了指代数组中的特定位置或元素，我们指定数组的名称和特定元素在数组中的位置编号。数组名称遵循与其他变量名相同的约定。



下标必须是整数或整数表达式，带下标的数组名是一个左值，它可以在赋值的左边使用，就像非数组变量名一样。通过调用函数可以知道数组的长度，例如 `c.size()`。包含下标的括号实际上是一个操作符，它与用于调用函数的括号具有相同的优先级。

Operators	Associativity	Type
<code>::</code> <code>()</code>	left to right <i>[See caution in Fig. 2.10 regarding grouping parentheses.]</i>	primary
<code>()</code> <code>[]</code> <code>++</code> <code>--</code> <code>static_cast<type>(operand)</code>	left to right	postfix
<code>++</code> <code>--</code> <code>+</code> <code>-</code> <code>!</code>	right to left	unary (prefix)
<code>*</code> <code>/</code> <code>%</code>	left to right	multiplicative
<code>+</code> <code>-</code>	left to right	additive
<code><<</code> <code>>></code>	left to right	insertion/extraction
<code><</code> <code><=</code> <code>></code> <code>>=</code>	left to right	relational
<code>==</code> <code>!=</code>	left to right	equality
<code>&&</code>	left to right	logical AND
<code> </code>	left to right	logical OR
<code>?:</code>	right to left	conditional
<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code>	right to left	assignment
<code>,</code>	left to right	comma

6.2 声明array

数组在内存中占据空间，使用声明的形式来指定数组所需的元素类型和元素个数。数组和向量都是类模板。使用数组必须使用头文件<array>

```
array<type, arraySize> arrayName;
```

例如：

```
array<int, 12> c; // c is an array of 12 int values
```

数组 (array)

- ◆ array是对内置数组的封装，提供了更安全，更方便的使用数组的方式
- ◆ array的对象的大小是固定的，定义时除了需要指定元素类型，还需要指定容器大小。
- ◆ 不能动态地改变容器大小

```
1 | array<type, arraySize> arrayName{值1, 值2, ...};
```

6.2.1 C++17关于array的新特性

C++17引入了一种新特性，对类模板的参数进行推导，例如：

```
1 | std::array a1{1, 4, 6};  
2 | std::array a2{'w', 'b', 'o'};
```

6.3 使用数组的例子

6.3.1 声明数组并使用循环初始化数组的元素

size_t是一种数据相关的无符号整数类型，它被设计得足够大以便能够内存中任意对象的大小。size_t由来：在C++中，设计size_t就是为了适应多个平台的。size_t的引入增强了程序在不同平台上的可移植性。在需要通过数组下标来访问数组时，通常建议将下标定义size_t类型，因为一般来说在进行下标访问时，下标都是正的。这种类型对于任何表示数组大小或数组下标的变量都是推荐的。头文件为<cstddef>。此头文件包含在各种其他的头文件中，如果程序出现错误，说size_t未定义，只需要在程序中加上#include<cstddef>

```
1 | #include <iostream>
```

```

2 #include <iomanip>
3 #include <array>
4 using namespace std;
5
6 int main(){
7     array<int,5> n;
8     for(size_t i{0};i<n.size();++i){
9         n[i]=0;
10    }
11    cout<<"element"<<setw(10)<<"value"<<endl;
12
13    for(size_t j{0};j<n.size();++j){
14        cout<<setw(7)<<j<<setw(10)<<n[j]<<endl;
15    }
16 }

```

上述代码将数组初始化并将数组内元素打印出来。

6.3.2 在声明里通过初始化列表初始化数组

```

1 // Fig. 7.4: fig07_04.cpp
2 // Initializing an array in a declaration.
3 #include <iostream>
4 #include <iomanip>
5 #include <array>
6 using namespace std;
7
8 int main() {
9     array<int, 5> n{32, 27, 64, 18, 95}; // list initializer
10
11    cout << "Element" << setw(10) << "Value" << endl;
12
13    // output each array element's value
14    for (size_t i{0}; i < n.size(); ++i) {
15        cout << setw(7) << i << setw(10) << n[i] << endl;
16    }
17 }

```

如果初始化列表内的元素个数小于数组长度，剩余的元素会被初始化为0。如果大于数组长度，则是错误。

6.3.3 用常量设定数组的大小并通过计算设置数组元素

```
1 // Fig. 7.5: fig07_05.cpp
2 // Set array s to the even integers from 2 to 10.
3 #include <iostream>
4 #include <iomanip>
5 #include <array>
6 using namespace std;
7
8 int main() {
9     // constant variable can be used to specify array size
10    const size_t arraySize{5}; // must initialize in declaration
11
12    array<int, arraySize> values; // array values has 5 elements
13
14    for (size_t i{0}; i < values.size(); ++i) { // set the values
15        values[i] = 2 + 2 * i;
16    }
17
18    cout << "Element" << setw(10) << "Value" << endl;
19
20    // output contents of array s in tabular format
21    for (size_t j{0}; j < values.size(); ++j) {
22        cout << setw(7) << j << setw(10) << values[j] << endl;
23    }
24 }
```

第10行使用const限定符声明一个常量变量arraySize，其值为5。常量变量(Constant variables)也被称为命名常量(named constants)或只读变量(read-only variables)。一个常量变量在声明时必须初始化，此后不能修改。尝试将arraySize初始化后进行修改，会出现错误。

6.3.4 数组元素的和

```
1 #include <iostream>
2 #include <array>
3 using namespace std;
4
5 int main() {
6     const size_t arraySize{4};
7     array<int, arraySize> a{10, 20, 30, 40};
8     int total{0};
9
10    for(size_t i{0}; i < a.size(); i++) {
11        total += a[i];
12    }
13
14    cout << "total of array elements is: " << total << endl;
15 }
```

上诉代码计算数组元素的和。

6.3.8 Static Local arrays and Automatic Local arrays

程序在首次遇到静态局部数组的声明时，对其进行初始化。如果你没有显式地初始化一个静态数组，那么在创建该数组时，编译器会将该数组的每个元素初始化为零。C++对其他局部变量不执行这样的默认初始化。

局部变量有时被称为自动变量，因为当函数执行完毕时，局部变量会被自动销毁。

```
1 #include <iostream>
2 #include <array>
3 using namespace std;
4
5 void staticArrayInit();
6 void automaticArrayInit();
7 const size_t arraySize{3};
8
9 int main() {
10     cout<<"first call to each function:\n";
11     staticArrayInit();
12     automaticArrayInit();
13
14     cout<<"\n\nSecond call to each function:\n";
15     staticArrayInit();
16     automaticArrayInit();
17     cout<<endl;
18 }
19
20 void staticArrayInit(){
21     static array<int, arraySize> array1;
22     cout<<"\nvalues on entering staticArrayInit:\n";
23
24     for(size_t i{0}; i<array1.size(); ++i){
25         cout<<"array1["<<i<<"] = "<<array1[i]<<" ";
26     }
27
28     cout<<"\nvalues on exiting staticArrayInit:\n";
29
30     for(size_t j{0}; j<array1.size(); ++j){
31         cout<<"array1["<<j<<"] = "<<(array1[j] += 5)<<" ";
32     }
33 }
```

```

33 }
34
35 void automaticArrayInit(){
36     array<int, arraySize> array2{1, 2, 3};
37     cout << "\n\nvalues on entering
automaticArrayInit:\n";
38
39     for (size_t i{0}; i < array2.size(); ++i) {
40         cout << "array2[" << i << "] = " << array2[i] << "
";
41     }
42     cout << "\nvalues on exiting automaticArrayInit:\n";
43
44     for (size_t j{0}; j < array2.size(); ++j) {
45         cout << "array2[" << j << "] = " << (array2[j] +=
5) << " ";
46     }
47 }

```

如前所述, static数组的值, 在函数执行一次后, 值不会销毁。

First call to each function:

```

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5

```

```

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

```

Second call to each function:

```

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10 array1[1] = 10 array1[2] = 10

```

```

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

```

6.4 基于范围的for循环(range based for statement)

如前所述，对数组中的所有元素进行处理是很常见的。基于范围的for循环允许您在不使用counter的情况下执行此操作，从而避免了"跳出"数组的可能性，也无需您执行自己的边界检查。

```
1 #include <iostream>
2 #include <array>
3 using namespace std;
4
5 int main() {
6     array<int,5> items{1,2,3,4,5};
7     cout<<"items before modification: ";
8
9     for(int item:items){
10         cout<<item<<" ";
11     }
12
13     for(int& itemRef: items){
14         itemRef*=2;
15     }
16
17     cout<<"\nitems after modification: ";
18     for(int item:items){
19         cout<<item<<" ";
20     }
21
22     cout<<endl;
23 }
```

```
1 for(int item:items){
2     cout<<item<<" ";
3 }
```

可以与如下的代码执行的功能一样：

```
for (int counter{0}; counter < items.size(); ++counter) {
    cout << items[counter] << " ";
```

基于范围的for语句简化了通过数组进行迭代的代码。上诉第二个代码块的for循环，可以讲是：对于每一次迭代，将items的下一个元素的值赋给int类型的变量item，然后执行循环。因此，对于每次迭代，item都代表items的一个值(不是下标)。在基于范围的for的函数头中，在冒号(:)的左边声明一个所谓的范围变量，并在右边指定一个数组的名称。**You can use the range-based for statement with most of the C++ Standard Library's prebuilt data structures (commonly called containers).**

当通过数组的代码在循环时不需要访问元素的下标时，可以使用基于范围的for语句代替counter控制的for语句。例如，计算数组中所有元素的和，就不需要数组下标。**但是，如果一个程序由于某种原因必须使用下标，而不是简单地通过数组(例如，在每个数组元素值的旁边打印一个下标数字,如本章前面的例子)循环，则应该使用counter进行for语句控制。**

6.6 数组的排序和查找

```
1 #include <iostream>
2 #include <iomanip>
3 #include <array>
4 #include <string>
5 #include <algorithm>
6 using namespace std;
7
8 int main() {
9     const size_t arraySize{5};
10    array<string, arraySize>
11    colors{"red", "orange", "yellow", "blue", "green"};
12    cout<<"unsorted array\n";
13    for(string color:colors){
14        cout<<color<<" ";
15    }
16    sort(colors.begin(), colors.end());
17    cout<<"sorted array:\n";
18    for(string item:colors){
19        cout<<item<<" ";
20    }
21    bool
22    found{binary_search(colors.begin(), colors.end(), "indigo")};
23
24    cout<<"\n\n\"indigo\" "<<(found? "was": "was not")
25    <<"found in colors"<<endl;
```

```
23
24     found=
25         binary_search(colors.begin(),colors.end(),"cyan");
26         cout<<"\cyan\ " <<(found?"was":"was not")
27         <<"found in colors"<<endl;
28 }
```

使用binary_search函数，首先必须对一组元素进行升值排序。函数的前两个实参表明寻找的范围，最后一个实参表示要寻找的元素。返回一个bool值指出是否寻找此元素。

6.7 多维数组

二维数组的声明：

```
1 | array<array<int,3>,3> a;
```

```
1 #include <iostream>
2 #include <array>
3 using namespace std;
4
5 const size_t rows{2};
6 const size_t columns{3};
7 void printArray(const array<array<int,columns>,rows>&);

8
9 int main(){
10     array<array<int,columns>,rows> array1{1,2,3,4,5,6};
11     array<array<int,columns>,rows> array2{1,2,3,4,};

12
13     cout<<"values in array1 by row are: "<<endl;
14     printArray(array1);
15
16     cout<<"\nvalues in array2 by row are: "<<endl;
17     printArray(array2);
18 }
19
20 void printArray(const array<array<int,columns>,rows>& a){
21     for(auto const& row:a){
22         for(auto const& element:row){
23             cout<<element<<' ';
```

```
24         }
25         cout<<endl;
26     }
27 }
```

为了处理二维数组的元素，我们使用了一个嵌套循环，其中外循环通过行进行迭代，内循环通过给定行的列进行迭代。如上诉代码21-22行所示。

上诉的嵌套循环代码可以替换为如下的形式：

```
for (size_t row{0}; row < a.size(); ++row) {
    for (size_t column{0}; column < a[row].size(); ++column) {
        cout << a[row][column] << ' ';
    }

    cout << endl;
}
total = 0;
for (size_t row{0}; row < a.size(); ++row) {
    for (size_t column{0}; column < a[row].size(); ++column) {
        total += a[row][column];
    }
}
```

上诉代码可以求得二维数组所有元素的总和。

二维数组的声明如下所示：

```
1 | array<array<int,columns>,rows> array1{1,2,3,4,5,6};
```

columns表示二维数组的列数，rows代表二维数组的行数。

对二维数组使用size函数，返回的是数组的行数。例如：a.size()返回数值2。

6.8 C++字符串类(string)

C++中使用string类处理字符串。操作string对象中的字符串时，有时会用到"index"。

创建string对象：

1.用无参构造函数创建一个空字符串

```
1 | string newString;
```

2.由一个字符串常量或字符串数组创建string对象

```

1 string message{"aloha world"};//  

2  

3 char charArr[]={'h','e','l','l','o'};  

4 string message1{charArr};//
```

string的成员函数可以参见std::basic_string - cppreference.com

String Operators (字符串运算符)

中国大学MOOC

```

string s1 = "ABC"; // The = operator
string s2 = s1; // The = operator
for (int i = s2.size() - 1; i >= 0; i--)
    cout << s2[i]; // The [] operator

string s3 = s1 + "DEFG"; // The + operator
cout << s3 << endl; // s3 becomes ABCDEFG

s1 += "ABC";
cout << s1 << endl; // s1 becomes ABCABC

s1 = "ABC";
s2 = "ABE";
cout << (s1 == s2) << endl; // Displays 0
cout << (s1 != s2) << endl; // Displays 1
cout << (s1 > s2) << endl; // Displays 0
cout << (s1 >= s2) << endl; // Displays 0
cout << (s1 < s2) << endl; // Displays 1
cout << (s1 <= s2) << endl; // Displays 1
```

Operator	Description
[]	用数组下标运算符访问字符串中的字符
=	将一个字符串的内容复制到另一个字符串
+	连接两个字符串得到一个新串
+=	将一个字符串追加到另一个字符串末尾
<<	将一个字符串插入一个流
>>	从一个流提取一个字符串，分界符为空格或者空结束符
==, !=, <, <=, >, >=	用于字符串比较

6.9 C++17 结构化绑定(structured binding)

结构化绑定声明是一个声明语句，意味着声明了一些标识符并对标识符做了初始化，在C++17标准中引入。

将指定的一些名字绑定到初始化器的子对象或者元素上。

```

1 cv-auto &/&& [标识符列表] = 表达式;
2 cv-auto &/&& [标识符列表] {表达式};
3 cv-auto &/&& [标识符列表] (表达式);
```

cv-auto:可能由const/volatile修饰的auto关键字

&/&&：左值引用或者右值引用

标识符列表：逗号分隔的标识符

6.9.1 用于原生数组的结构化绑定声明

若初始化表达式为数组类型，则标识符列表中的名字绑定到数组元素。

1. 标识符数量必须等于数组元素数量

2. 标识符类型与数组元素类型一致

```
1 int main(){
2     int priArr[] {42, 21, 7};
3
4     auto [a1, a2, a3]=priArr;//a1是priArr[0]的拷贝, a2, a3类推
5     const auto [b1, b2, b3] (priArr); //b1是priArr[0]的只读拷
6     //贝, b2, b3类推
7     auto &[c1, c2, c3] {priArr}; //c1是priArr[0]的引用, c2, c3类
8     //推
9     c3=14;//priArr[2]的值变为14
10    return 0;
11 }
```

6.9.2 用于std::array的结构化绑定声明

若初始化表达式为数组类型，则标识符列表中的名字绑定到数组元素。

1. 标识符数量必须等于array中的元素数量

2. 标识符类型与array中的元素类型一致

```
1 int main(){
2     std::array stdArr={'a', 'b', 'c'};//C++17
3     auto [d1, d2, d3]{stdArr};
4     return 0;
5 }
```

6.9.3 用于对象数据成员的结构化绑定

若初始化表达式为类/结构体类型，则标识符列表中的名字绑定到类/结构体的**非静态数据成员**上

1. 数据成员必须为公有成员
2. 标识符数量必须等于数据成员的数量
3. 标识符类型与数据成员类型一致

```
1 class C { // 可以改用 struct C, 然后去掉下面的public属性说明
2 public:
3     int i { 420 }; // 就地初始化
4     char ca[ 3 ] { 'O', 'K', '!' };
5 };
6
7 int main() {
8     C c;
9     auto [a1, a2] {c}; // a1是int类型, a2是char[]类型
10    std::cout << "c.i:" << a1 << " c.ca:" << b2 <<
11    std::endl;
12 }
```

auto后跟&, 则标识符是数据成员的引用

auto前可放置const, 表明标识符是只读的

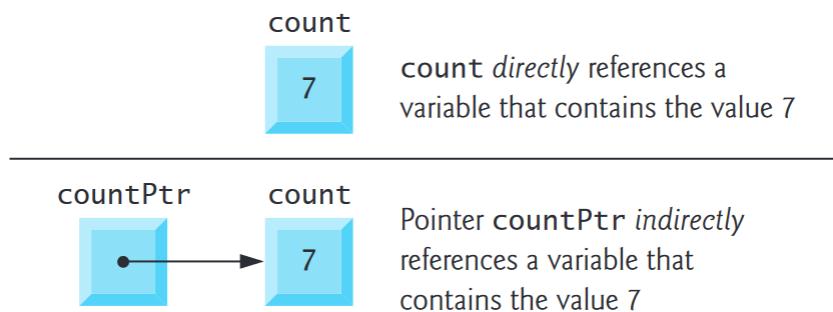
```
1 int main() {
2     C c; // c.i: 420; c.ca: 'O', 'K', '!'
3     auto [a1, a2] {c}; // a1是c.i的拷贝, a2是char[]类型
4     auto& [b1, b2] {c}; // b1是int&类型, 是c.i的引用,
5     // b2是char(&)[3]类型(数组的引用), 是c.ca的引用
6     a1 = 100;
7     std::cout << "c.i:" << c.i << std::endl; // 输出420, 改a1
8     不影响c.i
9     b1 = 200;
10    std::cout << "c.i:" << c.i << std::endl; // 输出200, 通过
11    b1修改了c.i
12 }
```

7.指针

指针也允许pass-by-reference, 并且可以用来创造和操作动态数据结构, 例如列表、队列、栈和树(lists, queues, stacks and trees)。

7.1 指针变量的声明和初始化

指针变量包含内存地址作为其值，**指针包含变量的内存地址，而变量的内存地址又包含一个特定的值。在这个意义上，变量名直接引用一个值，指针间接引用一个值。通过指针引用一个值称为间接引用。** 例如下图指针变量的间接引用以及 count 的直接引用。



7.2 声明指针

指针就像其他变量一样，在使用之前必须先声明，例如上图中的指针，可以声明为：

```
1 | int* countPtr, count;
```

指针可以被声明指向任何数据类型的对象。

Good Programming Practice 8.1

Although it's not a requirement, we like to include the letters `Ptr` in each pointer variable name to make it clear that the variable is a pointer and must be handled accordingly.

上诉代码，指针的类型为 `int*`, 指针变量的名字为 `countPtr`, 可以存放一个 `int` 类型数据的地址。

The declaration

```
int* ptr;
```

declares `ptr` to be a pointer to a variable of type `int` and is read, “`ptr` is a pointer to `int`.” The `*` as used here in a declaration indicates that the variable is a pointer.

`void`类型的指针只能用来存放地址，不能用此指针访问内存空间。

7.2.1 初始化指针

无论是在声明时还是在赋值时，都需要将指针初始化为nullptr或内存地址。具有nullptr的指针"point to nothing", 被称为空指针(null pointer). **初始化所有指针，防止指向未知或未初始化的内存区域。** 将指针初始化为NULL或者0是相同的操作，0是唯一的可以直接赋值给指针的整数，而不用事先将整数类型转化为指针类型(一般使用reinterpret_cast)

7.3 指针操作符

7.3.1 取地址操作符(&)

&是一元操作符，用来取得操作数的地址，例如下列代码：

```
1 int y{5}; // declare variable y
2 int* yPtr=nullptr; // declare pointer variable yPtr
3 yPtr = &y; // assign address of y to yPt
```

上述代码将变量y的内存地址通过&取得，并赋值给yPtr,现在，指针yPtr间接的引用了变量y的值5。地址运算符不能应用于文字，也不能应用于导致临时值(像计算结果一样)的表达式。

7.3.2 解引用操作符(*)

返回一个左值，表示其指针操作数指向的对象。例如对于上述代码，下列的两段代码的结果是相同的：

```
1 cout<<*yPtr<<endl;
```

```
1 cout<<y<<endl;
```

Using * in this manner is called dereferencing a pointer.还可以作为左值被赋值：

```
1 *yPtr=9;
```

还可以进行如下操作。

```
cin >> *yPtr;
```



Common Programming Error 8.2

Dereferencing an uninitialized pointer results in undefined behavior that could cause a fatal execution-time error. This could also lead to accidentally modifying important data, allowing the program to run to completion, possibly with incorrect results.



Error-Prevention Tip 8.2

Dereferencing a null pointer results in undefined behavior and typically causes a fatal execution-time error. Ensure that a pointer is not null before dereferencing it.

7.3.3 使用&和*

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int a{7};
5     int* aPtr=&a;
6
7     cout<<"the address of a is:"<<&a<<"\n the value of aPtr
8 is: "<<aPtr;
9     cout<<"\n\nthe value of a is: "<<a<<"\nthe value of
10 *aPtr is:"<<*aPtr<<endl;
11 }
```

7.3.4 二级指针

指针(指针变量的简称)用于存放普通变量的地址, 二级指针用于存放指针变量的地址。二级指针的声明格式如下:

```
1 | 数据类型** 指针名
```

使用指针的目的: 1.传递地址 2.存放动态分配内存的地址

在函数中, 如果传递普通变量的地址, 形参用指针; 传递指针变量的地址, 形参用二级指针。

7.3.5 空指针

用0或者NULL都可以表示空指针。声明指针后, 在赋值前让指针指向空, 表示没有指向任何地址。C++11建议用nullptr表示空指针, 也就是(void*)0。

1.如果对空指针解引用, 程序会崩溃。

2.在函数中, 应该有判断参数是否为空指针的代码, 目的是保证程序的健壮性。

出现野指针的情况有三种:

1. 指针在定义时，如果没有进行初始化，它的值是不确定的。
2. 如果用指针指向了动态分配的内存，内存被释放以后，指针不会置空，但是，指向的地址已经失效。
3. 指针指向的变量已经超过变量的作用域(变量的内存空间已经被收回)

避免方法：

1. 指针在定义时，如果没有可以指向的地址，就初始化为nullptr。
2. 动态分配的内存释放以后，将其置为nullptr。
3. 函数不要返回局部变量的地址。

7.3.6 野指针

野指针即指针指向的不是一个有效(合法)的地址。在程序中访问野指针可能会造成程序的崩溃。

7.4 函数的参数传递

在C++中有三种方式将实参传递给函数：

1. **pass-by-value.(传值)**

2. **pass-by-reference with a reference argument.(传引用)**

3. **pass-by-reference with a pointer argument.(传地址)**

指针和引用一样，也可以用来修改调用者中的变量或者通过引用传递大数据对象，避免复制对象的开销。

Comparision: 3 swap() functions

```
//pass by value
void swap(int x, int y){
    int t;
    t=x; x=y; y=t;
}
int main() {
    int a=5, b{10};
    cout << "Before: a=" << a <<
        " b=" << b << endl;
    swap( a, b );
    cout << "After: a=" << a <<
        "b=" << b << endl;
    return 0;
}
```

Before: a=5 b=10
After: a=5 b=10

```
//pass by pointer
void swap(int* x, int* y){
    int t;
    t=*x; *x=*y; *y=t;
}
int main() {
    auto a{5}, b{10};
    cout << "Before: a=" << a <<
        " b=" << b << endl;
    swap( &a, &b );
    cout << "After: a=" << a <<
        "<<b=<<b<<endl;
    return 0;
}
```

Before: a=5 b=10
After: a=10 b=5

```
//pass by reference
void swap(int& x, int& y){
    int t;
    t=x; x=y; y=t;
}
int main() {
    auto a{5}, b{10};
    cout << "Before: a=" << a <<
        " b=" << b << endl;
    swap( a, b );
    cout << "After: a=" << a <<
        "b=" << b << endl;
    return 0;
}
```

Before: a=5 b=10
After: a=10 b=5

7.4.1 Pass-By-Reference with a Pointer Actually Passes the Pointer By Value

用指针通过引用传递一个变量实际上没有通过引用传递任何东西(一个指向该变量的指针被值传递并复制到函数对应的指针参数中)。被调用的函数只需解引用指针就可以访问调用者的该变量，从而完成pass-by-reference。

7.4.2 三种传递方式的使用原则

1) 不需要在函数中修改实参

1. 如果实参很小，如C++内置的数据类型或小型结构体，则按值传递

2. 如果实参是数组，则使用const指针，这是唯一的选择(无法为数组建立引用)

3. 如果实参是较大的结构体，则使用const指针或const引用

4. 数据实参是类，则使用const引用，传递类的标准方式是按引用传递

2) 需要在函数中修改实参

1. 如果实参是内置数据类型，则使用指针

2. 如果实参是数组，则只能使用指针

3. 如果实参是结构体，则使用指针或引用

4. 如果实参是类，使用引用

7.5 built-in arrays(内置数组)

在前面的内容，介绍了array，一种固定大小的元素的列表。现在介绍另一种固定大小的数据结构built-in arrays。

7.5.1 声明和访问built-in arrays

```
type arrayName[arraySize];
```

声明的格式如上，arraysize必须是大于零的整数常量。 []不会提供边界检查的功能。

7.5.2 初始化built-in arrays

```
int n[5]{50, 20, 30, 10, 40};
```

如果提供比元素个数更少的初始化值，剩下的元素都是值初始化的-基本数值类型设置为0，bools设置为false，指针设置为nullptr，类对象由它们的默认构造函数初始化。如果提供过多的初始化值，就会出现编译错误。

如果从带有初始化值列表的声明中省略了built-in数组的大小，则编译器将built-in数组大小调整为初始化器列表中的元素个数。

数组在内存中占用的空间是连续的。

7.5.3 将数组作为参数传递给函数

数组的名字就是数组第一个元素的地址。所以arrayName就等同于`&arrayName[0]`。因此就不需要使用`&`获取数组的地址从而传递给函数，可以直接传递数组的名字。

正如在之前中所看到的那样，接收到调用者中变量的指针的函数可以修改调用者中的变量。对于内置数组，这意味着被调用的函数可以修改调用者内置数组的所有元素- -除非该函数在对应的内置数组参数前加上`const`表示不应该修改元素。

7.5.4 声明内置数组形参

```
int sumElements(const int values[], const size_t numberElements)
```

可以使用上述代码在函数头中声明内置数组。与array对象不一样，内置数组不知道自身的大小，所以函数的形参列表应该同时包括内置数组以及数组的大小。

一维内置数组用于函数的参数时，只能传递数组的地址，并且要把数组的长度也传递进去，除非数组中有最后一个元素的标志。

上述代码还可以写为如下形式：

```
int sumElements(const int* values, const size_t numberElements)
```

当声明一个内置数组形参时，为了清晰起见，使用`[]`符号而不是指针符号。

7.5.5 C++11: Standard Library Functions begin and end

sort函数也可以用在内置数组里，例如：

```
sort(begin(n), end(n)); // sort contents of built-in array n
```

begin和end函数的头文件为<iterator>。都将内置数组作为实参，并返回一个指针，该指针可用于表示C++标准库函数中需要处理的元素范围，如sort函数。

7.5.6 内置数组的限制

内置数组有几个限制：

- 1.They cannot be compared using the relational and equality operators—you must use a loop to compare two built-in arrays element by element.
- 2.They cannot be assigned to one another—an array name is effectively a pointer that is const.
- 3.They don't know their own size—a function that processes a built-in array typically receives both the built-in array's name and its size as arguments.
- 4.They don't provide automatic bounds checking—you must ensure that array-access expressions use subscripts that are within the built-in array's bounds.

7.5.7 Built-In Arrays Sometimes Are Required

一些情况下，必须使用内置数组，例如处理程序的command-line arguments。

In contemporary C++ code, you should use the more robust array (or vector) objects to represent lists and tables of values. However, there are cases in which built-in arrays *must* be used, such as processing a program's [command-line arguments](#). You supply these to a program by placing them after the program's name when executing it from the command line. Such arguments typically pass options to a program. For example, on a Windows system, the command

```
dir /p
```

uses the /p argument to list the contents of the current directory, pausing after each screen of information. Similarly, on Linux or OS X, the following command uses the -la argument to list the contents of the current directory with details about each file and directory:

```
ls -la
```

Command-line arguments are passed to `main` as a built-in array of pointer-based strings (Section 8.10)—C++ inherited these capabilities from C. Appendix F shows how to process command-line arguments.

7.5.6 清空内置数组

用`memset()`函数可以将内置数组的全部元素清零。(只适用于C++基本数据类型)

C 库函数 **void *memset(void *str, int c, size_t n)**** 复制字符 **c** (一个无符号字符) 到参数 **str** 所指向的字符串的前 **n** 个字符。

此函数的原型：

```
1 void *memset(void *str, int c, size_t n)
2
3 //str -- 指向要填充的内存块。
4 //c -- 要被设置的值。该值以 int 形式传递，但是函数在填充内存块时是
5 //n -- 要被设置为该值的字符数。
```

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main ()
5 {
6     char str[50];
7
8     strcpy(str, "This is string.h library function");
9     puts(str);
10
11     memset(str, '$', 7);
12     puts(str);
13
14     return(0);
15 }
```

```
D:\clionproject\cpp_project\test_11_27\cmake-build-debug\test_11_27.exe
This is string.h library function
$$$$$$ string.h library function

进程已结束，退出代码为 0
```

7.5.7 复制内置数组

C 库函数 **void *memcpy(void *str1, const void *str2, size_t n)**** 从存储区 **str2** 复制 **n** 个字节到存储区 **str1**。

下面是 **memcpy()** 函数的声明：

```
1 void *memcpy(void *str1, const void *str2, size_t n)
```

- **str1** -- 指向用于存储复制内容的目标数组，类型强制转换为 **void*** 指针。
- **str2** -- 指向要复制的数据源，类型强制转换为 **void*** 指针。
- **n** -- 要被复制的字节数。

```

1 // 将字符串复制到数组 dest 中
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     const char src[50] = "http://www.runoob.com";
8     char dest[50];
9
10    memcpy(dest, src, strlen(src)+1);
11    printf("dest = %s\n", dest);
12
13    return(0);
14 }

```

```

D:\clionproject\cpp_project\test_11_27\cmake-build-debug\test_11_27.exe
dest = https://www.runoob.com

进程已结束，退出代码为 0

```

7.6 指针和const

Many possibilities exist for using (or not using) const with function parameters, so how do you choose the most appropriate? 应该使用最小特权原则，总是赋予一个函数足够的访问其参数中的数据的权限来完成其指定的任务，但not more。

这一节就是讨论如何使用const和指针实现上述的最小特权原则。



Software Engineering Observation 8.3

If a value does not (or should not) change in the body of a function to which it's passed, the parameter should be declared const.

有四种方式将指针传递给函数：

- 1.a nonconstant pointer to nonconstant data,
- 2.a nonconstant pointer to constant data
- 3.a constant pointer to nonconstant data
- 4.a constant pointer to constant data

7.6.1 nonconstant pointer to nonconstant data

最高的访问权限是由非const指针授予非const数据：

1.数据可以通过解引用指针进行修改

2.指针可以修改指向其他数据

7.6.2 nonconstant pointer to constant data(常量指针)

常量指针就是指向常量值的一个指针变量：

1.一个指针可以被修改为可以指向任意合适类型的数据。

2.指针所指向的数据不能通过该指针进行修改(即，不能通过解引用的方法修改)。

这种类型的指针可以声明为： `const int * countPtr` 即countPtr是一个nonconstant pointer指向constant data

```
1 #include <iostream>
2 void f(const int*)
3
4 int main(){
5     int y{0};
6     f(&y);
7 }
8 void f(const int* xPtr){
9     *xPtr=100;
10 }
```

```
1 const int x=1;
2 const int* p1;
3 p1=&x;
4
5 *p1=10; //错误!
6
7 char* s1="hello"; //错误!
8 const char* s2="hello"; //正确!
```

上述代码试图通过指针修改const类型的数据，就会出现错误。

一般用于修饰函数的形参，表示不希望在函数里修改内存地址中的值。

当一个函数被调用，且用内置数组作为实参，它(内置数组)的内容被有效的pass-by-reference。因为内置数组的名字被隐式的转换为数组第一个元素的地址。但是在默认情况下，array和vector对象是pass-by-value(整个对象的副本被传递)。对对象中的每个数据项做副本并存储在函数调用栈上需要消耗执行时间。当对象的指针被传递，只有对象地址的副本必须传递，而对象本身不需要。

如果数据不需要被被调用函数修改，使用constant data的指针或者引用传递大型的对象，可以减少使用pass-by-value使用副本带来的开销。还可以提供数据数值传递的安全性。

7.6.3 Constant Pointer to Nonconstant Data(指针常量)

A constant pointer to nonconstant data is a pointer that:

1.永远指向相同的内存地址

2.该内存地址的数据可以通过指针修改

指针在声明为const时，必须初始化，但是如果指针是函数的形参，则使用传递给函数的指针对其进行初始化。

```
1 // Fig. 8.11: fig08_11.cpp
2 // Attempting to modify a constant pointer to nonconstant data.
3
4 int main() {
5     int x, y;
6
7     // ptr is a constant pointer to an integer that can be modified
8     // through ptr, but ptr always points to the same memory location.
9     int* const ptr{&x}; // const pointer must be initialized
10
11    *ptr = 7; // allowed: *ptr is not const
12    ptr = &y; // error: ptr is const; cannot assign to it a new address
13 }
```

Microsoft Visual C++ compiler error message:

```
'ptr': you cannot assign to a variable that is const
```

```
1 int x=1,y=1;
2 int* const p2=&x;
3
4 p2=10; //correct! x=10
5 p2=&y; //error! 指针p2是constant的
```

数组名就是数组的首地址的别名。可以说数组名就是一个指针常量。

7.6.4 Constant Pointer to Constant Data

最小访问权限由const指针授予const数据：

- 1.指针永远指向相同的内存地址
- 2.该内存地址的数据不能通过指针修改

这就是一个内置数组应该如何传递给一个只从数组中读取的函数，使用数组下标符号，而不对其进行修改。

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int x{5},y;
6     const int* const ptr{&x};
7
8     cout<<*ptr<<endl;
9     *ptr=7;
10    ptr=&y
11 }
```

上述代码试图修改const指针和const数据，就会出现错误。

7.7 sizeof操作符

当对内置数组使用sizeof操作符以后，sizeof会返回一个size_t类型的内置数组的总字节数。当在函数中使用指针作为形参，接收内置数组作为实参，sizeof操作符返回的是指针的字节数，而不是内置数组的总字节数。

```
1 #include <iostream>
2 using namespace std;
3 size_t getSize(double* );
4
5 int main() {
6     double numbers[20];
7     cout<<"the number of bytes in the array is "
8     <<sizeof(numbers);
9
10    cout<<"\nthe number of bytes returned by getSize is "
11    <<getSize(numbers)<<endl;
12 }
```

```
11
12 size_t getSize(double* ptr){
13     return sizeof(ptr);
14 }
```

计算内置数组含有多少个元素可以使用如下命令：

```
1 sizeof numbers / sizeof(numbers[0])
```

下列程序使用sizeof操作符判断基本数据类型、内置数组和指针的字节数

```
1 // Fig. 8.14: fig08_14.cpp
2 // sizeof operator used to determine standard data type sizes.
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     char c; // variable of type char
8     short s; // variable of type short
9     int i; // variable of type int
10    long l; // variable of type long
11    long long ll; // variable of type long long
12    float f; // variable of type float
13    double d; // variable of type double
14    long double ld; // variable of type long double
15    int array[20]; // built-in array of int
16    int* ptr{array}; // variable of type int *
```

```

17
18     cout << "sizeof c = " << sizeof c
19     << "\tsizeof(char) = " << sizeof(char)
20     << "\nsizeof s = " << sizeof s
21     << "\tsizeof(short) = " << sizeof(short)
22     << "\nsizeof i = " << sizeof i
23     << "\tsizeof(int) = " << sizeof(int)
24     << "\nsizeof l = " << sizeof l
25     << "\tsizeof(long) = " << sizeof(long)
26     << "\nsizeof ll = " << sizeof ll
27     << "\tsizeof(long long) = " << sizeof(long long)
28     << "\nsizeof f = " << sizeof f
29     << "\tsizeof(float) = " << sizeof(float)
30     << "\nsizeof d = " << sizeof d
31     << "\tsizeof(double) = " << sizeof(double)
32     << "\nsizeof ld = " << sizeof ld
33     << "\tsizeof(long double) = " << sizeof(long double)
34     << "\nsizeof array = " << sizeof array
35     << "\nsizeof ptr = " << sizeof ptr << endl;
36 }

```

```

sizeof c = 1    sizeof(char) = 1
sizeof s = 2    sizeof(short) = 2
sizeof i = 4    sizeof(int) = 4
sizeof l = 8    sizeof(long) = 8
sizeof ll = 8   sizeof(long long) = 8
sizeof f = 4    sizeof(float) = 4
sizeof d = 8    sizeof(double) = 8
sizeof ld = 16   sizeof(long double) = 16
sizeof array = 80
sizeof ptr = 8

```



Portability Tip 8.2

The number of bytes used to store a particular data type may vary among systems. When writing programs that depend on data type sizes, always use sizeof to determine the number of bytes used to store the data types.

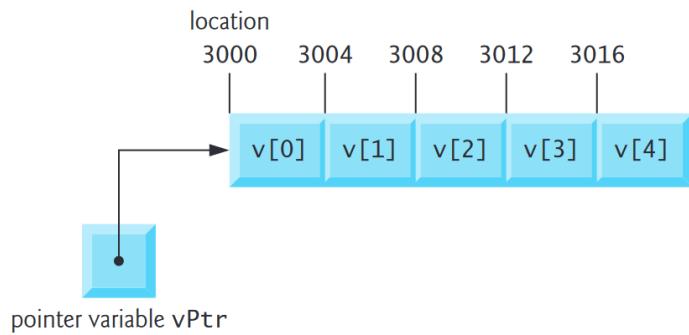
当sizeof的操作数是一个表达式时，与sizeof一起使用的括号是不需要的。记住sizeof是一个编译时(compile-time)操作符，所以它的操作数在运行时不被评估。

7.8 指针表达式及指针算数运算

指针的算数运算只适用于指向内置数组的指针。

一个指针可以被递增(`++`)或递减(`--`)，一个整数可以被加到一个指针(`+或+=`)或从一个指针(`-或-=`)中减去，或者一个指针可以从另一个相同类型的指针中减去`-`这种特殊的操作只适用于指向同一个内置数组的两个指针。

```
int* vPtr{v};  
int* vPtr{&v[0]};
```



7.8.1 对指针进行加减整数操作

对指针加上整数，并不是直接将此整数加到指针的值上，例如，对于上述的指针 vPtr，原始值为3000，命令 $3000+2$ ，vPtr的值并不是3002，加上的整数还要乘以指针指向的对象的字节数，即 $vPtr=3000+(2*4)=3008$ (认定指针指向的内存对象的大小为4个字节)。

7.8.2 指针相减

指向同一内置数组的指针变量可以相互减去。

例如，指针变量v1和v2分别包含地址3008和3000，

```
1 | x=v2-v1;
```

会将v2和v1之间的元素个数的值赋给变量x， $x=2$ 。 **Pointer arithmetic is meaningful only on a pointer that points to a built-in array.** 我们不能假设同一类型的两个变量在内存中连续存储，除非它们是内置数组的相邻元素。

7.8.3 指针赋值

如果两个指针的类型相同，就可以将一个指针赋给另一个指针。否则，就必须使用cast operator(一般是reinterpret_cast)，将赋值语句的右边的指针的类型转换为左侧指针的类型。该规则的例外是指向void (即, void)的指针，它是一个通用指针，能够表示任何指针类型。

说可以用任意类型的指针对 void 指针对 void 指针赋值,不需要类型转换。如果要将 void 指针赋给其他类型的指针，则需要强制类型转换。

7.8.4 Cannot Dereference a void*

一个void*的指针不能进行解引用操作。因为void*的指针指向的是未知类型的内存地址。



Common Programming Error 8.6

The allowed operations on void pointers are: comparing void* pointers with other pointers, casting void* pointers to other pointer types and assigning addresses to void* pointers. All other operations on void* pointers are compilation errors.*

7.8.5 指针相比较

指针可以使用等式和关系运算符进行比较。使用关系运算符比较指针是无意义的，除非两个指针指向的是同一个内置数组。指针之间的比较，比较的是指针存储的地址。A common use of pointer comparison is determining whether a pointer has the value nullptr, 0 or NULL (i.e., the pointer does not point to anything).

7.9 指针和内置数组的关系

指针可以用来做任何涉及数组下标的操作。

考虑如下的声明：

```
1 | int b[ 5 ]; // create 5-element int array b; b is a const
   | pointer
2 | int* bPtr; // create int pointer bPtr, which isn't a const
   | pointer
```

可以将数组的第一个元素的地址赋给指针变量

```
1 | bPtr=b;
```

上诉语句和下述代码效果相同

```
1 | bPtr=&b[0];
```

而

```
1 | b+=3;
```

会造成编译错误，因为它试图通过指针数运算修改内置数组的名字。

7.9.1 Pointer/Offset Notation

还是考虑上述的代码，内置数组的第三个元素，即b[3],可以用指针表示为：

```
1 | *(bPtr + 3)
```

3代表指针的偏移量。当指针指向一个内置数组的起始元素时，偏移量表示应该引用哪个内置数组元素，且偏移量的值与下标相同。这种表示法被称为 Pointer/Offset Notation。括号是必要的，因为括号的优先级高于+。在没有括号的情况下，前面的表达式会在* bPtr的值(即在b 中加入3 ,假设bPtr指向内置数组的开始)的副本中加上3。

正如内置数组的元素可以用指针表达式引用一样，地址：

```
1 | &b[3]
```

可以写为：

```
1 | bPtr + 3
```

7.9.2 Pointer/Offset Notation with the Built-In Array's Name as the Pointer

内置数组名可以作为指针处理，用于指针算术运算。例如

```
1 | *(b + 3)
```

也表示数组的第三个元素。

一般情况下，所有带下标的内置数组表达式都可以用一个指针和一个偏移量写出。在这种情况下，使用指针/偏移量表示法，内置数组的名称作为指针。前面的表达式不修改内置数组的名称；b仍然指向内置数组的第1个元素。

7.9.3 Pointer/Subscript Notation

指针可以带下标正如内置数组一样。例如，表达式

```
1 | bPtr[1]
```

refers to b[1]; this expression uses pointer/subscript notation.

7.9.4 Demonstrating the Relationship Between Pointers and Built-In Arrays

四种表示方法：

1.array subscript notation

2.pointer/offset notation with the built-in array's name as a pointer

3.pointer subscript notation

4.pointer/offset notation with a pointer

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int b[]{10,20,30,40};
6     int* bPtr{b};
7 // output built-in array b using array subscript notation
8     cout<<"array b displayed with:\n\narray subscript
9 notation\n";
10
11     for(size_t i{0};i<4;++i){
12         cout<<"b["<<i<<"]="<<b[i]<<'\n';
13     }
14 // output built-in array b using array name and
15 // pointer/offset notation
16     cout<<"\npointer/offset notation where"<<"the pointer
17 is the array name\n";
18
19     for(size_t offset1{0};offset1<4;+offset1){
20         cout<<"*(b+"<<offset1<<")="<<*(b+offset1)<<'\n';
21     }
22 // output built-in array b using bPtr and array subscript
23 // notation
24     cout<<"\npointer subscript notation\n";
25
26     for(size_t j{0};j<4;++j){
27         cout<<"bPtr["<<j<<"]="<<bPtr[j]<<'\n';
28     }
29 // output built-in array b using bPtr and pointer/offset
30 // notation
```

```
26     cout<<"\npointer/offset notation\n";
27
28     for(size_t offset2{0};offset2<4;++offset2){
29         cout<<"*(bPtr+"<<offset2<<")="<<*(bPtr+offset2)
30         <<'\n';
31     }
```

运行结果为：

```
array subscript notation
b[0]=10
b[1]=20
b[2]=30
b[3]=40

pointer/offset notation wherethe pointer is the array name
*(b+0)=10
*(b+1)=20
*(b+2)=30
*(b+3)=40

pointer subscript notation
bPtr[0]=10
bPtr[1]=20
bPtr[2]=30
bPtr[3]=40

pointer/offset notation
*(bPtr+0)=10
*(bPtr+1)=20
*(bPtr+2)=30
*(bPtr+3)=40
```

7.10 基于指针的字符串(Pointer-Based Strings)

字符和字符常量：

字符是C++源程序的基本构件。一个程序可能包含字符常量。字符常量是在单引号中表示为字符的整数值。

字符串：

字符串是作为单个单位处理的一系列字符。 一个字符串可能包括字母、数字和各种特殊字符，如+、-、*、/和\$。C++中的字符串字面量(String literals)，或字符串常量，用双引号写成如下：

"John Q. Doe"	(a name)
"9999 Main Street"	(a street address)
"Maynard, Massachusetts"	(a city and state)
"(201) 555-1212"	(a telephone number)

基于指针的字符串：

基于指针的字符串是一个存储字符的内置数组，并以空字符(null character('\0'))结尾，其标记了字符串在内存中终止的位置。字符串通过指针访问其第一个字符。字符串文本sizeof的结果是包含终止空字符的长度。

字符串字面量作为初始化值：

The declaration `char color[] = "blue";` could also be written

```
char color[]{'b', 'l', 'u', 'e', '\0'};
```

当声明内置字符串数组包含字符串时，内置数组必须足够大，以存储字符串及其终止空字符。

没有在一个内置的字符数组中分配足够的空间来存储终止字符串的空字符是一个逻辑错误。

创建或使用不包含终止空字符的C字符串会导致逻辑错误。

在内置字符数组中存储字符串时，确保内置数组足够大，以容纳将要存储的最大字符串。C++允许任意长度的字符串。如果一个字符串比待存储字符串的内置数组长，那么内置数组末尾以外的字符将跟随内置数组覆盖内存中的数据，从而导致逻辑错误和潜在的安全漏洞。

访问C字符串中的字符

由于C字符串是一个内置的字符数组，我们可以直接用数组下标记法访问字符串中的单个字符。例如，在前面的声明中，`color[0]`是字符' b '，`color[2]`是字符' u '，`color[4]`是空字符。

用cin读取字符串到char内置数组中

一个字符串可以使用`cin`读入到一个内置的`char`类型数组中，例如，下面的语句读取一个字符串到内置的20个元素`char`数组中：

```
1 | cin>>word;
```

用户输入的字符串存储在word数组中。上述语句读取字符，直到遇到空白字符或EOF字符才会停止。字符串不应超过19个字符，为终止空字符留有余地。**使用setw流操作符可以保证读入word的字符串不超过内置数组的大小。** 例如：

```
1 | cin>>setw(20)>>word;
```

规定cin最多读取19个字符到word，并保留第20个位置，用于存储字符串的终止空字符。The setw stream manipulator is not a sticky setting—it applies only to the next value being input.如果输入了19个以上的字符，剩下的字符虽然没有保存在word中，但是会在输入流中，可以被下一次的输入操作读取。

EOF(end of file)就是文件的结束，通常来判断文件的操作是否结束的标志。

EOF不是特殊字符，而是定义在头文件<stdio.h>的常量，一般等于-1；

在微软的DOS和Windows中，读取数据时终端不会产生EOF。此时，应用程序知道数据源是一个终端（或者其它“字符设备”），并将一个已知的保留的字符或序列解释为文件结束的指明；最普遍地说，它是ASCII码中的替换字符（Control-Z，代码26）。

在C语言中，或更精确地说成C标准函数库中表示文件结束符（end of file）。在while循环中以EOF作为文件结束标志，这种以EOF作为文件结束标志的文件，必须是文本文件。在文本文件中，数据都是以字符的ASCII代码值的形式存放。我们知道，ASCII代码值的范围是0~127，不可能出现-1，因此可以用EOF作为文件结束标志。

档案存取或其它I/O功能可能传回等于象征符号值（巨集）EOF指示档案结束的情形发生。实际上EOF的值通常为-1，但它依系统有所不同。巨集EOF会在编译原始码前展开实际值给预处理器。

C语言中，EOF常被作为文件结束的标志。还有很多文件处理函数处错误后的返回值也是EOF，因此常被用来判断调用一个函数是否成功。

Reading Lines of Text into Built-In Arrays of char with cin.getline

在某些情况下，希望将整行文本输入到一个内置的字符数组中。为此，cin对象提供成员函数getline，它包含三个参数- -一个内置的用于存储文本行的char数组、一个长度和一个分隔符。

A delimiter is one or more characters that separate text strings. Common delimiters are **commas** (,), semicolon (;), quotes (" , '), braces ({}), pipes (|), or slashes (/). When a program stores sequential or tabular data, it delimits each item of data with a predefined character.

```
char sentence[80];
cin.getline(sentence, 80, '\n');
```

函数在遇到分隔符'\n'时停止读取字符，在输入EOF符时或者输入的字符数量比函数的第二个参数少1时停止读取字符。内置数组中的最后一个字符被保留为终止空字符。如果最后一个字符遇到'\n',会读取后将其丢弃。函数的第三个参数是默认的，所以上述代码可以写为：

```
1 | cin.getline(sentence, 80);
```

使用cout可以将char数组的内容输出：

```
1 | cout<<sentence;
```

像cin一样，cout并不关心内置char数组有多大。字符被输出，直到遇到一个终止的空字符；**空字符不显示**。 [注: cin和cout假设char的内置数组应该被处理成以空字符结尾的字符串。cin和cout没有为其他内置数组类型提供类似的输入输出处理能力]

7.11 Smart Pointers(智能指针)

带指针的动态内存管理，它允许你在执行时根据需要创建和销毁对象。对这一过程的不当管理是导致细微错误的根源。我们将讨论"智能指针"，它通过提供内置指针之外的附加功能，帮助您避免动态内存管理错误。

提供了三种智能指针：

1.unique_ptr:不允许多个指针共享资源，可以用标准库中的move函数转移指针

2.shared_ptr:多个指针共享资源

3.weak_ptr:可以复制shared_ptr，但其构造或者释放对资源不产生影响

7.12 左值、纯右值和将亡值

区分左值右值的真正说法是：能否用“取地址&”运算符获得对象的内存地址。

对于临时对象，它可以存储于寄存器中，所以是没办法用“取地址&”运算符；

对于常量，它可能被编码到机器指令的“立即数”中，所以是没办法用“取地址&”运算符；

这也是C/C++标准的规定。

字符串字面值常量是C++标准中明确指明的特例，为常量左值，所以可以取地址&运算，其地址属于进程的只读内存空间。其它的字面值常量都是“纯右值”

左值：例如变量、数组元素、结构体成员、引用和解引用的指针

1.左值

指定了一个函数或者对象，它是一个可以取地址的表达式

```
1 int lval{ 42 }; // Object
2 int main() {
3     int& lval2{ lval }; // Lvalue reference to Object
4     int* lval3{ &lval }; // Pointer to Object
5 }
6
7 int& lval4() { return lval; } // Function returning Lvalue
Reference
```

1.1左值例子

- (1) 解引用表达式*p
- (2) 字符串字面量"abc"
- (3) 前置自增/自减表达式 ++i / --i
- (4) 赋值或复合运算符表达式(x=y或m*=n等)

2.C++11: 纯右值

是不和对象相关联的值(字面量)或者其求值结果是字面量或者一个匿名的临时对象

2.1纯右值例子

- (1) 除字符串字面量以外的字面量，比如 32, 'a'
- (2) 返回非引用类型的函数调用 int f() { return 1;}
- (3) 后置自增/自减表达式i++/i--

(4) 算术/逻辑/关系表达式 (a+b、 a&b、 a<<b) (a&&b、 a||b、 ~a)

(a==b、 a>=b、 a<b)

(5) 取地址 (&x)

左值可以当成右值使用

3.C++11: 将亡值

xvalue(expiring Value, 将亡值): 将亡值也指定了一个对象, 是一个将纯右值转换为右值引用的表达式

```
1 int prv(int x) { return 6 * x; } // pure rvalue
2
3 int main() {
4     const int& lvr5{ 21 }; // 常量左值引用可引用纯右值
5     int& lvr6{ 22 }; // 错! 非常量左值引用不可引用纯右值
6     int&& rvr1{ 22 }; // 右值引用可以引用纯右值
7     int& lvr7{ prv(2) }; // 错! 非常量左值引用不可引用纯右值
8     int&& rvr2{ prv(2) }; // 右值引用普通函数返回值
9     rvr1 = ++rvr2; // 右值引用做左值使用
10 }
```

7.12.1 右值引用

&&是右值引用, 函数返回的临时变量是右值

7.13 指针类型的函数

即函数的返回类型为指针类型, 需要注意的是: 不要将非静态局部地址用作函数的返回值, 因为非静态的局部变量的地址在函数执行完毕以后, 就销毁了。

正确的例子: 1. 主函数中定义的数组, 在子函数中对该数组的元素进行某种操作以后, 返回其中一个元素的地址, 这就是合法有效的地址。2. 在子函数中通过动态内存分配操作取得的内存地址返回给主函数是合法有效的, 但是内存分配和释放不在同一级别, 要注意不能忘记释放(在主函数中进行释放), 避免内存泄漏。

7.14 函数指针

如果把函数的地址作为参数传递给函数, 就可以在函数中灵活的调用函数。

使用函数指针的步骤:

1. 声明函数指针

2.使函数指针指向函数的地址

3.通过函数指针调用函数

如果在程序中定义了一个函数，那么在编译时系统就会为这个函数代码分配一段存储空间，这段存储空间的首地址称为这个函数的地址。而且函数名表示的就是这个地址。既然是地址我们就可以定义一个指针变量来存放，这个指针变量就叫作函数指针变量，简称函数指针。声明函数指针时，必须提供函数的类型，函数的类型就是**返回值和参数列表**。例如：

```
1 | int func(int a, string b);
```

上述函数的函数指针可以声明为：

```
1 | int (*FuncPtr)(int, string)
```

*FuncPtr两端的括号必不可少。如果不写括号，则FuncPtr是一个返回值是int指针的函数。

当我们把函数名作为一个值使用时，该函数自动地转换为指针。

```
1 | FuncPtr = length; //FuncPtr指向名为length的指针
2 | FuncPtr = &length; //和上述语句等价。取地址符是可选的
```

使用时可以直接使用指向函数的指针调用该函数，不需要解引用操作：

```
1 | int b1=FuncPtr(7, "hello");
2 | int b1=(*FuncPtr)(7, "hello"); //和上述语句的调用等价
3 | int b1=length(7, "hello"); //直接调用函数
```

指向不同函数类型的指针之间不存在转换规则。

需要注意的是，指向函数的指针变量没有 ++ 和 -- 运算。

```
1 | #include <iostream>
2 | using namespace std;
3 |
4 | int Max(int, int); //函数声明
5 |
6 | int main() {
7 |     int(*p)(int, int); //定义一个函数指针
8 |     int a, b, c;
9 |     p = Max; //把函数Max赋给指针变量p，使p指向Max函数
```

```
10     cout<<"please enter your number: "<<endl;
11     cin>>a>>b;
12     c = p(a, b); //通过函数指针调用Max函数
13     cout<<a<<b<<c<<endl;
14 }
15
16 int Max(int x, int y) //定义Max函数
17 {
18     int z;
19     if (x > y)
20     {
21         z = x;
22     }
23     else
24     {
25         z = y;
26     }
27     return z;
28 }
```

7.15 指针数组

指针数组即数组的元素是指针类型，例如：

```
1 | Point *pa[2];//由pa[0]和pa[1]两个指针组成
```

7.16 内置数组的排序

C++的内置函数qsort()可以用来对内置数组的元素进行排序。

函数的原型为：

```
1 | void qsort(void *base, size_t nitems, size_t size, int
(*compar)(const void *, const void*))
```

- base -- 指向要排序的数组的第一个元素的指针。
- nitems -- 由 base 指向的数组中元素的个数。
- size -- 数组中每个元素的大小，以字节为单位。
- compar -- 用来比较两个元素的函数。

函数不返回任何值。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int values[] = { 88, 56, 100, 2, 25 };
5
6 int cmpfunc (const void * a, const void * b)
7 {
8     return ( *(int*)a - *(int*)b );
9 }
10
11 int main()
12 {
13     int n;
14
15     printf("排序之前的列表: \n");
16     for( n = 0 ; n < 5; n++ ) {
17         printf("%d ", values[n]);
18     }
19
20     qsort(values, 5, sizeof(int), cmpfunc);
21
22     printf("\n排序之后的列表: \n");
23     for( n = 0 ; n < 5; n++ ) {
24         printf("%d ", values[n]);
25     }
26
27     return(0);
28 }
```

7.17 对象指针

对象指针的定义形式:

```

1 类名 *对象指针名;
2 Point a{5,10};
3 Point *ptr;
4 ptr=&a;
```

7.18 简化的C++内存模型

将内存划分为4个部分

1.stack(栈)

编译器自动分配释放，栈向低地址方向生长。

2.heap(堆)

一般由程序员分配释放，若程序员不释放，程序结束可能由操作系统回收。在C++中，即使用new和delete运算符实现。堆向高地址方向生长。

3.global/static(全局区/静态区)

全局变量和静态变量的存储是放在一块的。

可以简单的认为：

程序启动全局/静态变量就在此处

程序结束后释放

4.constant(常量区)

可以简单理解为所有常量都放在一块

该区域内容不可修改

7.19 using,typedef和#define的用法

#define和typedef:

#define是一个预处理指示符

1.用来定义宏。编译器不做类型检查

2.

```
1 #define TRUE 1 //结尾无分号，宏名为TRUE
2 //会将程序中所有出现TRUE的地方替换为1
```

typedef创建能在任何位置替代类名的别名

```
1 typedef someType newTypeName
2 示例：
3 typedef _Bool bool; //将bool作为类型_Bool的别名
```

用using替代typedef:

语法: using identifier=type-id

```
1 //类型别名, 等同于typedef unsigned int UINT;
2 using UInt=unsigned int;//名称'UInt'现在指代类型: unsigned int
3 UInt x=42u;
4
5 //类型别名, 等同于typedef void (*FuncType)(int,int);
6 using FuncType=void(*) (int,int);
7 //'FuncType'现在指代指向函数的指针
8 void example(int,int){}
9 FuncType f=example;
```

定义模板的别名, 只能使用using

8. 类: 更深层次的理解

We demonstrate how client code can access a class's public members via

- the name of an object and the dot operator (.)
- a reference to an object and the dot operator (.)
- a pointer to an object and the arrow operator (->)

8.1 接口与实现分离

我们的每个先前的类定义示例都将一个类放在一个头文件中进行重用, 然后将类的定义包含到一个包含main的源代码文件中, 这样我们就可以创建和操作类的对象。

传统的思想认为, 使用类的一个对象, 客户(例如main函数)只需要知道:

- 1.调用什么成员函数
- 2.需要向每个成员函数提供什么实参
- 3.期望从每个成员函数得到什么返回类型

客户端代码不需要知道这些功能是如何实现的。

隐藏类的实现细节使得更容易更改类的实现, 同时最小化, 并有希望消除对客户端代码的更改。

两个重要的C++软件工程概念：

1. 接口与实现分离(Separating interface from implementation.)
2. 在头文件中使用include guard以防止头文件中的代码被包含在同一个源代码文件中不止一次。由于一个类只能被定义一次，因此使用这样的预处理指令可以防止多定义错误。
宏保护除了文件名外，最好再加上包名或者自动生成的时间信息，避免异包同名文件的情况

8.1.1 类的接口

接口定义和规范了人和系统等事物之间的交互方式。例如，收音机的控件作为收音机用户与其内部组件之间的接口。控件允许用户执行一组有限的操作(如更改台站、调整音量、在AM台和FM台之间选择等)。不同的收音机可以实现操作的方式不同- -有的提供按钮，有的提供拨号，有的支持语音指令。该接口规定了无线电允许用户执行哪些操作，但没有说明这些操作是如何在无线电内部实现的。

类似地，类的接口描述了类的客户可以使用哪些服务以及如何请求这些服务，而不是类如何执行这些服务。类的Public接口由类的public成员函数(也被称为类的public services)组成。就像你很快看到的那样，你可以通过编写一个类定义来指定一个类的接口，这个类定义只列出类的成员函数原型和类的数据成员。

8.1.2 Separating the Interface from the Implementation(分离类的接口与实现)

为了将类的接口从实现中分离出来，我们将类Time分解成两个文件- -定义类Time的头文件Time . h和定义类Time成员函数的源代码文件Time . cpp ,因此：

1. 类是可重用的
2. 类的客户知道类提供哪些成员函数，如何调用它们以及期望的返回类型
3. 客户端不知道类的成员函数是如何实现的

根据约定，成员函数的定义放在和类同名的源文件中，且文件扩展名为.cpp。即成员函数在一个单独的.cpp文件中定义，类在.h文件中定义，客户即包含main函数的程序在另一个.cpp文件中定义。

8.1.3 ifndef/define/endif

ifndef/define/endif主要目的是防止头文件的重复包含和编译[C/C++中#ifndef的用法 - 简书 \(jianshu.com\)](#)

```
1 #include <string>
2 #ifndef TEST_10_13_TIME_H
3 #define TEST_10_13_TIME_H
4 class Time{
5 public:
6     void setTime(int,int,int);
7     std::string toUniversalString() const;
8     std::string toStandardString() const;
9 private:
10    unsigned int hour{0};
11    unsigned int minute{0};
12    unsigned int second{0};
13 };
14 #endif //TEST_10_13_TIME_H
```

当我们构建更大的程序时，其他的定义和声明也会放在头文件中。由于各种原因，头文件可能会出现问题。在项目编译期间，通常会编译每个.cpp文件。简单来说，这意味着编译器将获取您的.cpp文件，打开包含在其中的所有文件，将它们全部连接成一个大文本文件，然后执行语法分析，最后将其转换为一些中间代码，进行优化/执行其他任务，最后生成目标体系结构的程序集输出。因此，如果一个文件被多次包含在一个.cpp文件下，则编译器会将其文件内容添加两次，因此，**如果该文件中包含定义，则会出现编译器错误，提示您重新定义了一个变量**。当文件在编译过程中由预处理器步骤处理时，第一次到达其内容时，前两行将检查是否已为预处理器定义了CLASS_H。如果没有，它将定义CLASS_H并继续处理它与#endif指令之间的代码。下一次预处理器看到文件的内容时，对FILE_H的检查将为false，因此它将立即向下扫描到#endif并在此之后继续。这样可以防止重新定义错误。

当Time.h被首次#include时，标识符TIME_H还没有被定义。在这种情况下，#define指令定义了TIME_H，预处理器包含了.cpp文件中Time.h头文件的内容。如果再#include头文件，则由于已经定义了TIME_H，预处理器将忽略#ifndef和#endif之间的代码。

使用#endif, #defined和#endif预处理指令来形成一个include guard，以防止头文件在源代码文件中被多次包含。

8.1.4 作用域解析运算符(Scope Resolution Operator)

在代码中，在类的名字后面加作用域解析运算符，再加成员函数的名称。此运算符将成员函数与类的定义联系在一起。Time::告诉编译器每个成员函数都在该类的范围内，并且它的名字为其他类成员所知道。

“::”指明了成员函数所属的类。如：M::f(s)就表示f(s)是类M的成员函数。作用域，如果想在类的外部引用静态成员函数，或在类的外部定义成员函数都要用到。使用命名空间里的类型或函数也要用到（如：std::cout, std::cin, std::string 等等）

没有Time::，编译器就无法知道成员函数属于Time类。相反，编译器会认为它们是“自由(free)的”或“松散(loose)的”函数，比如main--这些函数也被称为全局函数。这种函数无法访问类的private数据或者调用类的成员函数。

8.1.5 Time Class Member Function toUniversalString and String Stream Processing

cout是标准输出流，ostringstream类的对象(来自头文件<sstream>)提供了相同的功能，但是会将输出写入内存中的string对象。使用ostringstream类的str成员函数得到格式化的string。

```
1 string Time::toUniversalString() const {
2     ostringstream output;
3     output<<setw(2)<<hour<<":"<<setw(2)
4     <<minute<<":"<<setw(2)<<second;
5     return output.str();
6 }
```

在上述代码中，ostringstream类创造了output对象。像使用cout一样使用output即可。参数化流操作符setfill用来指定填充字符，当一个整数在一个比该值的位数更宽的字段中输出时。填充字符出现在数字的左边，因为数字默认右对齐。例如分钟值为2，将会显示为02，因为填充字符设定为'0'。如果数值为两位数，如28，就不会进行填充，一旦操作符setfill使用，后面的所有值都会进行填充(sticky setting)。Ostringstream的str成员函数得到格式化字符串，返回给客户端。

Each sticky setting (such as a fill character or precision) should be restored to its previous setting when it's no longer needed. Failure to do so may result in incorrectly formatted output later in a program.

8.1.6 Implicitly Inlining Member Functions

如果在类的体中定义了成员函数，则成员函数被隐式地声明为内联的。请记住，编译器保留不内联任何函数的权利。

在类定义内部定义一个成员函数，将成员函数(如果编译器选择这样做)内联。这样可以提高性能。

在类头文件中只定义最简单、最稳定的成员函数(也就是说,其实现方式不可能改变)，因为对头的每一次更改都需要重新编译依赖于该头(在大型系统中,一项耗时的工作)的每个源代码文件。

8.1.7 成员函数对比全局函数

使用面向对象编程方法在调用函数时往往需要较少的参数。这种好处源于将数据成员和成员函数封装在一个类中，赋予了成员函数访问数据成员的权利。

成员函数通常比非面向对象程序中的函数更短，因为数据成员中存储的数据在理想情况下已经被构造函数或存储新数据的成员函数验证。由于数据已经在对象中，成员函数调用往往没有参数或者参数少于非面向对象语言中的函数调用。因此，函数调用、函数定义和函数原型都更短。这改善了程序开发的许多方面。

成员函数调用通常比常规函数调用(非面向对象的语言)不带或少带参数，这降低了传递错误参数、错误参数类型或错误参数数量的可能性。

8.1.8 对象的尺寸(Object Size)

刚接触面向对象编程的人常常假设对象必须相当大，因为它们包含数据成员和成员函数。从逻辑上讲，这是真的- -你可以把对象看作是包含数据和函数的(而我们的讨论也肯定鼓励了这一观点)；然而，从物理上看，事实并非如此。

对象只包含数据，因此对象比同时包含成员函数时要小得多。编译器从类的所有对象中分别创建成员函数的一个副本(仅)。类中的所有对象共享这一个副本。当然，每个对象都需要自己的类的数据副本，因为数据可以在对象之间变化。函数代码对于类中的所有对象都是一样的，因此，它们之间可以共享。

8.2 编译和链接过程

下图展示了编译和链接过程，生成可供使用的可执行Time应用程序。通常一个类的接口和实现将由一个程序员创建和编译，并由单独的程序员使用，程序员实现使用该类的客户代码。因此，该图显示了类实现程序员和客户代码程序员的需求。图中的虚线分别显示了类实现程序员、客户代码程序员和Time application user所需的片段。

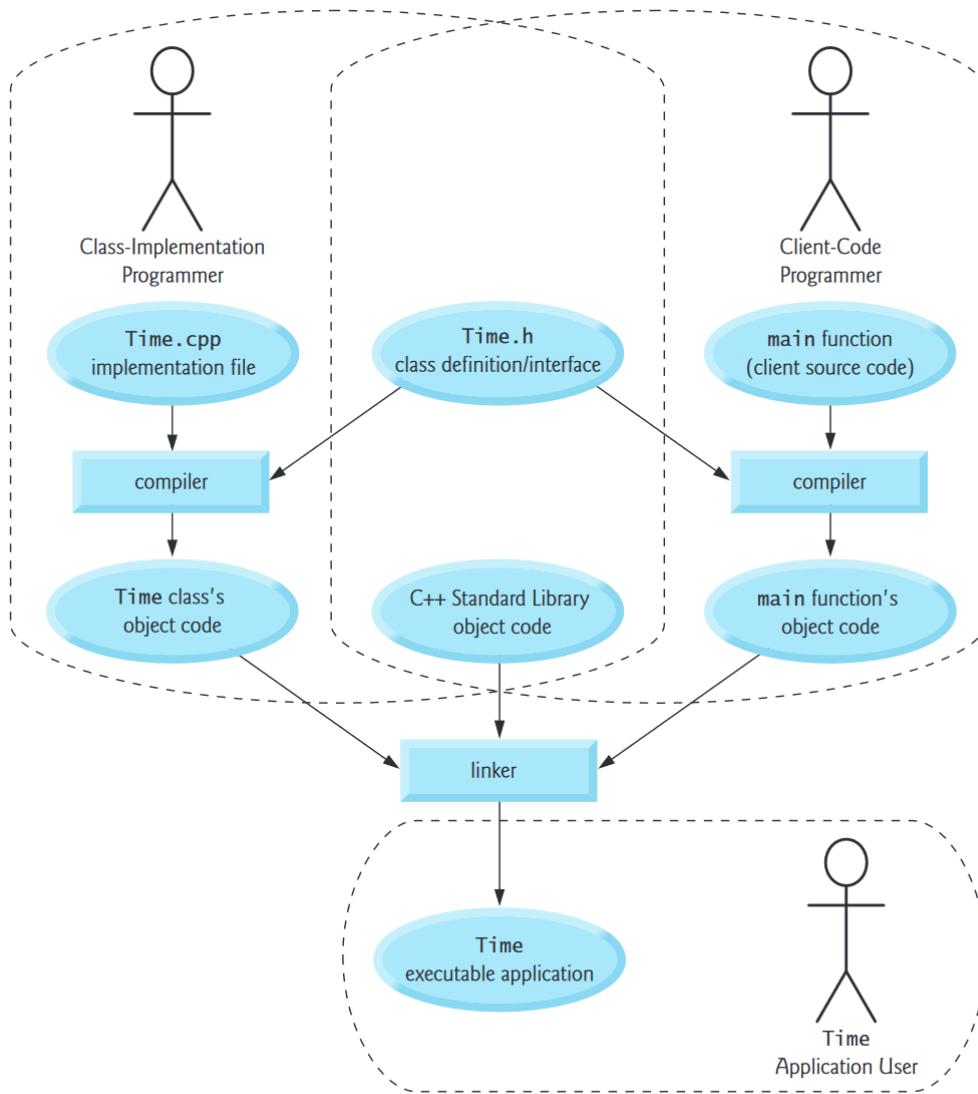


Fig. 9.4 | Compilation and linking process that produces an executable application.

一个类实现程序员负责创建一个可重用的Time类，创建头部Time . h和包含头部的源代码文件Time . cpp，然后编译源代码文件来创建Time的目标代码。为了隐藏类的成员函数实现细节，类实现程序员会向客户代码程序员提供报头Time . h (它规定了类的接口和数据成员)和Time对象代码(即代表Time成员函数的机器码指令)。客户代码程序员没有被赋予Time . cpp，因此客户仍然不知道Time的成员函数是如何实现的。

客户端代码程序员只需要知道Time的接口就可以使用类，并且必须能够链接它的目标代码。由于类的接口是Time . h头中类定义的一部分，因此客户代码程序员必须有权访问该文件，并且必须将其包含在客户的源代码文件中。在客户端代码编译时，编译器使用Time . h中的类定义，保证主函数正确创建和操作类Time的对象。

要创建可执行的Time应用程序，最后一步就是链接：

1. the object code for the main function (i.e., the client code),

2. the object code for class Time's member-function implementations, and
3. the C++ Standard Library object code for the C++ classes (e.g., string) used by the class-implementation programmer and the client-code programmer.

8.3 Class Scope and Accessing Class Members(类的范围和访问类的成员)

类的数据成员和成员函数属于类的范围。默认情况下，非成员函数被定义在全局命名范围(global namespace scope)。在类的范围内，类的成员可以被所有的类的成员函数访问并且可以通过名字被引用。在类的范围之外，public类成员可以通过下列三种方式被访问：

- 1.对象的名字
- 2.对一个对象的引用
- 3.一个对象的指针

我们称之为对象上的句柄。句柄的类型使编译器能够确定客户端通过该句柄可访问的接口(例如,成员函数)。

在类的作用域之外，类成员通过对象上的一个句柄（对象名、对对象的引用或对对象的指针）进行引用。

可以通过对象的名字或对象的引用，在后面加.访问对象的成员。通过指向对象的指针引用对象的成员，指针的名字和(>)后跟成员的名字，即：pointerName->memberName。

8.4 Access Functions and Utility Functions(访问函数和辅助函数)

8.4.1 访问函数(access functions)

访问函数可以读取或显示数据，而不修改数据。访问函数的另一个常用的应用是检验条件的真伪--这种函数通常被叫做谓词函数(predicate functions)，例如，任何容器类的empty函数- -一个能够容纳许多对象的类，如数组或向量。程序可能在试图从容器对象中读取另一个项目之前测试empty函数。

8.4.2 辅助函数

辅助函数(helper function)是一个private成员函数，它支持类的其他成员函数的操作。辅助函数被声明为private的，因为他们不是被类的客户使用的。一个常用的辅助函数的用途是在一个函数中放置一些通常的代码，这些代码本来可以在其他几个成员函数中复制。

8.5 构造函数及析构函数

如果某个类的成员函数已经提供了该类的构造函数或其他成员函数所需的全部或部分功能，则从该构造函数或其他成员函数中调用该成员函数。这简化了代码的维护，并降低了如果修改代码实现时出错的可能性。一般来说：避免重复代码。

构造函数可以调用类的其他成员函数，如set或get函数，但由于构造函数是对对象的初始化，数据成员可能还没有被初始化。在数据成员被正确初始化之前使用数据成员会导致逻辑错误。

将数据成员私有化，通过public成员函数控制对这些数据成员的访问，特别是写访问，有助于保证数据的完整性。

数据完整性的好处并不是自动生成的，因为数据成员是private的--您必须提供适当的有效性检查。

8.5.1 重载构造函数以及委托构造函数

类的构造函数和成员函数也可以重载。重载构造函数允许对象初始化不同类型 and/or 数量的实参。要重载一个构造函数，应该在类定义中为每个版本的构造函数提供一个原型，并为每个重载版本提供一个单独的构造函数定义。这也适用于类的成员函数。例如，对于下列代码：

```
1 | clock(int newH, int newM, int
2 |     newS) : hour(newH), minute(newM), second(newS) {} //构造函数
  | clock() : hour(0), minute(0), second(0) {} //构造函数
```

可以使用委托构造函数的形式，将上述代码改写为：

```
1 | clock(int newH, int newM, int
2 |     newS) : hour(newH), minute(newM), second(newS) {}
  | clock : clock(0, 0, 0) {}
```

就像构造函数可以调用类的成员函数，C++11允许构造函数调用同一类中的其他构造函数。调用的这个构造函数叫做委托构造函数，将自己的工作委托给其他构造函数。

8.5.2 委托构造函数/代理构造函数

即一个构造函数可以调用另外的构造函数

```
1 class A{  
2     public:  
3         A():A(0){}//调用了带一个参数的构造函数  
4         A(int i):A(i,0){}  
5         A(int i,int j){  
6             num1=i;  
7             num2=j;  
8             average=(num1+num2)/2;  
9         }  
10    private:  
11        int num1;  
12        int num2;  
13        int average;  
14    };
```

被委托的构造函数要放在主调构造函数的初始化构造列表位置。

避免递归调用构造函数

8.6 析构函数(Destructors)

析构函数是一种成员函数的类型。析构函数的参数列表是空的。析构函数的命名格式为~加类的名字：

```
1 ~类名()
```

析构函数也许不会指定形参以及返回类型。当对象销毁时，析构函数被隐式的调用。例如，当程序执行离开对象被实例化的范围时，对象就会被破坏。析构函数本身并没有释放对象的内存---它执行终止事务管理(termination housekeeping)的功能在对象的内存再生之前，所以内存可能被用于存储新的对象。

每个类都有析构函数，如果没有显式的定义析构函数，编译器会定义一个"empty"析构函数。

析构函数不可以重载，可以手动调用。构造函数不能手动调用。

8.7 当构造函数和析构函数都被调用

这些函数调用发生的顺序取决于执行进入和离开对象实例化的范围的顺序。一般的，析构函数的调用和它对应的构造函数的调用顺序相反。

8.7.1 全局范围内对象的构造函数和析构函数

在其他程序中的任何函数(包括main函数)开始执行之前(尽管全局对象的构造函数的执行顺序并不保证)，构造函数被定义在全局范围的对象调用。当main终止时，相应的析构函数被调用。exit函数强迫程序立即终止并且不执行本地对象的析构函数。exit函数通常出现在程序发生不可修复的错误时。abort函数和exit函数类似，会迫使程序立即终止，并且不允许调用程序员定义的任何类型的清理代码。

8.7.2 non-static局部对象的构造函数和析构函数

非静态局部对象的构造函数在执行到达该对象被定义的位置时被调用，相应的析构函数在执行离开对象的范围(也就是说,该对象所定义的块已经执行完毕)时被调用。

如果程序以exit函数或abort函数的方式终止，则不调用本地对象的析构函数。

8.7.3 static局部对象的构造函数和析构函数

静态局部对象的构造函数只调用一次，当执行首先到达对象被定义的地方时- -当main程序终止或程序调用exit函数退出时，相应的析构函数被调用。

全局和static对象销毁的顺序和他们创建的顺序相反。static对象不会调用析构函数如果程序调用abort函数终止。

8.7.4 构造函数和析构函数何时被调用

```
1 //  
2 // Created by 22364 on 2023/10/15.  
3 //  
4 #include <string>  
5  
6 #ifndef TEAT_10_15_2_CREATEANDDESTROY_H  
7 #define TEAT_10_15_2_CREATEANDDESTROY_H  
8  
9 class CreateAndDestroy{  
10 public:  
11     CreateAndDestroy(int, std::string);
```

```
12     ~CreateAndDestroy();
13
14 private:
15     int objectID;
16     std::string message;
17 };
18 #endif //TEAT_10_15_2_CREATEANDDESTROY_H
```

```
1 #include <iostream>
2 #include "CreateAndDestroy.h"
3 using namespace std;
4
5 CreateAndDestroy::CreateAndDestroy(int ID, string
6     messageString)
7     :objectID{ID}, message{messageString} {
8     cout<<"object "<<objectID<<" constructor runs "
9     <<message<<endl;
10 }
11 CreateAndDestroy::~CreateAndDestroy(){
12     cout<<(objectID==1 || objectID==6?"\n":(""));
13     cout<<"object "<<objectID<<" destroy runs "
14     <<message<<endl;
15 }
```

```
1 #include <iostream>
2 #include "CreateAndDestroy.h"
3 using namespace std;
4
5 void create();
6 CreateAndDestroy first{1, "(gobal before main)"}; //全局对象
7
8 int main() {
9     cout<<"\nMAIN FUNCTION: EXECUTION BEGINS"<<endl;
10     CreateAndDestroy second{2, "(local in main)"};
11     static CreateAndDestroy third{3, "(local static in
12     main)"};
13
14     create();
15
16     cout<<"\nMAIN FUNCTION: EXECUTION RESUMES"<<endl;
17     CreateAndDestroy fourth{4, "(local in main)"};
18 }
```

```

17     cout<<"\nMAIN FUNCTION: EXECUTION ENDS" << endl;
18 }
19
20 void create() {
21     cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
22     CreateAndDestroy fifth{5, "(local in create)"};
23     static CreateAndDestroy sixth{6, "(local static in
24 create)"};
25     CreateAndDestroy seventh{7, "(local in create)"};
26     cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
27 }
```

```

Object 1 constructor runs (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2 constructor runs (local in main)
Object 3 constructor runs (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5 constructor runs (local in create)
Object 6 constructor runs (local static in create)
Object 7 constructor runs (local in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7 destructor runs (local in create)
Object 5 destructor runs (local in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4 constructor runs (local in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4 destructor runs (local in main)
Object 2 destructor runs (local in main)

Object 6 destructor runs (local static in create)
Object 3 destructor runs (local static in main)

Object 1 destructor runs (global before main)
```

上述代码在main函数中声明了三个对象，当程序执行到对象声明处时，每个对象的构造函数被调用。fourth和second对象的析构函数调用的顺序和他们构造函数的调用顺序相反，当程序的执行到达main函数的结尾时。因为third对象声明为static的，所以在程序终止以后仍然存在。third对象的析构函数的调用在first对象的析构函数调用之前，但是在所有其他对象销毁之后。

seventh的析构函数先调用再调用fifth对象的析构函数(和他们构造函数的调用顺序相反), 在create函数终止时。sixth对象被声明为static,在程序终止以后仍然存在。sixth对象的析构函数调用在third和first对象的析构函数的调用之前, 但是在所有其他对象销毁之后。

8.8 Default Memberwise Assignment(默认逐成员初始化)

```
1 //  
2 // Created by 22364 on 2023/10/17.  
3 #include <string>  
4  
5 #ifndef TEST_10_17_DATE_H  
6 #define TEST_10_17_DATE_H  
7  
8 class Date{  
9 public:  
10     explicit Date(unsigned int=1,unsigned int =1,unsigned  
11     int=2000);  
12     std::string toString() const;  
13 private:  
14     unsigned int month;  
15     unsigned int day;  
16     unsigned int year;  
17 };  
18 #endif //TEST_10_17_DATE_H
```

```
1 #include <sstream>  
2 #include <string>  
3 #include "Date.h"  
4 using namespace std;  
5  
6 Date::Date(unsigned int m, unsigned int d, unsigned int y)  
7     :month{m},day{d},year{y} {}  
8 string Date::toString() const {  
9     ostringstream output;  
10    output<<month<<'/''<<day<<'/''<<year;  
11    return output.str();  
12 }
```

```

1 #include <iostream>
2 #include "Date.h"
3 using namespace std;
4
5 int main() {
6     Date date1{7,4,2004};
7     Date date2;
8
9     cout<<"date1= "<<date1.tostring()
10    <<"\n"date2="<<date2.tostring()<<"\n\n";
11
12    date2=date1;
13
14    cout<<"after default memberwise assignment,date2="
15    <<date2.tostring()<<endl;
16
17 }

```

赋值操作符=可以用来将同一个类的一个对象赋值给另一个对象。默认情况下，这种指派是通过memberwise assignment(也称为copy assignment)进行的- -赋值运算符右侧对象的每个数据成员被单独赋值到赋值运算符左侧对象中的同一个数据成员。

注意： 当使用的类的数据成员包含指向动态地址的指针时，Memberwise assignment会引起严重的问题；我们在后续内容讨论了这些问题，并给出了如何处理这些问题的方法。

对象可以作为函数参数传递，也可以从函数返回。这种传递和返回默认情况下使用数值传递执行- -传递或返回对象的一个副本。在这种情况下，C + +创建一个新的对象，并使用一个拷贝构造函数将原对象的值拷贝到新对象中。对于每个类，编译器提供一个默认的副本构造器，将原对象的每个成员复制到新对象的相应成员中。同成员分配一样，拷贝构造函数在与数据成员包含指针的类一起使用动态分配内存时会引起严重的问题。

8.9 实例化(instantiation)

就像有人在实际驾驶汽车之前必须从工程图纸中创建一个汽车一样，在一个程序能够执行类的成员函数定义的任务之前，必须从类中创建一个对象。这样做的过程被称为实例化。然后，一个对象被称为它类的一个实例。

8.10 const对象和const成员函数

一些对象不需要被改变，就可以声明为const。 You may use const to specify that an object is not modifiable and that any attempt to modify the object should result in a compilation error.

```
1 | const Time noon{12,0,0};
```

C++不允许const对象调用成员函数，除非成员函数也被声明为const的。即使对于不修改对象的get成员函数也是如此。这也是我们将所有不修改被调用对象的成员函数都声明为const的一个关键原因。

const对象只能调用const成员函数

const成员函数只能调用const成员函数

const成员函数中不能改变成员变量的值

定义一个const成员函数调用non-const成员函数是编译错误。

const对象调用non-const对象是编译错误

试图将构造函数和析构函数声明为const是编译错误。

即使成员函数没有改变对象，但是const对象仍然不能调用它，成员函数必须显式的声明为const，才可以被const对象调用。

8.11 Composition: Objects as Members of Classes(组成：对象作为类的成员)

```
1 //Date.h
2 // Created by 22364 on 2023/10/17.
3 #include <string>
4
5 #ifndef TEST_10_17_DATE_H
6 #define TEST_10_17_DATE_H
7
8 class Date{
9 public:
10     static const unsigned int monthsPerYear{12};
11     explicit Date(unsigned int=1,unsigned int =1,unsigned
12     int=1900);
13     std::string toString() const;
```

```

13     ~Date() // provided to confirm destruction order
14
15     private:
16         unsigned int month;
17         unsigned int day;
18         unsigned int year;
19
20         unsigned int checkDay(int) const;
21
22 #endif //TEST_10_17_DATE_H

```

```

1 //Date.cpp
2 #include <sstream>
3 #include "Date.h"
4 #include <array>
5 #include <iostream>
6 #include <stdexcept>
7 using namespace std;
8
9 Date::Date(unsigned int mn, unsigned int dy, unsigned int
10 yr)
11     :month{mn}, day{dy}, year{yr} {
12     if(mn<1 || mn>monthsPerYear){
13         throw invalid_argument("month must be 0-12");
14     }
15     cout<<"date object constructor for date"<<tostring()
16     <<endl;
17 }
18 string Date::toString() const {
19     ostringstream output;
20     output<<month<<'/'<<day<<'/'<<year;
21     return output.str();
22 }
23
24 Date::~Date(){
25     cout<<"date object destructor for date"<<tostring()
26     <<endl;
27 }
28
29 unsigned int Date::checkDay(int testDay) const {
30     static const array<int, monthsPerYear + 1>
31     daysPerMonth{

```

```
28     0,31,28,31,30,31,30,31,31,30,31,30,31,30,31};  
29     if(testDay>0&&testDay<=daysPerMonth[month]) {  
30         return testDay;  
31     }  
32     if (month==2&&testDay==29&&(year%400==0 ||  
33 (year%4==0&&year%100!=0))) {  
34         return testDay;  
35     }  
36     throw invalid_argument("Invalid day for current month  
and year");  
37 }
```

```
1 //Employee.h
2 #include <string>
3
4 #ifndef TEST_10_17_EMPLOYEE_H
5 #define TEST_10_17_EMPLOYEE_H
6
7 #include <string>
8 #include "Date.h"
9
10 class Employee{
11 public:
12     Employee(const std::string&, const std::string&, const
13 Date&, const Date&);
14     std::string toString() const;
15     ~Employee();
16
17 private:
18     std::string firstName;
19     std::string lastName;
20     const Date birthDate;
21     const Date hireDate;
22 };
23 #endif //TEST_10_17_EMPLOYEE_H
```

```
1 //Employ.cpp
2 #include <iostream>
3 #include <sstream>
4 #include "Employee.h"
5 #include "Date.h"
6 using namespace std;
```

```

7
8 Employee::Employee(const std::string& first, const
9   std::string& last, const Date &dateOfBirth, const Date
10  &dateOfHire)
11    :firstName(first), lastName(last),
12    birthDate(dateOfBirth), hireDate(dateOfHire){
13      cout<<"Employee object constructor: "<<firstName<<
14      ' '<<lastName<<endl;
15
16
17  string Employee::toString() const {
18    ostringstream output;
19    output<<lastName<<, " <<firstName<<" Hired: "
20    <<hireDate.toString()<<" birthday: "
21    <<birthDate.toString();
22    return output.str();
23
24
25  Employee::~Employee(){
26    cout<<"Employee object destructor: "<<lastName << ", "
27    << firstName << endl;
28 }
```

```

1 //main.cpp
2 #include <iostream>
3 #include "Date.h"
4 #include "Employee.h"
5 using namespace std;
6
7
8 int main() {
9     Date birth{7,24,1949};
10    Date hire{3,12,1988};
11    Employee manager{"Bob","Blue",birth,hire};
12
13    cout<<"\n"<<manager.toString()<<endl;
14 }
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob Hired: 3/12/1988 Birthday: 7/24/1949
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

在一个类中包含其他类的对象，就叫做composition(组合)(or aggregation(聚合))。现在，我们展示了一个类的构造函数如何通过成员初始化子(member initializers)将参数传递给成员对象构造函数。

数据成员按照在类定义(不是按照它们在构造函数member-initializer list的顺序)中声明的顺序进行构造，并在构造其包含的类对象之前进行构造。

8.11.1 类的默认副本构造函数

在类class的定义内，构造函数没有接收Date类型的形参，那么，Employee构造函数的成员初始化列表为什么可以通过向Date类的构造函数传递Date类的对象来初始化birthDate和hireDate。正如我们在之前提到的，编译器为每个类提供了一个默认的复制构造函数，它将构造函数的参数对象的每个数据成员复制到被初始化对象的相应成员中。

即，birth和hire均是Date的对象，Employee的对象manager将对象birth和hire作为实参传递给Employee的构造函数，在Employee的构造函数的成员初始化列表里，将Date对象dateOfBirth和dateOfHire传递给在Employee类中定义的private数据成员，此数据成员即为const的Date对象birthDate和hireDate。问题在于，Date类的构造函数并没有接收对象的形参，原因就在于类的默认副本构造函数。

上诉运行的结果，可以看出析构函数执行的顺序。这些输出证实了Employee对象是从外部被销毁的，即Employee析构函数先运行(输出显示从输出窗口底部开始的五条线)，然后成员对象被破坏的顺序与它们被构造的顺序相反。

string类的析构函数不包含输出语句。

8.11.2 如果不使用成员初始化列表？

如果一个成员对象不通过成员初始化列表初始化，成员对象的默认构造函数会被隐式调用。默认构造函数建立的值(如果有的话)可以用set函数重写。然而，对于复杂的初始化，这种方法可能需要大量的额外工作和时间。

Initialize member objects explicitly through member initializers. This eliminates the overhead of “doubly initializing” member objects—once when the member object’s default constructor is called and again when set functions are called in the constructor body (or later) to initialize the member object.

如果一个数据成员是另一个类的对象，`public`该成员对象并不违反对该成员对象`private`成员的封装和隐藏。但是，它却违背了封装类实现的封装性和隐藏性，所以类的成员对象仍然应该是`private`的。

8.12 友元函数和友元类(friend Functions and friend Classes)

类的友元函数是一个非成员函数，有权访问类的public和non-public成员。单独的函数，整个类或者其他类的成员函数都可以被声明为其他类的friends。友元的目的就是让一个函数或者类访问另一个类中的`private`成员

在当前类以外定义的、不属于当前类的函数也可以在类中声明，但要在前面加`friend`关键字，这样就构成了友元函数。友元函数可以是不属于任何类的非成员函数，也可以是其他类的成员函数。

友元函数可以访问当前类中的所有成员，包括`public`、`protected`、`private`属性的。

还可以将整个类声明为另一个类的“friend”，这就是友元类。友元类中的所有成员函数都是另外一个类的友元函数。类的友元关系是单向的。例如：如果声明B类是A类的友元，B类的成员函数就可以访问A类的私有和保护数据，但A类的成员函数却不能访问B类的私有保护数据。

8.12.1 友元的声明

将`classTwo`声明为`classOne`的友类，只要在`classOne`的定义中声明如下语句：

```
1 | friend class ClassTwo;
```

`friend`的声明可以出现在类的任意位置，不受访问限定符`public`和`private`和`protected`的影响。友谊是被给予的，而不是被接受的——为了让B类成为a类的朋友，a类必须明确声明B类是它的朋友。友谊不是对称的，如果A类是B类的朋友，你就不能推断B类是A类的朋友。友谊是不可传递的——如果A类是B类的朋友，B类是C类的朋友——你就不能推断出A类是C类的朋友。

8.12.2 通过友元函数修改类的private成员

```
1 #include <iostream>
2 using namespace std;
3
4 class Count{
5     friend void setX(Count&, int);
6 public:
7     int getX() const{return x;}
8 private:
9     int x{0};
10};
11
12 void setX(Count& c, int val){
13     c.x=val;
14}
15
16 int main(){
17     Count counter;
18     cout<<"counter.x after instantiation: " <<
19     counter.getX() << endl;
20     setX(counter, 8); //25 set x using a friend function 友元
21     //函数直接在main函数中调用
22     cout<<"counter.x after call to setX friend function: " <<
23     counter.getX() << endl;
24 }
```

setX函数是一个独立(全局)函数，不是Count类的成员函数。由于此原因，当setX函数被counter对象调用，19行的语句将counter作为实参传给setX，而不是使用对象的名字调用函数---例如(counter.setX(8))。函数setX被允许访问类Count的私有数据成员x，只是因为setX被声明为该类的friend。

可以将重载函数指定为类的友元。每个想要成为友元的函数都必须在类定义中明确声明为类的友元。

即使友元函数的原型出现在类定义中，友元也不是成员函数。

将所有friend声明首先放在类定义的主体内，并且不要在它们之前使用任何访问说明符。

友元类和友元函数打破了类的封装性

8.13 使用this指针

每个类的功能只有一个副本，但一个类可以有很多对象，那么成员函数如何知道要操作哪个对象的数据成员呢？每个对象都可以通过一个名为this（C++关键字）的指针访问自己的地址。this指针不是对象本身的一部分，也就是说，this指针占用的内存不会反映在对象的sizeof操作的结果中。相反，this指针（由编译器）作为隐式参数传递给对象的每个**非静态成员函数**。后续介绍了静态类成员，并解释了为什么this指针没有隐式传递给静态成员函数。

this是C++中的一个关键字，也是一个**const**指针，它指向当前对象，通过它可以访问当前对象的所有成员。

所谓当前对象，是指正在使用的对象。例如对于`stu.show();`，stu就是当前对象，this就指向stu。this只能用在类的内部，通过this可以访问类的所有成员，包括private、protected、public属性的。**

this虽然用在类的内部，但是只有在对象被创建以后才会给this赋值，并且这个赋值的过程是编译器自动完成的，不需要用户干预，用户也不能显式地给this赋值。

8.13.1 使用this指针避免命名冲突

```
1 void Time::setHour(int hour){  
2     if(hour>=0&&hour<24){  
3         this->hour=hour;  
4     }  
5     else{  
6         throw invalid_argument("hour must be 0-23");  
7     }  
8 }
```

一种通常的显式的使用this指针的方法是防止类的数据成员和成员函数形参（或者是其他局部变量）的命名冲突

例如，一个成员函数的局部变量和类的数据成员名字相同，如上述代码所示，局部变量被认为隐藏hide或遮盖shadow了数据成员---仅使用函数体内的局部变量的名字代表局部变量而不是数据成员。但是可以使用->操作符访问数据成员。

```
1 this->hour=hour;
```

A widely accepted practice to minimize the proliferation of identifier names is to use the same name for a set function's parameter and the data member it sets, and to reference the data member in the set function's body via this->.

8.13.2 this指针的类型

this指针的类型取决于对象的类型，以及在其中使用this指针的成员函数是否声明为const:

1.在Employee类的non-const成员函数中，this指针的类型为Employee* const——一个指向nonconstant Employee的constant指针。

2.在const成员函数，this指针的类型为const Employee* const——指向constant Employee的constant指针。

8.13.3 隐式和显式的使用this指针访问对象的数据成员

```
1 #include <iostream>
2 using namespace std;
3
4 class Test{
5 public:
6     explicit Test(int);
7     void print() const;
8 private:
9     int x{0};
10 };
11
12 Test::Test(int value)
13 :x{value} {
14 }
15
16 void Test::print() const {
17     cout << " x = " << x;
18     cout << "\n this->x = " << this->x;
19     cout << "\n(*this).x = " << (*this).x << endl;
20 }
21
22 int main() {
23     Test testObject{12};
24     testObject.print();
25 }
```

```
x = 12
this->x = 12
(*this).x = 12
```

this指针的一个有趣的用途是防止对象被分配给它自己

8.13.4 使用this指针完成级联函数(Cascaded Function)调用

this指针的另一个用途是启用级联成员函数调用——也就是说，在同一语句中顺序调用多个函数。

```
1 //  
2 // Created by 22364 on 2023/10/20.  
3 //  
4 #include <string>  
5  
6 #ifndef TEST_10_20_TIME_H  
7 #define TEST_10_20_TIME_H  
8  
9 class Time{  
10 public:  
11     explicit Time(int=0,int=0,int=0);  
12  
13     Time& setTime(int,int,int);  
14     Time& setHour(int);  
15     Time& setMinute(int);  
16     Time& setSecond(int);  
17  
18     unsigned int getHour() const;  
19     unsigned int getMinute() const;  
20     unsigned int getSecond() const;  
21     std::string toUniversalString() const;  
22     std::string toStandardString() const;  
23 private:  
24     unsigned int hour{0};  
25     unsigned int minute{0};  
26     unsigned int second{0};  
27  
28 };  
29 #endif //TEST_10_20_TIME_H
```

```
1 #include <iomanip>  
2 #include <sstream>  
3 #include <stdexcept>  
4 #include "Time.h"  
5 using namespace std;
```

```
6
7 Time::Time(int hr, int min, int sec) {
8     setTime(hr, min, sec);
9 }
10
11 Time& Time::setTime(int h, int m, int s){
12     setHour(h);
13     setMinute(m);
14     setSecond(s);
15     return *this;
16 }
17
18 Time& Time::setHour(int h){
19     if (h >= 0 && h < 24){
20         hour = h;
21     }
22     else{
23         throw invalid_argument("hour must be 0-23");
24     }
25     return *this;
26 }
27
28 Time& Time::setMinute(int m){
29     if (m >= 0 && m < 60){
30         minute = m;
31     }
32     else{
33         throw invalid_argument("minute must be 0-59");
34     }
35     return *this;
36 }
37 Time& Time::setSecond(int s){
38     if (s >= 0 && s < 60){
39         second = s;
40     }
41     else{
42         throw invalid_argument("second must be 0-59");
43     }
44     return *this;
45 }
46
47 unsigned int Time::getHour() const {return hour;}
```

```

48
49     unsigned int Time::getMinute() const {return minute;}
50
51     unsigned int Time::getSecond() const {return second;}
52
53     string Time::toUniversalString() const{
54         ostringstream output;
55         output << setfill('0') << setw(2) << getHour() << ":"
56         <<setw(2) << getMinute() << ":" << setw(2) <<
57         getSecond();
58         return output.str();
59     }
60
61     string Time::toStandardString() const{
62         ostringstream output;
63         output << ((getHour() == 0 || getHour() == 12) ? 12 :
64         getHour() % 12)
65         << ":" << setfill('0') << setw(2) << getMinute() << ":"
66         << setw(2)
67         <<getSecond() << (hour < 12 ? " AM" : " PM");
68         return output.str();
69     }

```

```

1 #include <iostream>
2 #include "Time.h"
3 using namespace std;
4
5 int main(){
6     Time t;
7     t.setHour(18).setMinute(30).setSecond(22); // cascaded
8     function calls
9
10    cout<<"Universal time: " << t.toUniversalString()
11    <<"\nStandard time: " << t.toStandardString();
12
13    cout<<"\n\nNew standard time: "
14    <<t.setTime(20, 20, 20).toStandardString()
15    <<endl; //cascading
16 }

```

member-function calls (lines 10 and 18). Why does the technique of returning `*this` as a reference work? The dot operator (.) associates from left to right, so line 10

```
t.setHour(18).setMinute(30).setSecond(22);
```

first evaluates `t.setHour(18)`, then returns a reference to (the updated) object `t` as the value of this function call. The remaining expression is then interpreted as

```
t.setMinute(30).setSecond(22);
```

The `t.setMinute(30)` call executes and returns a reference to the (further updated) object `t`. The remaining expression is interpreted as

```
t.setSecond(22);
```

8.14 static类成员(静态成员)

这个规则有一个重要的例外，即类的每个对象都有自己的类的所有数据成员的副本。在某些情况下，类的所有对象只应共享变量的一个副本。使用静态数据成员是出于这些和其他原因。这样的变量表示“类范围”的信息，即由所有实例共享的数据，并且不特定于类的任何一个对象。即，静态数据成员为该类的所有对象共享，静态数据成员具有静态生存期。

8.14.1 Motivating Classwide Data

当一个类的所有对象都有一个数据副本就足够了时，使用静态数据成员来节省存储空间，比如一个可以由该类所有对象共享的常量。

8.14.2 static数据成员的范围和初始化

基本类型静态数据成员默认初始化为0。 如果静态数据成员是提供默认构造函数的类的对象，则无需初始化该静态数据成员，因为将调用其默认构造函数。静态成员变量只能放在所有成员函数的外部进行初始化。

1. 声明为“`constexpr`”类型的静态数据成员必须在类中声明并初始化。
2. 声明为“`inline`”(C++17起)或者“`const int`”类型的静态数据成员可以在类中声明并初始化。
3. 其他情况下，静态数据成员必须在类的外部进行定义并初始化，且不带`static`关键字。

8.14.3 访问静态数据成员

通常使用类的`public`的成员函数或者`friends`访问类的`private`(以及`protected`)静态成员。类的静态成员存在，即使不存在该类的对象。要在不存在类的对象时访问`public`静态类成员，只需在数据成员的名称前面加上类名和作用域解析运算符`(::)`。例如，如果我们前面的变量`martianCount`是公共的，则可以使用表达

式Martian:: martianCount来访问它，即使在没有火星对象的情况下也是如此。**（当然，不鼓励使用公共数据。）**

要在不存在私有或受保护的静态类成员的对象时访问该类成员，请提供一个公共静态成员函数，并通过在其名称前加类名和作用域解析运算符来调用该函数。静态成员函数是类的服务，而不是类的特定对象的服务。

类的静态数据成员和静态成员函数存在，即使没有实例化该类的对象，也可以使用。

被访问 主调函数	静态	非静态
静态	通过类名/对象名	只能通过对象名
非静态	只能通过对象名	☺

例如以下代码：

```
1 class A{
2     public:
3         A(int a=0){
4             x=a;
5         }
6         static void f1();
7         static void f2(A a);
8     private:
9         int x;
10        static int y;
11    };
12 void A::f2(A a){
13     cout<<A::y;
14     cout<<x;//error 静态成员函数访问非静态成员变量，只能通过对象
的名字访问
15     cout<<a.x;//right
16 }
17 void A::f1(){
18     cout<<A::y<<endl;
19 }
20 int main(){
21     A::f1();
22     A mA(3);
23     A::f2(mA);
24     mA.A::f1();
```

```
25 }
```

当变量和函数不依赖于类的实例时，在类中使用静态成员。

8.14.4 Demonstrating static Data Members

```
1 #ifndef TEST_10_20_2_EMPLOYEE_H
2 #define TEST_10_20_2_EMPLOYEE_H
3 #include <string>
4
5 class Employee{
6 public:
7     Employee(const std::string&, const std::string&);
8     ~Employee();
9     std::string getFirstName() const;
10    std::string getLastName() const;
11
12    static unsigned int getCount();
13 private:
14    std::string firstName;
15    std::string lastName;
16
17    static unsigned int count;
18 };
19#endif //TEST_10_20_2_EMPLOYEE_H
```

```
1 #include <iostream>
2 #include "Employee.h"
3 using namespace std;
4
5 unsigned int Employee::count{0};
6
7 unsigned int Employee::getCount() {return count;}
8
9 Employee::Employee(const std::string& first, const
10    std::string& last)
11    :firstName(first), lastName(last){
12    ++count;
13    cout << "Employee constructor for " << firstName
14    << ' ' << lastName << " called." << endl;
15 }
```

```

16 Employee::~Employee() {
17     cout<<"~Employee() called for " << firstName
18     << ' ' << lastName << endl;
19     --count;
20 }
21
22 string Employee::getFirstName() const {return firstName;}
23
24 string Employee::getLastName() const {return lastName;}

```

```

1 #include <iostream>
2 #include "Employee.h"
3 using namespace std;
4
5 int main() {
6     cout<<"Number of employees before instantiation of any
7     objects is "
8     <<Employee::getCount()<<endl;
9     {
10         Employee e1{"Susan", "Baker"};
11         Employee e2{"Robert", "Jones"};
12
13         cout<<"Number of employees after objects are
14         instantiated is "
15         <<Employee::getCount();
16
17         cout<<"\n\nEmployee 1: "
18         <<e1.getFirstName() << " " << e1.getLastName()
19         <<"\nEmployee 2: "
20         <<e2.getFirstName() << " " << e2.getLastName() <<
21         "\n\n";
22     }
23
24     cout << "\nNumber of employees after objects are
25     deleted is "
26     <<Employee::getCount()<<endl;
27 }

```

通过上图可以看到，在Employee的对象创造之前，就使用了getCount函数，在Employee对象销毁后，仍然可以使用getCount函数。如果成员函数不访问类的非静态数据成员或非静态成员函数，则应将其声明为静态。与非静态成员函数不同，静态成员函数没有this指针，因为静态数据成员和静态成员函数独立于类的

任何对象而存在。this指针必须引用类的特定对象，并且当调用静态成员函数时，内存中可能没有该类的任何对象。

Declaring a static member function const is a compilation error. The const qualifier indicates that a function cannot modify the contents of the object on which it operates, but static member functions exist and operate independently of any objects of the class.

8.15.4.1 nested scope

If two different entities named by the same identifier are in scope at the same time, and they belong to the same name space, the scopes are nested (no other form of scope overlap is allowed), and the declaration that appears in the inner scope hides the declaration that appears in the outer scope:

```
1 // The name space here is ordinary identifiers.
2
3 int a;    // file scope of name a begins here
4
5 void f(void)
6 {
7     int a = 1; // the block scope of the name a begins
here; hides file-scope a
8     {
9         int a = 2;          // the scope of the inner a
begins here, outer a is hidden
10        printf("%d\n", a); // inner a is in scope, prints 2
11    }                      // the block scope of the inner a
ends here
12    printf("%d\n", a);    // the outer a is in scope,
prints 1
13 }                      // the scope of the outer a ends
here
14
15 void g(int a); // name a has function prototype scope;
hides file-scope a
```

8.14.5 命名空间

8.14.5.1 概述

在C++中，名称（name）可以是符号常量、变量、函数、结构、枚举、类和对象等等。工程越大，名称互相冲突性的可能性越大。另外使用多个厂商的类库时，也可能导致名称冲突。为了避免在大规模程序的设计中，以及在程序员使用各种各样的C++库时，这些标识符的命名发生冲突，标准C++引入关键字 `namespace`（命名空间/名字空间/名称空间），可以更好地控制标识符的作用域。

8.14.5.2 定义

```
1 //定义一个名字为A的命名空间（变量、函数）
2 namespace A {
3     int a = 100;
4 }
5 namespace B {
6     int a = 200;
7 }
8 void test02()
9 {
10     //A::a a是属于A中
11     cout<<"A中a = "<<A::a<<endl;//100
12     cout<<"B中a = "<<B::a<<endl;//200
13 }
```

8.14.5.3 命名空间只能全局范围内定义（以下为错误写法）

```
void test(){
    //局部定义命名空间是错误的
    namespace A{
        int a = 10;
    }
    namespace B{
        int a = 20;
    }
    cout << "A::a : " << A::a << endl;
    cout << "B::a : " << B::a << endl;
}
```

8.14.5.4 命名空间可以嵌套

```
1 namespace A {  
2     int a = 1000;  
3     namespace B {  
4         int a = 2000;  
5     }  
6 }  
7 void test03()  
8 {  
9     cout<<"A中的a = "<<A::a<<endl; //1000  
10    cout<<"B中的a = "<<A::B::a<<endl; //2000  
11 }
```

8.14.5.5 命名空间是开放的，即可以随时把新的成员加入已有的命名空间中(常用)

```
1 namespace A {  
2     int a = 100;  
3     int b = 200;  
4 }  
5 //将c添加到已有的命名空间A中  
6 namespace A {  
7     int c = 300;  
8 }  
9 void test04()  
10 {  
11     cout<<"A中a = "<<A::a<<endl;//100  
12     cout<<"A中c = "<<A::c<<endl;//200  
13 }
```

8.14.5.6 命名空间 可以存放 变量 和 函数

```
1 namespace A {  
2     int a=100;//变量  
3  
4     void func()//函数  
5     {  
6         cout<<"func遍历a = "<<a<<endl;  
7     }  
8 }  
9 void test05()
```

```
10 {
11     //变量的使用
12     cout<<"A中的a = "<<A::a<<endl;
13
14     //函数的使用
15     A::func();
16 }
```

8.14.5.7 命名空间中的函数 可以在“命名空间”外 定义

```
1 namespace A {
2     int a=100;//变量
3
4     void func();
5 }
6
7 void A::func()//成员函数 在外部定义的时候 记得加作用域
8 {
9     //访问命名空间的数据不用加作用域
10    cout<<"func遍历a = "<<a<<endl;
11 }
12
13 void funb()//普通函数
14 {
15     cout<<"funb遍历a = "<<A::a<<endl;
16 }
17 void test06()
18 {
19     A::func();
20     funb();
21 }
```

无名命名空间，意味着命名空间中的标识符只能在本文件内访问，相当于给这个标识符加上了static，使得其可以作为内部连接（了解）

```
1 namespace{
2     int a = 10;
3     void func(){
4         cout<<"hello namespace"<<endl;
5     }
6 }
7 void test(){
8
9     //只能在当前源文件直接访问a 或 func
10    cout<<"a = "<<a<<endl;
11    func();
12 }
```

8.15 对象数组

声明方式1：

```
1 | circle ca1[10];
```

声明方式2：用匿名对象构成的列表初始化数组

```
1 | auto ca2[3]={circle{3},circle{},circle{5}};
```

声明方式3：使用C++11列表初始化，列表成员为隐式构造的匿名对象

```
1 | circle ca3[3]{3.1,{},5};
2 | circle ca4[3]={3.1,{},5};
```

声明方式4：用new在堆区生成对象数组

```
1 | auto* p1=new circle[3];
2 | auto p2=new circle[3]{3.1,{},5};
3 | delete []p1;
4 | delete []p2;
5 | p1=p2=nullptr;
```

9. 运算符重载

本章展示了如何使C++的运算符能够处理类对象--一个称为运算符重载的过程。当运算符作用于类类型的运算对象时，可以通过运算符重载重新定义该运算符的含义。明智的使用运算符重载能令我们的程序更易于编写和阅读。

9.1 使用标准类库string中的重载操作符

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main(){
6     string s1{"happy"};
7     string s2{"birthday"};
8     string s3;// creates an empty string
9
10    cout<<"s1 is \"<<s1<< "\"; s2 is \"<<s2
11    <<"\"; s3 is \"<<s3<<'\""
12    <<"\n\nThe results of comparing s2 and s1:"<<boolalpha
13    <<"\ns2 == s1 yields "<<(s2==s1)
14    <<"\ns2 != s1 yields "<<(s2!=s1)
15    <<"\ns2 > s1 yields "<<(s2>s1)
16    <<"\ns2 < s1 yields "<<(s2<s1)
17    <<"\ns2 >= s1 yields "<<(s2>=s1)
18    <<"\ns2 <= s1 yields "<<(s2<=s1);
19
20    cout<<"\n\ntesting s3.empty():\n";
21
22    if(s3.empty()){
23        cout<<"s3 is empty; assigning s1 to s3;\n";
24        s3=s1;
25        cout<<"s3 is \"<<s3<<\"";
26    }
27
28    cout << "\n\ns1 += s2 yields s1 = ";
29    s1 += s2; // test overloaded concatenation
30    cout << s1;
31
32    cout << "\n\ns1 += \" to you\" yields\n";
33    s1 += " to you";//39行
34    cout << "s1 = " << s1;
```

```
35
36     // test string concatenation with a C++14 string-
37     // object literal
38     cout << "\n\ns1 += \\", have a great day!\" yields\n";
39     s1 += ", have a great day!"s;
40     cout << "s1 = " << s1 << "\n\n";
41
42     cout << "The substring of s1 starting at location 0
43 for\n"
44     <<"14 characters, s1.substr(0, 14), is:\n"
45     <<s1.substr(0, 14)<<"\n\n";
46
47     cout << "The substring of s1 starting at\n"
48     <<"location 15, s1.substr(15), is:\n"<<s1.substr(15)
49     <<"\n";
50
51     // test copy constructor
52     string s4{s1};
53     cout << "\n\ns4 = " << s4 << "\n\n";
54
55     // test overloaded copy assignment (=) operator with
56     // self-assignment
57     cout << "assigning s4 to s4\n";
58     s4=s4;
59     cout << "s4 = " << s4;
60
61     s1[0] = 'H';
62     s1[6] = 'B';
63     cout << "\n\ns1 after s1[0] = 'H' and s1[6] = 'B'
64     is:\n"
65     <<s1<<"\n\n";
66
67     try{
68         cout<<"attempting to assign 'd' to s1.at(100)
69         yields:\n";
70         s1.at(100)='d';
71     }
72     catch(out_of_range& x){
73         cout<<"An exception occurred: " << x.what() <<
74         endl;
75     }
76 }
```

运行结果如下：

```
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 > s1 yields false
s2 < s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
s1 = happy birthday to you

s1 += ", have a great day!" yields
s1 = happy birthday to you, have a great day!

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at
location 15, s1.substr(15), is:
to you, have a great day!

s4 = happy birthday to you, have a great day!

assigning s4 to s4
s4 = happy birthday to you, have a great day!

s1 after s1[0] = 'H' and s1[6] = 'B' is:
Happy Birthday to you, have a great day!

Attempt to assign 'd' to s1.at(100) yields:
An exception occurred: invalid string position
```

s3使用默认的string构造函数创造空字符串。

used the stream manipulator boolalpha to set the output stream to display condition values as the strings "false" and "true".

string的成员函数，如果字符串为空，则返回true,反之，返回false。

string的成员函数substr,返回string对象的一部分。

函数 at() 在使用时会检查下标是否有效。如果给定的下标超出字符的长度范围，系统会抛出 out_of_range 异常。

1.如果at函数被non-const对象调用，此函数会返回一个可修改的左值，可以用在赋值操作符的左侧，可以赋给那个位置新的值。

2.如果at函数被const string对象调用，函数会返回一个不可修改的左值。只能获得那个值，而不能修改。

9.1.1 string类中的函数

[c++ string 的常用库函数的用法_c++ string arg-CSDN博客](#)

9.1.2 运算符与函数

1. Special Operators Usage with Objects (与对象一起用的运算符)

1.1. string类：使用“+”连接两个字符串

```
1 string s1("Hello"), s2("World!");
2
3 cout << s1 + s2 << endl;
```

1.2. array 与 vector类：使用[] 访问元素

```
1 array<char, 3> a{};
2
3 vector<char> v(3, 'a'); //'a', 'a', 'a'
4
5 a[0] v[1] = 'b';
```

1.3. path类：使用“/”连接路径元素

```
1 std::filesystem::path p**{};**
2
3 p = p / "c:" / "Users" / "cyd";
```

2. The operator vs function (运算符与函数的异同)

2.1. 运算符可以看做是函数

2.2. 不同之处

2.2.1. 语法上有区别

```
1 3 * 2          //中缀式
2
3 * 3 2          //前缀式
4
5 multiply(3,2); //前缀式
6
7 3 2 *          //后缀式(RPN)
```

2.2.2. 不能自定义新的运算符 (只能重载)

```
1 | 3 ** 2      // C/C++中错误
2 |
3 | pow(3, 2) // 3的平方
```

2.2.3. 函数可overload, override产生任何想要的结果，但运算符作用于内置类型的行为不能修改

```
1 | multiply (3, 2) // 可以返回1
2 |
3 | 3 * 2           // 结果必须是6
```

2.3. 函数式编程语言的观念

一切皆是函数

Haskell中可以定义新的运算符

9.2 运算符重载基础

9.2.1 运算重载不是自动的

必须编写运算符重载函数才可以执行所需的操作。运算符通过向常一样编写non-static成员函数或者非成员函数来重载。只是函数名称以operator开头，后跟重载运算符的符号。例如，函数名为operator+将用于重载运算符+，以便用于特定用户定义类型的对象。当运算符作为成员函数重载时，它们必须是non-static，因为它们必须在类的某个对象上被调用，并在该对象上进行操作。

9.2.2 不需要进行重载的操作符

为了在一个类的对象上使用一个运算符，必须为该类定义重载的运算符函数-除了三个例外：

1.赋值操作符(=)，在大多数的类中用来数据成员的赋值。对于带指针成员的类，成员指派是危险的，因此我们将显式地重载此类类的指派运算符。

2.取地址操作符(&)

3.逗号操作符(,)

9.2.3 不能进行重载的运算符

Operators that cannot be overloaded

. .* (pointer to member) :: ?:

sizeof运算符也不能重载。

#运算符不可以重载

9.2.4 运算符重载的规则和限制

当你准备为自己的类重载运算符时，有几个规则和限制是你应该牢记的：

1.重载不会改变操作符的优先级

2.操作符的结合性不会因为重载而改变

3.An operator's "arity" (that is, the number of operands an operator takes) cannot be changed

4.只有现存的操作符可以重载

5.不可以重载操作符以改变操作符对基本类型变量的作用，例如，不能让+表示两个整数相减。操作符重载仅仅工作在用户自定义的类的对象或者（用户自定义类型的对象和基本类型对象的混合）。**运算符作用于C++内部提供的数据类型时，原来含义保持不变**

6.像+和+=，必须单独重载

7.当重载(),[],->或者任何赋值操作符时，操作符重载函数必须声明为类的成员。其他的情况，操作符重载函数可以是成员函数或者非成员函数。

表 11.1

可被重载的操作符

+	-	*	/	%	^
&		~=	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	=	<<	>>	>>=
<<=	==	!=	<=	>=	&&
	++	--	,	->*	->
0	[]	new	delete	new []	delete []

8.不能创造新的运算符

9.2.5 运算符重载函数的格式

重载的运算符是具有特殊名字的函数：他们的名字由关键字operator和其后要定义的运算符号共同组成。和其他函数一样，重载的运算符也包括返回类型、参数列表以及函数体。

1. 运算符重载函数作为类的成员函数

```
1 | 函数类型 operator 重载运算符(形参列表){  
2 |     函数体;  
3 | } //形参个数=原操作数个数-1(后置++、--除外)
```

双目运算符重载规则

- ◇ 如果要重载 B 为类成员函数，使之能够实现表达式 oprd1 B oprd2，其中 oprd1 为 A 类对象，则 B 应被重载为 A 类的成员函数，形参类型应该是 oprd2 所属的类型。
- ◇ 经重载后，表达式 oprd1 B oprd2 相当于 oprd1.operator B(oprd2)

前置单目运算符重载规则

- ◇ 如果要重载 U 为类成员函数，使之能够实现表达式 U oprd，其中 oprd 为 A 类对象，则 U 应被重载为 A 类的成员函数，无形参。
- ◇ 经重载后，表达式 U oprd 相当于 oprd.operator U()

后置单目运算符 ++ 和 -- 重载规则

- ◇ 如果要重载 ++ 或 -- 为类成员函数，使之能够实现表达式 oprd++ 或 oprd--，其中 oprd 为 A 类对象，则 ++ 或 -- 应被重载为 A 类的成员函数，且具有一个 int 类型形参。
- ◇ 经重载后，表达式 oprd++ 相当于 oprd.operator ++(0)

2. 运算符重载函数作为类的友元函数

```
1 friend 函数类型 operator 重载运算符(形参列表){  
2     函数体;  
3 }
```

3. 运算符重载为非成员函数

函数的形参代表自左向右排列的各操作数。

重载为非成员函数时，参数个数=原操作数个数(后置++、--除外)，至少应该有一个自定义类型的参数。

后置单目运算符++和--的重载函数，形参列表中要增加一个int，但不必写形参名。

如果在运算符的重载函数中需要操作某类对象的私有成员，可以将此函数声明为该类的友元

运算符重载为非成员函数的规则

- ◊ 双目运算符 B重载后，
表达式 oprd1 B oprd2
等同于operator B(oprd1,oprd2)
- ◊ 前置单目运算符 B重载后，
表达式 B oprd
等同于operator B(oprd)
- ◊ 后置单目运算符 ++和--重载后，
表达式 oprd B
等同于operator B(oprd,0)

“函数类型”指出重载运算符的返回值类型，operator是定义运算符重载函数的关键词，“重载运算符”指出要重载的运算符名字，是C++中可重载的运算符，比如要重载加法运算符，这里直接写“+”即可，“形参表”指出重载运算符所需要的参数及其类型。

9.3 重载一元运算符

9.3.1 重载++和--运算符

“++”和“-”重载运算符也有前缀和后缀两种运算符重载形式，以“++”重载运算符为例，其语法格式如下：

1 | 函数类型 `operator ++ ()`

1 | 函数类型 `operator ++(int)`

例如下述代码：

```
1 #include<iostream>
2 using namespace std;
3
4 class MyClass2
5 {
6 public:
7     MyClass2(int i){ n = i; }
8     int operator ++(){ n++; return n; }
9     int operator ++(int){ n += 2; return n; }
10    void display()
11    {
12        cout << "n=" << n << endl;
13    }
14 private:
15     int n;
16 };
17
18 int main()
19 {
20     MyClass2 A(5), B(5);
21     A++;
22     ++B;
23     A.display();
24     B.display();
25     system("pause");
26 }
27
```

```
D:\clionproject\test_10_24\cmake-build-debug\test_10_24.exe
n=7
n=6
请按任意键继续. . .
```

9.3.2 重载->运算符

“->”运算符是成员访问运算符，这种单目运算符只能被重载为成员函数，一般成员访问运算符的格式如下：

```
1 | 对象->成员
```

成员访问运算符“->”函数重载的一般形式为：

```
1 | 数据类型 类名::operator->();
```

例如下述代码：

```
1 #include <iostream>
2 using namespace std;
3
4 class PClass
5 {
6     int n; double m;
7 public:
8     PClass *operator->()
9     {
10         return this;
11     }
12     void setvalue(int n1, double m1)
13     {
14         n = n1; m = m1;
15     }
16     void disp()
17     {
18         cout<< "n=" << n << ",m=" << m<<endl;
19     }
20 }
21
22 int main()
23 {
24     PClass s;
```

```
25     s->setvalue(10, 20.5);
26     s->disp();
27     s.setvalue(20, 89.8);
28     s.disp();
29 }
```

```
D:\clionproject\test_10_24\cmake-build-debug\test_10_24.exe
n=10,m=20.5
n=20,m=89.8
```

上述程序中，重载“->”运算符的成员函数，该函数返回当前对象的指针。从而导致“s->disp();”和“s.disp();”两个语句都是正确的，实际上，前者通过调用重载“->”运算符成员函数转换成后者的格式。

9.4 重载二元操作符

二元操作符可以重载为含有一个形参的non-static成员函数或者重载为含两个形参(其中一个形参必须是类的对象或者类的对象的引用)的非成员函数。一个非成员操作符函数经常因为性能原因被声明为类的friend。

9.4.1 二元操作符重载函数(non static成员函数)

考虑使用<比较自行定义的属于string类的两个对象。当将二元运算符<重载为non-static成员函数，如果y和z是string类的对象，那么y<z会被编译器看作y.operator <(z)，y通过this指针传递，z通过参数传递。例如下列代码，重载<运算符：

```
1 class String{
2 public:
3     bool operator<(const String&) const
4     ...
5 };
```

只有当二元操作符的左操作数是类的对象，二元操作符的重载函数才可以是类的成员函数。

若要将二元运算符函数声明为非静态成员函数，您必须用以下形式声明它：

```
ret-type operator op(arg)
```

其中，ret-type 是返回类型，op 是上表中列出的运算符之一，而 arg 必须是类的对象。

9.4.2 二元操作符重载函数(非成员函数)

若要将二元运算符函数声明为全局函数，您必须用以下形式声明它：

```
ret-type operator op(arg1,arg2)
```

其中，ret-type 和 op 是成员运算符函数，而 arg1 和 arg2 是自变量。至少有一个参数必须是类的对象(或者是对象的引用)。

对二元运算符的返回类型没有限制；但是，大多数用户定义的二元运算符将返回类类型或对类类型的引用。

如果y和z都是类的对象或者对象的引用，编译器会看作operator<(y,z)

```
1 | bool operator<(const string&, const string&);
```

9.5 重载二元流插入和提取操作符

```
1 #ifndef TEST_10_24_PHONENUMBER_H
2 #define TEST_10_24_PHONENUMBER_H
3
4 #include <iostream>
5 #include <string>
6
7 class PhoneNumber{
8     friend std::ostream& operator<<(std::ostream&, const
9     PhoneNumber&);
10    friend std::istream& operator>>
11    (std::istream&, PhoneNumber&);
12
13 private:
14     std::string areaCode;
15     std::string exchange;
16     std::string line;
17 };
18 #endif //TEST_10_24_PHONENUMBER_H
```

```
1 // 
2 // Created by 22364 on 2023/10/24.
3 //
4 #include <iomanip>
5 #include "PhoneNumber.h"
6 using namespace std;
7
```

```

8 ostream& operator<<(ostream& output, const PhoneNumber&
9   number){
10   output << "Area code: " << number.areaCode <<
11   "\nExchange: "
12   <<number.exchange << "\nLine: " << number.line << "\n"
13   <<"(" << number.areaCode << ")" " << number.exchange <<
14   "_"
15   <<number.line << "\n";
16   return output;// enables cout << a << b << c;
17 }
18 // overloaded stream extraction operator; cannot be a
19 member function
20 // if we would like to invoke it with cin >>
21 somePhoneNumber;
22 istream& operator>>(istream& input, PhoneNumber& number){
23   input.ignore();
24   input >> setw(3) >> number.areaCode;
25   input.ignore(2);
26   input >> setw(3) >> number.exchange;
27   input.ignore();
28   input >> setw(4) >> number.line;
29   return input;
30 }
```

```

1 #include <iostream>
2 #include "PhoneNumber.h"
3 using namespace std;
4
5 int main(){
6   PhoneNumber phone;
7
8   cout<<"Enter phone number in the form (555) 555-5555:"<<endl;
9
10  // cin >> phone invokes operator>> by implicitly
11  // issuing
12  // the non-member function call operator>>(cin, phone)
13  cin>>phone;
14
15  // cout << phone invokes operator<< by implicitly
16  // issuing
```

```
15     // the non-member function call operator<<(cout,
16     cout<<phone<<endl;
17 }
```

在第二张图中，对于<<的重载函数的定义，当在main函数中调用此函数后，引用形参input就变成了cin的alias，引用参数number就变成了phone的alias.由于两个函数都被声明为类的friend,所以可以访问类的private成员。流操作符setw限制了读入string的字符。When used with cin and strings, setw restricts the number of characters read to the number of characters specified by its argument (i.e., setw(3) allows three characters to be read).调用istream的成员函数ignore可以略过括号、空格和破折号。

dash characters---破折号

```
1 | cin>>phone
```

相当于：

```
1 | operator>>(cin,phone)
```

```
1 | cout<<phone
```

相当于：

```
1 | operator<<(cout,phone)
```

operator<<和operator>>函数都被声明为非成员friend函数，他们是非成员函数是因为PhoneNumber类的对象必须是操作符的右操作数。**二元操作符的重载函数只有在左操作符是类的对象时才可以成为类的成员函数。Overloaded operator functions for binary operators can be member functions only when the left operand is an object of the class in which the function is a member.**

成员函数重载：可以通过this指针访问本类的成员，可以少写一个参数，但是表达式左边的第一个参数必须是类对象，通过该类对象来调用成员函数。即表达式左侧的左侧操作数就是对象本身。例如对于cout<<classobject, classobject是用户自定义的类的对象，通过该类调用成员函数，即使用cout对象调用operator<<函数，这样的话，operator<<需要是cout对象从属的ostream类的成员函数，但是ostream类属于C++的标准库类，不允许修改。因此，如果右操作数是用户自定义的类的对象，必须将<<重载为非成员函数。

9.6 重载一元操作符

一元操作符可以重载为没有参数的non-static成员函数或者一个参数的非成员函数。此参数必须是对象或者对象的引用。成员函数必须是non-static的才可以访问类的每个形的non-static成员。

9.6.1 重载一元操作符函数作为成员函数

考虑重载一元操作符`!`，以此检验输入的string类的对象是否为空。这个函数会返回一个bool值。例如将操作符`!`重载函数重载为成员函数(没有参数)，对于对象`s`,`s!`,编译器会将之认为`s.operator!()`，操作数`s`是string类的对象，`operator!`为string类的成员函数。

```
1 | class String{  
2 | public:  
3 |     bool operator!() const;  
4 |     ...  
5 | };
```

9.6.2 一元操作符重载函数为非成员函数

还是考虑string类的对象`s`,`s`必须是类的对象或者对象的引用。`!s`会被编译器认为是：`operator!(s)`

```
1 | bool operator!(const string&);
```

9.7 重载`++`和`--`

为了重载`++`和`--`操作符，编译器需要区分操作符是操作数的前缀还是后缀，因此重载函数必须有明显的特征以供编译器区分。

9.7.1 前缀递增操作符重载

考虑将整数1加到Date类的对象`d1`上。如果重载函数被定义为成员函数，编译器会生成成员函数调用：

```
1 | d1.operator++();
```

此成员函数的原型为：

```
1 | Date& operator++(); //类名后的&必不可少
```

如果重载函数被声明为非成员函数，编译器会生成函数调用：

```
1 | operator++(d1);
```

此函数的函数原型为：

```
1 | Date& operator++(Date&);
```

9.7.2 后缀递增操作符重载

为了与前缀进行区分，对于d1++，编译器会生成成员函数调用：

```
1 | d1.operator ++ (0); //0只用来区分前缀和后缀操作
```

函数原型为：

```
1 | Date operator ++ (int); //在实现此函数时，不应该使用括号内的整型  
形参
```

参数0是一个虚职，为了让编译器区分前缀和后缀。如果后缀操作符重载函数为非成员函数，当编译器遇到d1++，会生成函数调用：

```
1 | operator++(d1, 0)
```

函数原型为：

```
1 | Date operator++(Date&, int);
```

x++最终返回的是临时变量的值，而临时变量的值不能作为左值。x++是右值，编译器会先生成一份x值的临时复制，然后再对x递增，最后返回临时复制内容。++x不同，是对x递增后马上返回其自身，所以++x是左值。

The extra object that's created by the postfix increment (or decrement) operator can result in a performance problem—especially when the operator is used in a loop. For this reason, you should prefer the overloaded prefix increment and decrement operators.

leap year---闰年

Wrap-around Error：当值的增量超过其类型的最大值时，就会发生环绕错误，进而导致“环绕”到非常小、负值或未定义的值。

helpIncrement函数用来防止出现上述错误。

9.8 动态内存管理(Dynamic Memory Management)和nullptr

可以控制内置或者用户自定义类型的对象或者数组的内存的分配和释放。这就叫做动态内存管理，使用操作符new和delete实现。可以使用new操作符在程序执行期间动态的分配精确数量的内存来存储对象或者数组。对象或内置数组在自由存储区(free store)创造，自由存储区是内存中分配给程序用来存储对象的区域。**new和delete都是运算符，不是库函数，不需要单独添加头文件。**

9.8.1 free store VS heap

[C++ 自由存储区是否等价于堆？ - melonstreet - 博客园 \(cnblogs.com\)](#)

当我问你C++的内存布局时，你大概会回答：

“在C++中，内存区分为5个区，分别是堆、栈、自由存储区、全局/静态存储区、常量存储区”。

如果我接着问你自由存储区与堆有什么区别，你或许这样回答：

“malloc在堆上分配的内存块，使用free释放内存，而new所申请的内存则是在自由存储区上，使用delete来释放。”

这样听起来似乎也没错，但如果我接着问：

自由存储区与堆是两块不同的内存区域吗？它们有可能相同吗？

你可能就懵了。

事实上，我在网上看的很多博客，划分自由存储区与堆的分界线就是new/delete与malloc/free。然而，尽管C++标准没有要求，但很多编译器的new/delete都是以malloc/free为基础来实现的。那么请问：借以malloc实现的new，所申请的内存是在堆上还是在自由存储区上？

从技术上来说，堆 (heap) 是C语言和操作系统的术语。堆是操作系统所维护的一块特殊内存，它提供了动态分配的功能，当运行程序调用malloc()时就会从中分配，稍后调用free可把内存交还。而自由存储是C++中通过new和delete动态分配和释放对象的抽象概念，通过new来申请的内存区域可称为自由存储区。基本上，所有的C++编译器默认使用堆来实现自由存储，也即是缺省的全局运算符new和delete也许会按照malloc和free的方式来被实现，这时藉由new运算符分配的对象，说它在堆上也对，说它在自由存储区上也正确。但程序员也可以通过

重载操作符，改用其他内存来实现自由存储，例如全局变量做的对象池，这时自由存储区就区别于堆了。我们所需要记住的就是：

堆是操作系统维护的一块内存，而自由存储是C++中通过new与delete动态分配和释放对象的抽象概念。堆与自由存储区并不等价。

一旦内存分配完毕，就可以通过new返回的指针访问它。当不再需要此内存，就可以通过delete将它返回到自由存储区以释放内存。

9.8.2 使用new获取动态内存

new 运算符的用法如下：

```
1 Time* timePtr{new Time};  
2 new <类型名>(初值); //申请一个变量的空间  
3 new <类型名>[常量表达式]; //申请数组
```

Time是任意类型名，timePtr是类型为Time*的指针。这样的语句会动态分配出一片大小为 sizeof(Time) 字节的内存空间，并且将该内存空间的起始地址赋值给 timePtr。如果要求分配的空间太大，操作系统找不到足够的内存来满足，那么动态内存分配就会失败，此时程序会抛出异常，或者返回空指针nullptr。

例如：

```
1 int* p;  
2 p = new int;  
3 *p = 5;  
4  
5 int *aPtr=new int[8];  
6 int* arr1 = new int[5];  
7 int* nums1 = new int[5] { 1, 3, 2, 5, 4 };
```

第二行动态分配了一片 4 个字节大小的内存空间，而 p 指向这片空间。通过 p 可以读写该内存空间。

new运算符的初始化

```
1 int* buffer = new int{}; // 初始化为0  
2 int* buffer = new int{0}; // 初始化为0  
3 char* s=new char('a');  
4 int* buffer = new int[512]{}; // 512个int都初始化为0  
5 int* buffer = new int{5}; // 初始化为5
```

动态内存使用完毕以后，要使用delete运算符释放。销毁一个动态分配的对象并释放对象的内存，使用delete操作符：

```
1 | delete <指针名>; //删除一个变量/对象
2 | delete []<指针名>; //删除一个数组空间
```

不能删除未被new操作符分配的内存。这样做会导致未定义的行为。

删除一个动态分配的内存块后，一定不要再删除同一个块。防止这种情况的一种方法是立即将指针设置为nullptr。删除nullptr没有影响。

nullptr作为空指针

```
1 | int* q=nullptr;
```

C++保证用删除数组的形式删除非数组的内存分配是安全的，如下例：

```
1 | char* p=new char(32);
2 | //可以使用下列两种方式删除
3 | delete p;
4 | delete [] p;
```

Operators new and delete can be overloaded, but this is beyond the scope of the book. If you do overload new, then you should overload delete in the same scope to avoid subtle dynamic memory management errors.

new和delete操作符也可以重载。如果重载了new，就要同时重载delete。

动态分配出来的内存没有变量名，只能通过指向它的指针来操作内存中的数据。

动态分配的内存生命周期和程序相同，程序退出时，如果没有释放，系统会自动回收。

当动态分配的内存不再需要时，不释放动态分配的内存会导致系统过早耗尽内存。这有时被称为“内存泄漏”

当类的对象包含指向动态分配的内存的指针时，不提供拷贝构造函数以及重载赋值操作符是一个潜在的错误。

9.8.3 在堆中创建对象

在堆中创建对象，由程序员控制对象的生存期。

```
1 | className *pobject=new className{}; //用无参构造函数创建对象
2 | className *pobject=new className{arguments}; //用有参构造函数
   | 创建对象
```

9.9 拷贝构造函数

The null terminated strings are basically a sequence of characters, and the last element is one null character (denoted by '\0').

对于动态分配内存的内置数组，`range-based for`不能正常工作

当类的对象包含指向动态分配的内存的指针时，不提供拷贝构造函数以及重载赋值操作符是一个潜在的错误。

9.9.1 拷贝构造函数

它是一种特殊的构造函数，它在创建对象时，是使用同一类中之前创建的对象来初始化新创建的对象。**拷贝构造函数**就是函数名是当前类的名字，参数为当前类的另一个对象的函数，拷贝构造函数通常用于：

- 通过使用另一个同类的对象来初始化新创建的对象。
- 复制对象把它作为参数传递给函数。
- 复制对象，并从函数返回这个对象。

声明拷贝构造函数：

```
1 | Circle (Circle&);
2 | Circle (const Circle&);
```

如果在类中没有定义拷贝构造函数，编译器会自行定义一个。编译器自定义生成的拷贝构造函数可以实现对应数据成员的复制。自定义的拷贝构造函数可以实现特殊的复制功能。当类中拥有指针类型的成员变量时，拷贝构造函数中需要以深拷贝（而非浅拷贝）的方式复制该指针成员。

The argument to a copy constructor should be a `const` reference to allow a `const` object to be copied.

如果不希望对象被复制构造，可以使用"=delete"指示编译器不生成默认拷贝构造函数。例如：

```
1 class Point{  
2     public:  
3         Point(int xx=0,int yy=0){x=xx;y=yy;}//构造函数  
4         Point(const Point& p)=delete;  
5     private:  
6         int x,y//私有数据  
7 }
```

9.9.2 浅拷贝和深拷贝

1. **浅拷贝**：拷贝指针，而非指针指向的内容。简单的复制拷贝操作。

以下两种情况会出现浅拷贝：

1. 创建新对象时，调用类的隐式/默认构造函数。

2. 为已有对象赋值时，使用赋值运算符。

2. **深拷贝**：拷贝指针指向的内容。在堆区重新申请内存空间，进行拷贝操作。当被复制的对象数据成员是指针类型时，不是复制该指针成员本身，而是将指针所指对象进行复制。

而使用移动语义和移动构造函数，可以避免深拷贝的情况，节省资源。

如何深拷贝：

1. 自行编写拷贝构造函数，不使用编译器隐式生成的拷贝构造函数

2. 重载赋值运算符，不使用编译器隐式生成的赋值运算符函数

9.9.3 何时执行拷贝构造函数

1. 用已有对象构造新对象的时候(通过使用另一个同类型的对象来初始化新创建的对象)

```
1 Student stu;//先定义一个对象  
2 Student stu2(stu);//(1)会调用拷贝构造函数  
3 Student stu3=stu ;//(2)会调用拷贝构造函数，这里和(1)是一样的，  
    仅仅是写法不同而已
```

如果对象已经被创建，会调用赋值操作符而不是复制构造函数：

```
1 Student stu;
2
3 Student stu2;//在这里被创建
4
5 stu2 = stu;//这里调用赋值操作符重载（见后文）
```

代码示例：

```
1 #include <iostream>//cin cout
2 using namespace std;
3
4 class Student
5 {
6 public:
7     //规则1：如果类的用户自己定义了构造函数
8     //，编译器就不会自动合成类的默认构造函数Student(void)
9     //这样类就不存在默认构造函数了
10
11     //类的用户自己定义了拷贝构造函数（拷贝构造函数也算是一个构造函数）
12     //按照规则1，此时类不再拥有Student(void)这个函数
13     Student(const Student& from)
14     {
15         cout << "copy constructor called." << endl;
16     }
17
18     //由于上面已经定义了一个构造函数，按照规则1，
19     //Student stu;这条语句就会因为类不存在默认构造函数而编译报错
20     //为了能够让我们可以写Student stu;这条语句来创建对象
21     //，我们在这里显示定义类的默认构造函数
22     Student(void)
23     {
24         cout << "default constructor called." << endl;
25     }
26 };
27
28 int main(int argc, char* argv[])
29 {
30     cout << "flag1" << endl;//打印标记信息，用来查看函数执行的
31     //顺序
32     Student stu;//先定义一个对象
```

```

32     cout << "flag2" << endl; //打印标记信息，用来查看函数执行的
33    顺序
34     Student stu2(stu); // (1)会调用拷贝构造函数
35     cout << "flag3" << endl; //打印标记信息，用来查看函数执行的
36    顺序
37
38     return 0;
39 }

```



```

Microsoft Visual Studio 调试控制台
flag1
default constructor called.
flag2
copy constructor called.
flag3
copy constructor called.
flag4

```

2. 给函数传递值类型参数的时候(复制对象把它作为参数传递给函数): 调用函数时, 将使用实参对象初始化形参对象, 发生拷贝构造。

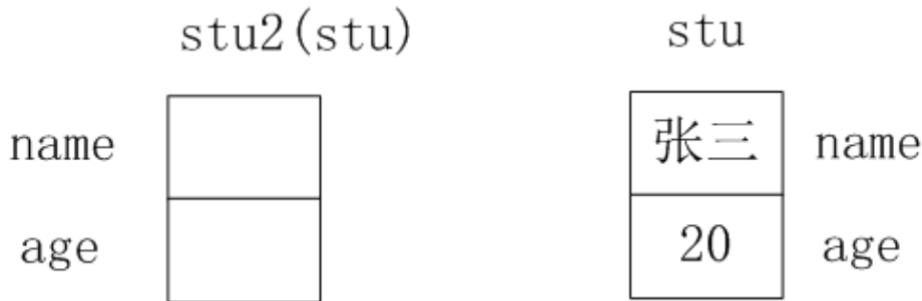
```

1 void test_function(Student s)
2 { // s在该函数被调用的时候创建, 该函数执行完之后释放
3     s.m_name = "李四"; // 修改s的名字
4 }
5 int main()
6 {
7     Student stu("张三");
8     test_function(stu); // (1) 创建stu的副本, 函数内使用副本
9     cout << stu.m_name; // 还是输出“张三”, 因为修改的是, 函数内的副本
10 }

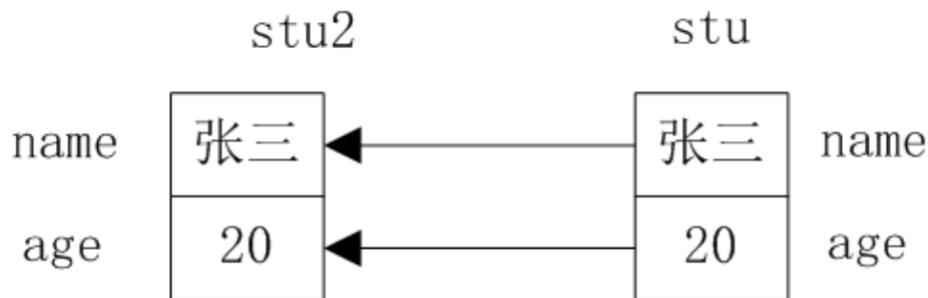
```

拷贝对象发生了什么

1 先构造对象：分配内存空间



2 对应成员赋值：拷贝数据



完整代码：

```
1 #include <iostream> //cin cout
2 #include <string>
3 using namespace std;
4
5 class Student
6 {
7 public:
8     Student(const Student& from) //拷贝构造函数
9     {
10         cout << "copy constructor called." << endl;
11     }
12     //如果只提供上面的拷贝构造函数，编译器就不再生成默认构造函数
13     //，会导致类对象就不可以直接创建，所以还需要提供一个默认构造函数
14     Student(void)
15     {
16         cout << "default constructor called." << endl;
17     }
18     Student(const string& name) :m_name(name)
19     {
20         cout << "string constructor called." << endl;
21     }
22 }
```

```
22     string m_name;//存放学生姓名
23 };
24 void test_function(Student s)
25 {//s在该函数被调用的时候创建, 该函数执行完之后释放
26     cout << "flag2" << endl;
27     s.m_name = "李四";//修改s的名字
28     cout << "flag3" << endl;
29 }//s释放的时刻
30 int main()
31 {
32     cout << "flag0" << endl;
33     Student stu("张三");
34     cout << "flag1" << endl;
35     test_function(stu);//(1)创建stu的副本, 函数内使用副本
36     cout << "flag4" << endl;
37     cout << stu.m_name;//还是输出“张三”, 因为修改的是, 函数内的副本
38 }
```

```
Microsoft Visual Studio 调试控制台
flag0
string constructor called.
flag1
copy constructor called.
flag2
flag3
flag4
张三
```

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Student {
6 public:
7     Student(); // default构造函数
8     Student(const Student& obj); // 拷贝构造函数
9
10    int getNumber();
11
12 private:
13    int number;
14 };
15
16 // 定义默认构造函数
17 Student::Student(){
```

```

18     this->number = 0;
19     cout << "default constructor" << endl;
20 }
21
22 // 定义拷贝构造函数
23 Student::Student(const Student& obj) {
24     this->number = obj.number;
25     cout << "copy constructor" << endl;
26 }
27
28 int Student::getNumber() {
29     return this->number;
30 }
31
32 // 输出某个对象的number
33 void showNumber(Student a) {
34     cout << a.getNumber();
35 }
36
37 int main(){
38     Student s; // 调用默认构造函数
39     showNumber(s);
40
41     return 0;
42 }

```

你认为输出结果会是什么？首先肯定会输出一个"default constructor"，因为s的默认构造函数输出了该字符串。那么，showNumber()会输出什么呢？正确的输出是：

```

1 default constructor
2 copy constructor
3 0

```

这就回到了我们开头的那句话，**拷贝构造函数定义了一个对象如何以值传递给函数**。

在函数showNumber()中，参数a是以值传递的方式传入的。所以主函数中调用showNumber()时，对象s被复制到了形参a当中。这个复制操作是通过调用了**拷贝构造函数完成的**，所以调用了自身的拷贝构造函数，自然会输出一遍"copy constructor"。

相当于执行了这样一个操作

```

1 Student a(s);

```

等待showNumber()结束之后，再析构a这个对象。

这也就是为什么传递用户自定类型通常用引用传递，同时也解释了，为什么拷贝构造函数传参为什么是引用传参？

设想，如果拷贝构造函数这么写，会发生什么？

```
1 | Student (const Student obj);
```

假设有个Student对象s。我值传递s进入了拷贝构造函数，那么按照之前所说，编译器会去申请一个空间给形参obj，同时调用自身的拷贝构造函数把s的数据成员的值传给obj。但是这个调用拷贝构造函数的过程，又是一个值传递，又需要开辟一个空间....

所以，只要以值传递的方式调用拷贝构造函数，就会申请空间，申请空间的过程中又会申请空间，又申请空间的过程又申请空间。如此往复，会陷入无限套娃的死循环。

所以拷贝构造函数一定不能是值传递。

3. 函数返回值类型对象的时候：函数执行完成返回主调函数时，将使用return语句中的对象初始化一个临时无名对象，传递给主调函数，此时发生拷贝构造。

```
1 | Student get_copy(void)
2 | {
3 |     Student s;
4 |     return s; //这里以s为参数构造构造一个副本，并返回副本，这里发生了拷贝
5 | }
```

完整代码：

```
1 | #include <iostream> //cin cout
2 | #include <string>
3 | using namespace std;
4 |
5 | class Student
6 | {
7 | public:
8 |     Student(const Student& from) //拷贝构造函数
9 |     {
10 |         cout << "copy constructor called." << endl;
11 |     }
12 |     //如果只提供上面的拷贝构造函数，编译器就不再生成默认构造函数
13 |     //，会导致类对象就不可以直接创建，所以还需要提供一个默认构造函数
14 |     Student(void)
```

```

15  {
16      cout << "default constructor called." << endl;
17  }
18 };
19 Student test_function(void)
20 {//s在该函数被调用的时候创建，该函数执行完之后释放
21     cout << "flag2" << endl;
22     Student stu;
23     cout << "flag3" << endl;
24     return stu;
25 }//s释放的时刻
26 int main()
27 {
28     cout << "flag1" << endl;
29     test_function();//(1)创建stu的副本，函数内使用副本
30     cout << "flag4" << endl;
31 }

```



```

Microsoft Visual Studio 调试控制台

flag1
flag2
default constructor called.
flag3
copy constructor called.
flag4

```

9.9.4 句柄(handle)

句柄不能是常量，在程序设计中，句柄是一种特殊的[智能指针](#)，当一个[应用程序](#)要引用其他系统（如数据库、操作系统）所管理的内存块或对象时，就要使用句柄。

9.10 Operators as Member vs. Non-Member Functions

当操作符函数为成员函数时，左操作数必须是一个对象（或者是对象的引用）。如果左操作数必须是不同类的对象或者基础类型，则操作符函数必须声明为非成员函数。如果需要非成员函数访问类的private和protected成员，可以将非成员函数声明为类的friend。

9.11 类型转换

对象的类型定义了对象能够包含的数据和能够参与的运算，其中一种运算被大多数类型支持，就是将对象从一种给定的类型转换为另一种相关类型。

当程序的某处我们使用了一种类型而其实对象应该取另一种类型时，程序会自动进行类型转换，当给某种类型的对象强行赋了另一种类型的值时，会发生什么呢？

```
1 bool b=42;//b为真
2 int i=b;//i的值为1
3 i=3.14;//i的值为3
4 double pi=i;//pi的值为3.0
5 unsigned char c=-1;//假设char为8bite,c的值为255
6 signed char c2=256;//假设char占8bite,c2的值是未定义的
```

类型所能表示的值的范围决定了转换的过程：

- 1.当我们把一个整数值赋给浮点类型时，小数部分记为0。如果该整数所占的空间超过了浮点类型的容量，精度可能有损失。
- 2.当我们赋给无符号类型一个超出它表示范围的值时，结果是初始值对无符号类型表示数值总数取模后的余数。例如，8比特大小的unsigned char可以表示0-255区间内的值，如果我们赋了一个区间以外的值，则实际的结果是该值对256取模后所得的余数。因此，把-1赋给8比特大小的unsigned char所得的结果是255。
- 3.当我们赋给带符号类型一个超出它表示范围的值时，结果是未定义的(undefined)。此时，程序可能继续工作、可能崩溃，也可能生成垃圾数据。

当在程序的某处使用了一种算术类型的值而其实所需的是另一种类型的值时，编译器同样会执行上诉的类型转换。例如，我们使用了一个非布尔值作为条件，那么它会被自动的转换成布尔值，这一做法和把非布尔值赋给布尔变量时的操作完全一样：

```
1 int i=42;
2 if(i)//if条件的值将为true
3     i=0;
```

如果i的值为0，则条件的值为false；i的所有非零取值都将使条件为true。如果我们把一个布尔值用在算术表达式里，则它的取值非0即1，所以一般不宜在算术表达式里使用布尔值。

尽管我们不会故意给无符号对象赋一个负值，却可能写出这么做的代码。例如，当一个算术表达式中既有无符号数又有int值时，那个int值就会转换为无符号数。把int转换为无符号数的过程和把int直接赋给无符号变量一样：

```
1 unsigned u=10;
2 int i=-42;
3 std::cout<<i+i<<std::endl;//输出-84
4 std::cout<<u+i<<std::endl;//如果int占32位，输出4294967264
```

在第二个表达式，相加前首先把整数-42转换为无符号数。把负数转换为无符号数类似于直接给无符号数赋一个负值，结果等于这个负数加上无符号数的模。当从无符号数中减去一个值时，不管这个值是不是无符号数，我们都必须确保结果不能是一个负值。

9.12 再探类型转换

类型转换分为隐式类型转换和显式类型转换(强制类型转换)。

9.12.1 隐式类型转换

如果两种类型有关联，那么当程序需要其中一种类型的运算对象时，可以用另一种关联类型的对象或值来替代。即，如果两种类型可以互相转换(conversion)，那么它们就是关联的。

例如,下列表达式的目的是将val初始化为6.

```
1 int val=3.783+3;//编译器可能会警告该运算损失了精度
```

加法的两个运算对象类型不同：3.783为double类型，3为int类型。C++不会直接将两个不同类型的值相加，而是先根据类型转换规则设法将运算对象的类型统一后再求值。上述的类型转换是自动进行的，被称为隐式转换(implicit conversion)。算术类型之间的隐式转换被设计的尽可能避免损失精度。

何时发生隐式类型转换：

- 1.在大多数表达式中，比int类型小的整数值首先提升为较大的整数类型
- 2.在条件中，非布尔值转换为布尔类型
- 3.初始化过程中，初始值转换为变量的类型；复制语句中，右侧运算对象转换成左侧运算对象的类型
- 4.如果算术运算或关系运算的运算对象有多种类型，需要转换成同一种类型
- 5.函数调用时也会发生类型转换

9.12.2 算术转换

算术转换的含义是把一种算术类型转换成另外一种算术类型。

整型提升：负责把小整数类型转换成较大的整数类型。对于bool、char、signed char、unsigned char、short和unsigned short等类型来说，只要他们所有可能的值都能存在int里，他们就会提升为int类型；否则，提升成unsigned int类型。

较大的char类型(wchar_t、char16_t、char32_t)提升成int、unsigned int、long、unsigned long、long long和unsigned long long中最小的一种类型，前提是转换后的类型要能够容纳原类型所有可能的值。

9.12.3 其他隐式类型转换

除了算术转换之外还有几种隐式类型转换：

数组转换成指针：在大多数用到数组的表达式中，数组自动转换成指向数组首元素的指针：

```
1 int ia[10];
2 int* p=ia; //ia转换成指向数组首元素的指针
```

当数组被用作decltype关键字的参数，或者作为取地址符(&)、sizeof及typeid等运算符的运算对象时，上述转换不会发生。如果用一个引用来初始化数组，上述转换也不会发生。

指针的转换：1.常量整数值0或者字面值nullptr能转换成任意指针类型

2.指向任意非常量的指针能转换成void*

3.指向任意对象的指针能转换成const void*

转换成布尔类型：存在一种从算术类型或指针类型向布尔类型自动转换的机制。如果指针或算术类型的值为0，转换结果是false；否则转换结果是true。

```
1 char *cp=get_string();
2 if(cp) //如果指针cp不是0, 条件为真
3 while(*cp) //如果*cp不是空字符, 条件为真
```

转换成常量：允许将指向非常量类型的指针转换成指向相应的常量类型的指针，对于引用也是这样。也就是说，如果T是一种类型，我们就能将指向T的指针或引用分别转换成指向const T的指针或引用。

```
1 int i;
2 const int &j=i;//非常量转换成const int的引用
3 const int *p=&i;//非常量的地址转换成const的地址
4 int &r=j,*q=p;//错误：不允许const转换成非常量
```

相反的转换不存在，因为它试图删掉底层const。

类类型定义的转换：类类型能定义由编译器自动执行的转换，不过编译器每次只能执行一种类类型的转换。如果同时提出多个转换请求，这些请求将被拒绝。

类类型转换的例子：在需要标准库string类型的地方使用c风格字符串；在条件部分读入istream

```
1 string s,t="a value";//字符串字面值转换成string类型
2 while(cin>>s);//while的条件部分把cin转换成布尔值
```

9.12.4 显式转换(强制类型转换)

在 C 语言中，我们大多数是用 (type_name) expression 这种方式来做强制类型转换，但是在 C++ 中，更推荐使用四个转换操作符来实现显式类型转换：

有时我们希望显式的将对象强制转换成另一种类型，例如，在下列代码中执行浮点数除法：

```
1 int i,j;
2 double slope=i/j;
```

就要使用某种方法将i和/或j显式的转换成double,这种方法称作强制类型转换(cast)

一个命名的强制类型转换的格式如下：

```
1 cast-name<type>(expression);
```

type是转换的目标类型，expression是要转换的值。如果type是引用类型，则结果是左值。cast-name是static_cast、dynamic_cast、const_cast和reinterpret_cast中的一种。

I .static_cast:

任何具有明确定义的类型转换，只要不包含底层const，都可以使用static_cast。

```
1 double slope=static_cast<double>(j)/i;
```

当需要把一个较大的算术类型赋值给较小的类型时，`static_cast`非常有用。对于编译器无法自动执行的类型转换也非常有用。

如果要在原生数据类型(基础数据类型)之间进行数据转换，应该使用`static_cast`关键字。

如果涉及到类的话，`static_cast`只能在**有相互联系的类型中进行相互转换**

1.将一个基本类型转换为另一个基本类型，例如将整数转换为浮点数或将字符转换为整数。

```
1 int a = 42;
2 double b = static_cast<double>(a); // 将整数a转换为双精度浮点数b
```

2.指针类型之间的转换

将一个指针类型转换为另一个指针类型，尤其是在**类层次结构中从基类指针转换为派生类指针**。

进行上行转换（把派生类的指针或引用转换成基类表示）是安全的

进行下行转换（把基类的指针或引用转换为派生类表示），由于没有动态类型检查，所以是不安全的

```
1 class Base {};
2 class Derived : public Base {};
3
4 Base* base_ptr = new Derived();
5 Derived* derived_ptr = static_cast<Derived*>(base_ptr); // 将基类指针base_ptr转换为派生类指针derived_ptr
6
```

3.引用类型之间的转换

类似于指针类型之间的转换，可以将一个引用类型转换为另一个引用类型。在这种情况下，也应注意安全性。

```
1 Derived derived_obj;
2 Base& base_ref = derived_obj;
3 Derived& derived_ref = static_cast<Derived&>(base_ref); // 将基类引用base_ref转换为派生类引用derived_ref
```

`static_cast`在编译时执行类型转换，在进行指针或引用类型转换时，需要自己保证合法性。

如果想要运行时类型检查，可以使用`dynamic_cast`进行安全的向下类型转换。

4. 把空指针转换成目标类型的空指针
5. 把任何类型的表达式转换为void类型

注意：static_cast不能转换掉expression的const、volatile或者_unaligned属性。

II. const_cast:

用法：

```
1 | const_cast <new_type> (expression)
```

expression必须是一个指针、引用或者指向对象类型成员的指针。

1. 修改const对象

当需要修改const对象时，可以使用const_cast来删除const属性。

```
1 | const int a = 42;
2 | int* mutable_ptr = const_cast<int*>(&a); // 删除const属性，使得可以
   | 修改a的值
3 | *mutable_ptr = 43; // 修改a的值
```

2. const对象调用非const成员函数

当需要使用const对象调用非const成员函数时，可以使用const_cast删除对象的const属性。

```
1 | class MyClass {
2 | public:
3 |     void non_const_function() { /* ... */ }
4 | };
5 |
6 | const MyClass my_const_obj;
7 | MyClass* mutable_obj_ptr = const_cast<MyClass*>(&my_const_obj); // 
   | 删除const属性，使得可以调用非const成员函数
8 | mutable_obj_ptr->non_const_function(); // 调用非const成员函数
```

不过上述行为都不是很安全，可能导致未定义的行为，因此应谨慎使用。const_cast用于强制去掉这种不能被修改的常数特性，但需要特别注意的是const_cast不是用于去除变量的常量性，而是去除指向常数对象的指针或引用的常量性，其去除常量性的对象必须为指针或引用。

III. reinterpret_cast

reinterpret_cast用于在不同类型之间进行低级别的转换。

expression必须是一个指针、引用、算术类型、函数指针或者成员指针。

首先从英文字面的意思理解，interpret是“解释，诠释”的意思，加上前缀“re”，就是“重新诠释”的意思；cast 在这里可以翻译成“转型”，它仅仅是重新解释底层比特（也就是对指针所指针的那片比特位换个类型做解释），**而不进行任何类型检查**。

reinterpret_cast主要有三种强制转换用途：**改变指针或引用的类型、将指针或引用转换为一个足够长度的整形、将整型转换为指针或引用类型**。

因此，reinterpret_cast可能导致未定义的行为，应谨慎使用。

1.指针类型之间的转换

在某些情况下，需要在不同指针类型之间进行转换，如将一个int指针转换为char指针。

```
1 int a = 42;
2 int* int_ptr = &a;
3 char* char_ptr = reinterpret_cast<char*>(int_ptr); // 将int指针转换
为char指针
```

IV.dynamic_cast

(1) 其他三种都是编译时完成的，dynamic_cast是运行时处理的，运行时要进行类型检查。

(2) 不能用于内置的基本数据类型的强制转换。

(3) dynamic_cast转换如果成功的话返回的是指向类的指针或引用，转换失败的话则会返回NULL。

(4) 使用dynamic_cast进行转换的，基类中一定要有虚函数，否则编译不通过。需要检测有虚函数的原因：类中存在虚函数，就说明它有想要让基类指针或引用指向派生类对象的情况，此时转换才有意义。这是由于运行时类型检查需要运行时类型信息，而这个信息存储在类的虚函数表。只有定义了虚函数的类才有虚函数表

(5) 在类的转换时，在类层次间进行上行转换时，dynamic_cast和static_cast的效果是一样的。在进行下行转换时，dynamic_cast具有类型检查的功能，比static_cast更安全。

向上转换，即为子类指针指向父类指针（一般不会出问题）；向下转换，即将父类指针转化为子类指针。

向下转换的成功与否还与将要转换的类型有关，即要转换的指针指向的对象的实际类型与转换以后的对象类型一定要相同，否则转换失败。

在C++中，编译期的类型转换有可能会在运行时出现错误，特别是涉及到类对象的指针或引用操作时，更容易产生错误。`dynamic_cast`操作符则可以在运行期对可能产生问题的类型转换进行测试。

编码规范：

类型转换必须显式声明，永远不要依赖隐式类型转换。

9.13 隐式的类类型转换(转换构造函数)

上述类型转换介绍了C++在内置类型之间定义了几种自动转换规则。我们也可以为类定义隐式转换规则。如果构造函数只接受一个实参，则它实际上定义了转换为此类类型的隐式转换机制，有时我们把这种构造函数称作转换构造函数(converting constructor)，此构造函数将实参类型的对象转换成类类型。

能通过一个实参调用的构造函数定义了一条从构造函数的参数类型向类类型隐式转换的规则。

例如：在Sales_data类中，接受string的构造函数和接受istream的构造函数分别定义了从这两种类型向Sales_data隐式转换的规则。也就是说，在需要使用Sales_data的地方，我们可以使用string或者istream代替：

```
1 string null_book="9-999-99999-9";
2 //构造一个临时的sales_data对象
3 //
4 item.combine(null_book);
```

只允许一步类类型转换：编译器只会自动的执行一步类型转换。

关键字`explicit`只对一个实参的构造函数有效。需要多个实参的构造函数不能用于执行隐式转换，所以无需将这些构造函数指定为`explicit`的。只能在类内声明构造函数时使用`explicit`关键字，在类外部定义时不应重复。

```
1 //错误：只能在类内声明构造函数时使用explicit关键字，在类外部定义时不应重
2 //复。
3 explicit sales_data::sales_data(istream& is){
4     read(is,*this);
5 }
```

explicit构造函数只能用于直接初始化：发生隐式转换的另一种情况是当我们执行拷贝形式的初始化时(使用`=`)。此时我们只能使用直接初始化而不能使用`explicit`构造函数：

```
1 sales_data item1(null_book); //正确：直接初始化
2 sales_data item2=null_book; //错误：不能将explicit构造函数用于拷贝形式
    的初始化过程
```

当我们用explicit关键字声明构造函数时，它将只能以直接初始化的形式使用。**而且编译器将不会在自动类型转换过程中使用该构造函数。**

9.14 类型转换运算符

类型转换运算符可以完成类类型的类型转换。类型转换运算符(conversion operator)是类的一种特殊成员函数，负责将一个类类型的值转换成其他类型。类型转换函数的一般形式如下所示：

```
1 operator type() const;
```

type表示某种类型。类型转换运算符可以面向任意类型(除了void以外)进行定义，只要该类型能作为函数的返回类型。因此，不允许转换成数组或者函数类型，但是允许转换成指针(包括数组指针及函数指针)或者引用类型。

类型转换运算符既没有显式的返回类型，也没有形参，而且必须定义成类的成员函数。类型转换运算符通常不应该改变待转换对象的内容，因此类型转换运算符一般声明为const。

```
1 class SmallInt{
2 public:
3     SmallInt(int i=0):val(i)
4     {
5         if(i<0 || i>255)
6             throw std::out_of_range("bad smallint value");
7     }
8     operator int() const{return val;}
9 private:
10    std::size_t val;
11};
```

上述代码将算术类型的值转换成SmallInt对象，而类型转换运算符将SmallInt对象转换成int。

提示：避免过度使用类型转换函数

和使用重载运算符的经验一样，明智地使用类型转换运算符也能极大地简化类设计者的工作，同时使得使用类更加容易。然而，如果在类类型和转换类型之间不存在明显的映射关系，则这样的类型转换可能具有误导性。

例如，假设某个类表示 Date，我们也许会为它添加一个从 Date 到 int 的转换。然而，类型转换函数的返回值应该是什么？一种可能的解释是，函数返回一个十进制数，依次表示年、月、日，例如，July 30, 1989 可能转换为 int 值 19890730。同时还存在另外一种合理的解释，即类型转换运算符返回的 int 表示的是从某个时间节点（比如 January 1, 1970）开始经过的天数。显然这两种理解都合情合理，毕竟从形式上看它们产生的效果都是越靠后的日期对应的整数值越大，而且两种转换都有实际的用处。

问题在于 Date 类型的对象和 int 类型的值之间不存在明确的一对一映射关系。因此在此例中，不定义该类型转换运算符也许会更好。作为替代的手段，类可以定义一个或多个普通的成员函数以从各种不同形式中提取所需的信息。

9.15 Overloading the Function Call Operator ()

赋值（=）、下标（[]）、调用（()）和成员访问箭头（->）运算符必须是成员函数。

对于赋值运算符来说，我们知道一个C++类，程序员如果没有为其定义了赋值操作符重载函数，编译器也会隐式的定义，这样倘若再定义全局赋值运算符重载函数，将会发生二义性。即使编译器允许这样的定义手法，在调用的时候也编译不过：

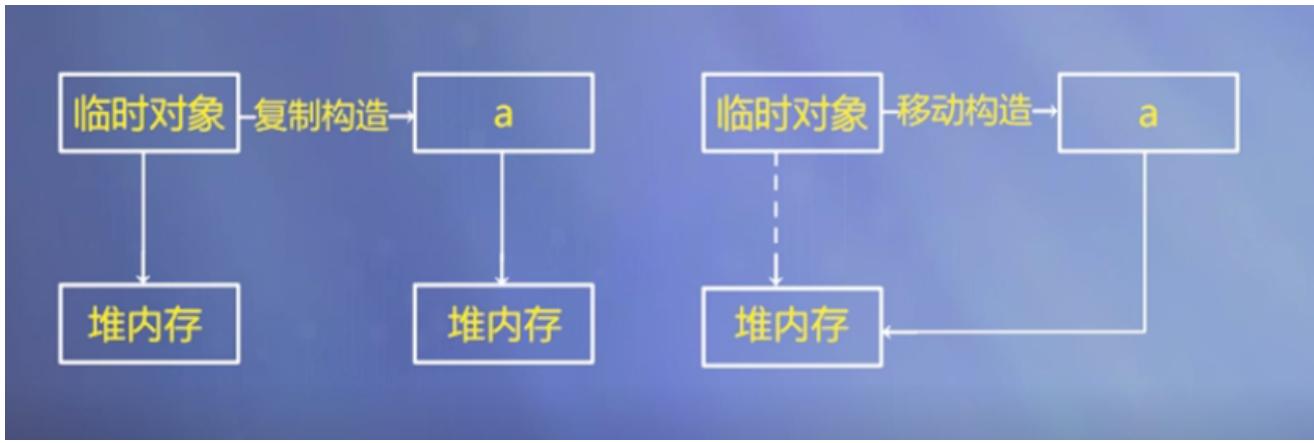
函数调用运算符必须是成员函数

9.16 移动语义

C++11标准中提供了一种新的构造方法---移动构造。在某些情况下，我们没有必要复制对象---只需要移动它们。

C++11引入移动语义：源对象资源的控制权全部交给目标对象

当临时对象在被复制后，就不再被利用了。我们完全可以把临时对象的资源直接移动，这样就避免了多余的复制操作。



&&是右值引用，函数返回的临时变量是右值

10. 继承(Inheritance)

派生类必须使用类派生列表明确指出它是从哪个基类继承而来的。因为每个派生类对象都是属于基类的，并且一个基类可以由多个派生类，基类代表的对象比任意派生类代表的对象多。For example, the base class Vehicle represents all vehicles, including cars, trucks, boats, airplanes, bicycles and so on. By contrast, derived-class Car represents a smaller, more specific subset of all vehicles.

继承关系形成类的层级结构(class hierarchies)，一个基类与派生类之间存在层级关系。虽然类可以独立存在，但一旦它们被使用在继承关系中，它们就成为其他类的附属。一个类要么成为基类- -向其他类提供成员，要么成为派生类- -从其他类继承成员，或者两者兼而有之。

10.1 基类和派生类

基类往往更加具有一般性，派生类往往更加具体。单继承(single inheritance)，一个类是从一个基类派生出来的。多重继承(multiple inheritance)，一个派生类同时从两个或者多个基类继承而来。

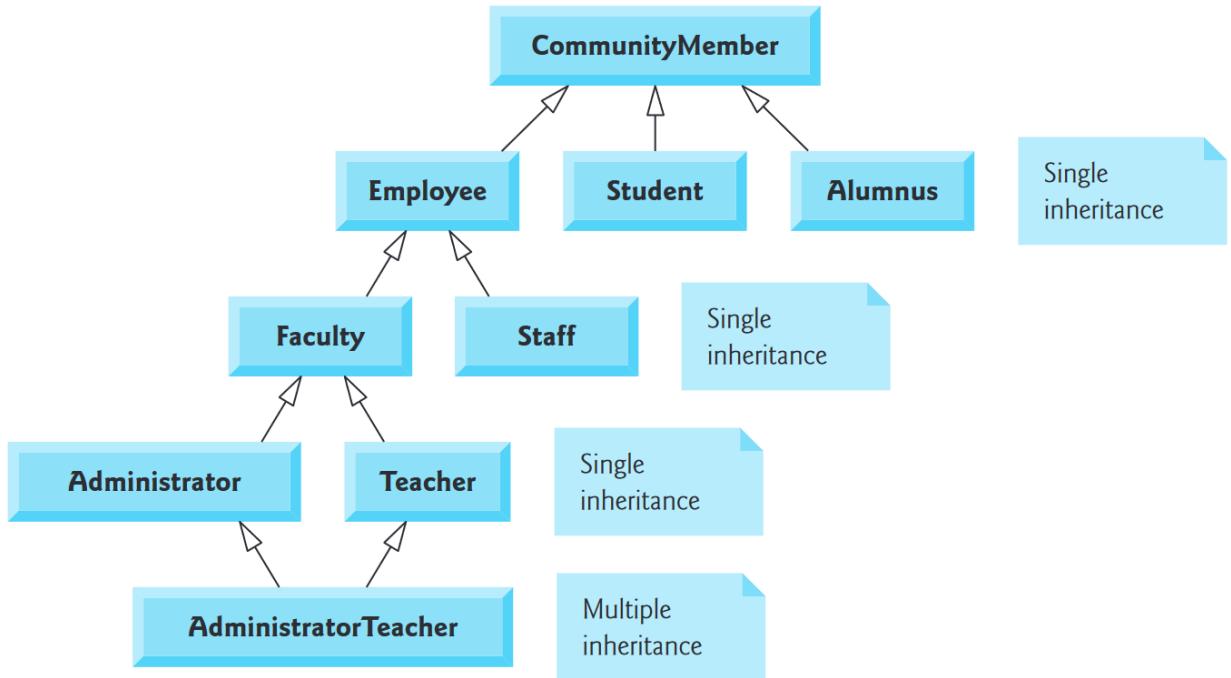


Fig. 11.2 | Inheritance hierarchy for university CommunityMembers.

例如，在上图中，CommunityMember是Employee\Student\Alumnus的直接基类。是图中其他类的间接基类。

10.1.1 继承的基本概念

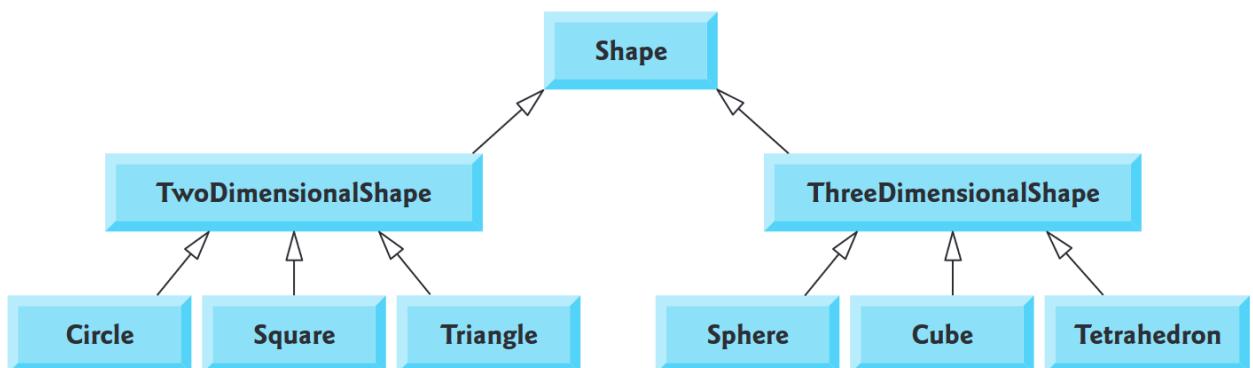


Fig. 11.3 | Inheritance hierarchy for Shapes.

现在重新考虑上图中的继承层次，继承开始于基类Shape，为了指明TwoDimensionalShape是由Shape派生而来，TwoDimensionalShape类的定义可以写为：

```
1 | class TwoDimensionalShape : public Shape
```

这是public继承的示例，也是最常用的一种形式。还有private和protected继承。对于所有形式的继承，基类的private成员不能由基类的派生类直接访问，但是这些基类的private成员仍然是继承的(即，他们仍然是派生类的一部分)。对于public继承，当基类变成派生类的一部分，除private外的成员仍然保持原始的成员访问权限。(例如，基类的

public成员仍然是派生类的Public成员, protected成员仍然是派生类的protected成员)。通过继承基类的成员函数, 派生类可以操控基类的private成员(如果这些继承的成员函数提供了基类的这种功能)。

继承关系并不适宜于所有的类的关系。一些情况下, 复合(composition)关系更加适宜。类似地处理基类对象和派生类对象是可能的; 它们的共性表现在基类成员身上。

10.1.2 基类和派生类的类型转换

公有(public继承)派生类对象可以被当作基类的对象使用, 反之不可以。因为:

1. 派生类的对象可以隐含转换为基类对象
2. 派生类的对象可以初始化基类的引用
3. 派生类的指针可以隐含转换为基类的指针

通过基类对象名、指针只能使用从基类继承的成员。

10.2 引用和指针

复合类型是指基于其他类型定义的类型, 引用和指针属于其中的两种两种类型。与声明变量相比, 定义复合类型的变量要复杂一些。**一条声明语句由一个基本数据类型(base type)和紧随其后的一个声明符列表组成。每个声明符命名了一个变量并指定该变量为与基本数据类型有关的某种类型。**

目前为止, 我们接触到的声明语句中, 声明符就是变量名, 此时变量的类型也就是声明的基本数据类型。还可以有更复杂的声明符, 它基于基本数据类型得到更复杂的类型, 并把它指定给变量。

10.2.1 引用

引用是引用变量的简称, 是C++新增的复合类型。引用是以定义的变量的别名。引用的主要用途是用作函数的形参和返回值。被引用的变量的生命周期一定要比引用长, 这是使用引用的基本原则。

引用就是为类的对象起了另外一个名字, 引用类型引用另外一种类型。通过将声明符写成&d的形式来定义引用类型, d为声明的变量名:

```
1 int ival=1024;
2 int& refval=ival;
3 int& refval;//错误, 引用必须初始化
```

一般在初始化变量时，初始值会被拷贝到新建的对象中。然而在定义引用时，程序把引用和它的初始值绑定在一起，而不是将初始值拷贝给引用。一旦初始化完成，引用将和它的初始值对象一直绑定在一起。因为无法令引用重新绑定到另外一个对象，因此引用必须初始化，初始化后不可改变。

通过pass-by-reference，调用者赋予被调用函数直接访问调用者数据，并修改该数据的能力。

Pass-by-reference会削弱安全性，被调用函数可能会损坏调用者的数据。Pass-by-reference可以消除拷贝大量数据的传值开销。

带&的是引用型参数，它是地址传递，其实参会随着形参的改变而改变；不带&的参数是一般参数，是值传递，其实参不会随着形参的改变而改变。所以，结构改变，并且需要传回这种改变的要用引用型参数，否则用一般参数。

1.可以在一行中同时定义被引用变量和引用变量。

2.引用在定义时要进行初始化。

```
1 int x,&number=x;  
2 or  
3 int x;  
4 int &number=x;
```

3.对引用的操作就是对被引用的变量的操作。

4.引用类型变量的初始值不能是常数。

```
1 int &ref=90;//错误！
```

5.可以使用动态分配的内存空间初始化一个引用变量。

6.C++指针与引用符号应该靠近类型而非名字

```
1 float* p;//不应该是float *p  
2 int& x//不应该是int &x
```

在函数调用中，只需提及变量的名字(例如,number)就可以通过引用来传递它。在被调用函数的函数体内，引用参数实际上指的是调用函数的原始变量，原始变量可以通过被调用函数直接修改。函数原型和头文件必须一致。

在引用类型前面加上const，const引用的目的是，禁止通过修改引用值来改变被引用的对象。

10.2.2 引用的本质

引用的本质是指针常量，传递的是变量的地址。传引用的代码更加简洁，传引用不必使用二级指针。

10.2.3 指针

指针是指向另外一种类型的复合类型。与引用相似，指针也实现了对其他对象的间接访问。指针本身就是一个对象，允许对指针赋值和拷贝，在指针的生命周期内它可以先后指向几个不同的对象。指针无需在定义时赋初值。在块作用域内定义的指针如果没有被初始化，也将拥有一个不确定的值。

指针的值（即地址）应属下列 4 种状态之一：

1. 指向一个对象。
2. 指向紧邻对象所占空间的下一个位置。
3. 空指针，意味着指针没有指向任何对象。
4. 无效指针，也就是上述情况之外的其他值。

因为引用不是对象，没有实际地址，所以不能定义指向引用的指针。

空指针：空指针不指向任何对象，在试图使用一个指针之前代码应该首先检查它是否为空。

```
1 int *p1=nullptr;//等价于 int *p1=0;
2 int *p2=0;//直接将p2初始化为字面量0
3 //需要首先#include <cstdlib>
4 int *p3=NULL;
```

得到空指针最直接的办法就是使用字面值nullptr来初始化指针。nullptr是一种特殊的字面值，它可以被转换为任意其他的指针类型。

10.2.4 引用和 const

如果引用的数据对象类型不匹配，当引用为const时，C++将创建临时变量，让引用指向临时变量。

如果函数的实参不是左值或与const引用形参的类型不匹配，那么C++将创建正确类型的匿名变量，将实参的值传递给匿名变量，并让形参来引用该变量。

1. 使用const可以避免无意中修改数据的编程错误
2. 使用const可以使函数能够处理const和非const实参，否则只能接收非const实参
3. 使用const，函数可以正确生成并使用临时变量

10.2.5 引用用于函数的返回值

传统的函数返回机制与值传递类似。函数的返回值被拷贝到一个临时位置(寄存器或栈)，然后调用者程序再使用这个值。返回引用的语法：

```
1 | 返回的数据类型& 函数名(形参列表){  
2 |  
3 | }
```

1. 如果返回局部变量的引用，此引用本质是野指针。
2. 可以返回函数的引用形参、类的成员、全局变量、静态变量。

10.3 基类和派生类的关系

使用继承，而不是复制粘贴

10.3.1 基类和派生类的构造函数和析构函数

派生类的构造函数必须调用基类的构造函数

析构函数和友元函数，不能从基类继承而来。

当派生类的构造函数调用基类的构造函数时，传向基类构造函数的实参必须与基类的构造函数指定的参数的个数和类型一致，否则会出现编译错误。

在派生类的头文件中，使用#include包含基类。

之前的章节介绍了public和protected访问修饰符。一个基类的公共成员可以在它的体内和任何地方访问，即程序对此类的对象有一个句柄(名字，引用或指针)或者此类的派生类。基类的private成员只能在体内或者friend类访问。

使用protected访问修饰符提供了一种介于public和private之间的访问等级。将基类的原本的private成员改为protected的，则派生类就可以直接访问基类的private成员。基类的protected成员可以在基类的体内访问，也可以被基类的成员和类的friend访问，还有派生类的成员和派生类的friend也可以访问。

使用继承可以使得代码更容易维护。

10.4 派生类中的构造函数和析构函数

构造函数：先调用基类的构造函数，再调用派生类的构造函数；基类先，派生类后

析构函数：先调用派生类的析构函数，再调用基类的析构函数。派生类先，基类后

当程序创建派生类对象时，派生类构造函数会立即调用基类构造函数，基类构造函数的主体将执行，派生类的成员初始值设定项将执行，最后派生类构造函数的主体将执行。如果层次结构包含两个以上的级别，则此过程将向上级联。

当派生类对象被销毁时，程序将调用该对象的析构函数。这开始了析构函数调用的链（或级联），其中派生类析构函数以及直接和间接基类的析构函数以及类成员的执行顺序与构造函数的执行顺序相反。当调用派生类对象的析构函数时，析构函数将执行其任务，然后调用层次结构中下一个基类的析构函数。此过程将重复进行，直到调用层次结构顶部的最后一个基类的析构函数。然后，从内存中删除该对象。

假设我们创建一个派生类的对象，其中基类和派生类都包含（通过组合）其他类的对象。创建该派生类的对象时，首先执行基类成员对象的构造函数，然后执行基类构造函数主体，然后执行派生类成员对象的构造函数，然后执行派生类的构造函数主体。派生类对象的析构函数的调用顺序与其相应构造函数的调用顺序相反。

默认情况下，基类构造函数、析构函数和重载赋值运算符不由派生类继承。但是，派生类构造函数、析构函数和重载赋值运算符可以调用基类版本。

10.4.1 继承基类的构造函数

子类为完成基类初始化，在 C++11 之前，需要在初始化列表调用基类的构造函数，从而完成构造函数的传递。

如果派生类没有构造函数，在这种情况下，C++会为派生类生成默认构造函数，并隐式的调用基类的默认构造函数。

有时，派生类的构造函数只是指定与基类的构造函数相同的参数，并简单地将构造函数参数传递给基类的构造函数。C++11 允许您指定派生类应继承基类的构造函数。为此，需要显式包含 using 声明

- C++11 规定
 - 可用 using 语句继承基类构造函数。
 - 但是只能初始化从基类继承的成员。
 - 派生类新增成员可以通过类内初始值进行初始化。

- 语法形式：

```
1 | using BaseClass::BaseClass; //继承基类的所有构造函数
```

- **如果派生类有自己新增的成员，且需要通过构造函数初始化，则派生类要自定义构造函数。可以在派生类构造函数中调用基类构造函数。**

除了少数例外（下面列出），对于基类中的每个构造函数，编译器都会生成一个派生类构造函数，该构造函数调用相应的基类构造函数。每个生成的构造函数都与派生类同名。

When you inherit constructors:

1. 每个生成的构造函数都具有与其对应的基类构造函数相同的访问说明符（public、protected 或 private）。
2. copy 和 move 构造函数不会被继承。
3. 如果通过在其原型中放置 = delete 来删除基类中的构造函数，则派生类中的相应构造函数也会被删除。
4. 如果派生类未显式定义构造函数，编译器仍会在派生类中生成默认构造函数。
5. 如果在派生类中显式定义的构造函数具有相同的参数列表，则不会继承给定的基类构造函数。
6. 基类构造函数的默认参数不会被继承。相反，编译器在派生类中生成重载构造函数。例如，如果基类声明构造函数

```
1 | BaseClass(int = 0, double = 0.0);
```

编译器生成以下两个不带默认参数的派生类构造函数

```
1 | DerivedClass();
2 | DerivedClass(int);
3 | DerivedClass(int, double);
```

它们都调用指定默认参数的 BaseClass 构造函数。

如果不继承基类的构造函数：

1. 派生类的新增成员：派生类定义构造函数初始化
2. 继承来的成员：调用基类构造函数进行初始化
3. 派生类的构造函数需要向基类的构造函数传递参数

单继承时构造函数的定义语法：

```
1 派生类名::派生类名(基类所需的形参, 本类成员所需的形参): 基类名(参数表), 本类
2 成员初始化列表
3 {
4     //其他初始化操作;
5 }
```

多继承且有对象成员时派生类的构造函数的定义：

```
1 派生类名::派生类名(形参表):
2 基类名1(参数), 基类名2(参数), ..., 基类名n(参数), 本类成员(含对象成员)初始化
3 列表
4 {
5     //其他初始化操作;
6 }
```

构造函数的执行顺序：

1. 调用基类的构造函数，基类构造函数的调用顺序按照它们被继承时声明的顺序(自左向右)
2. 对初始化列表中的成员进行初始化。顺序按照它们在类中定义的顺序；对象成员初始化时自动调用其所属类的构造函数
3. 执行派生类的构造函数体中的内容

10.4.2 继承中的默认构造函数

若基类的构造函数未被显式的调用，基类的默认构造函数就会被调用。

例如：

```
1 circle(){
2     radius=1.0;
3 } //等价于下列的函数形式
4
5 circle():Shape{} {
6     radius=1.0;
7 } //Shape为circle的基类
```

所以，要考虑给基类提供默认构造函数。

10.4.3 派生类的复制构造函数

从基类继承而来的成员的复制构造由基类的复制构造函数完成，派生类新增的成员的复制构造由派生类的复制构造函数完成。

若派生类没有声明复制构造函数：编译器会在需要时生成一个隐含的复制构造函数，先调用基类的复制构造函数，再为派生类新增的成员执行复制。

若派生类定义复制构造函数：1.一般要为基类的复制构造函数传递参数 2.复制构造函数只能接收一个参数，既用来初始化派生类定义的成员，也将被传递给基类的复制构造函数 3.基类的复制构造函数形参类型是基类对象的引用，实参可以是派生类对象的引用。

10.4.4 派生类的析构函数

析构函数不被继承，派生类如果需要，要自行声明析构函数

声明方法与无继承关系时类的析构函数相同

不需要显式的调用基类的析构函数，系统会自动的隐式调用

先执行派生类析构函数的函数体，再调用基类的析构函数

10.4.5 访问从基类继承的成员(继承中的名字隐藏)

当派生类与基类中有相同成员时：

1.若未特别限定，则通过派生类对象使用的是派生类中的同名成员

2.如要通过派生类对象访问基类中被隐藏的同名成员，应使用基类名和作用域操作符(::)来限定。

内部作用域的名字隐藏外部作用域的(同名)名字

而派生类视为内部作用域，基类视为外部作用域。

可以避免某些潜在的危险行为

10.5 重定义函数

1. Redefining Functions (重定义函数)

Shape::toString() 被Circle继承，因此可以通过Circle对象调用该函数。

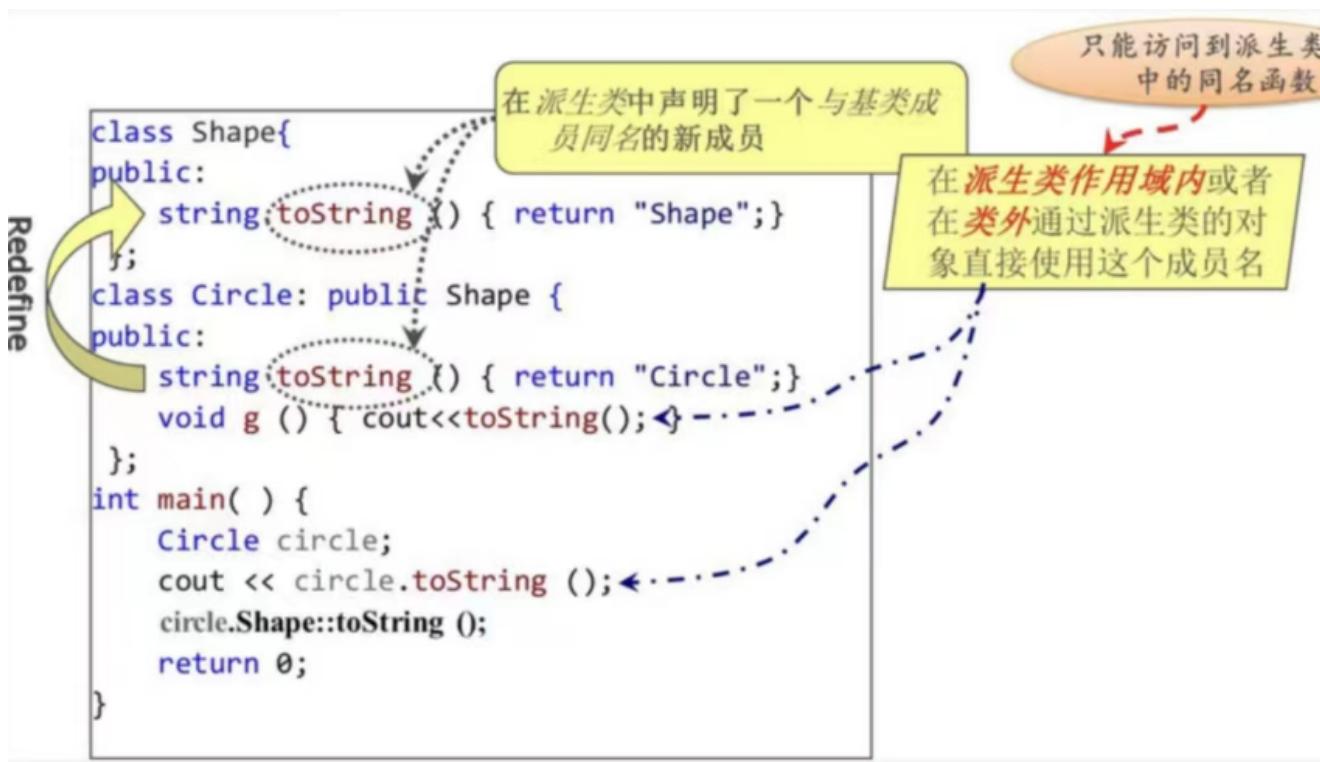
```
1 | std::cout << circle.toString();
```

但基类的toString()无法输出派生类对象信息

可以重定义派生类的toString()以描述派生类对象

```
1 #include <string>
2
3 std::string Shape::toString() {
4     using namespace std::string_literals;
5     return "Shape color "s + color
6         + ((filled) ? " filled"s : " not filled"s);
7 }
8
9 string Circle::toString() {
10    return "Circle color " + color +
11        " filled " + ((filled) ? "true" : "false");
12 }
13 string Rectangle::toString() {
14    return "This is a rectangle object";
15 }
```

2. Redefine (重定义)



3. Redefine v.s. Overload (重定义与重载)

3.1. Overload Functions (重载函数)

3.1.1. more than one function with the same name (多个函数名字相同)

3.1.2. But different in at least one of the signatures: (但至少一个特征不同)

- (1) parameter type (参数类型)
- (2) parameter number (参数数量)
- (3) parameter sequence (参数顺序)

3.2. **Redefine Functions (重定义函数)**

3.2.1. The functions have the same signature (函数特征相同)

- (1) Name (同名)
- (2) Parameters (including type, number and sequence) (同参数: 类型, 数量和顺序)
- (3) Return type (返回值类型)

3.2.2. Defined in base class and derived class, respectively (在基类和派生类中分别定义)

10.6 public, protected and private Inheritance(访问控制属性)

我们一般使用public继承, 使用protected继承比较少见。

Base-class member-access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
public	<p>public in derived class.</p> <p>Can be accessed directly by member functions, friend functions and nonmember functions.</p>	<p>protected in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>	<p>private in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>
protected	<p>protected in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>	<p>protected in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>	<p>private in derived class.</p> <p>Can be accessed directly by member functions and friend functions.</p>
private	<p>Hidden in derived class.</p> <p>Can be accessed by member functions and friend functions through public or protected member functions of the base class.</p>	<p>Hidden in derived class.</p> <p>Can be accessed by member functions and friend functions through public or protected member functions of the base class.</p>	<p>Hidden in derived class.</p> <p>Can be accessed by member functions and friend functions through public or protected member functions of the base class.</p>

当使用**public**继续生成派生类，基类中的**public**成员变成了派生类的**public**成员，基类的**protected**成员变成了派生类的**protected**成员。基类的**Private**成员不可以由派生类直接访问，但是可以通过基类的**public**和**protected**成员函数访问。

当使用基类派生派生类，使用**public\protected\private**都有可能。

public继承：

基类的**public**和**protected**成员，访问属性在派生类中保持不变；基类的**private**成员，派生类的成员函数不可以直接访问。派生类中的成员函数可以直接访问基类的**public**和**protected**成员，但是不能直接访问基类的**private**成员。通过派生类的对象，只能访问**public**成员。

private继承：是最严格的一种继承

基类的**public**和**protected**成员都以**private**的身份出现在派生类中；基类的**private**成员，派生类的成员函数不可以直接访问。派生类中的成员函数可以直接访问基类中的**public**和**protected**成员，但是不能直接访问基类的**private**成员。通过派生类的对象，不能访问从基类继承的任何成员。

protected继承：

基类的public和protected成员都以protected的身份出现在派生类中；基类的private成员，不可以直接访问。派生类中的成员函数，可以直接访问基类中的public和protected成员，但是不能直接访问基类的private成员。通过派生类的对象，不能访问从基类继承的任何成员。

10.7 虚基类

当派生类从多个基类继承，而这些基类又有共同基类，则在访问此共同基类的成员时，将产生冗余，并有可能因冗余带来不一致性。

虚基类的声明：以virtual说明基类继承方式。例如：

```
1 | class B1:virtual public B
```

虚基类主要用来解决多继承时可能发生的对同一基类继承多次而产生的二义性问题；为最远的派生类提供唯一的基类成员，而不重复产生多次复制。

在第一级继承时就要将共同基类设计为虚基类

虚基类及其派生类构造函数

- ◆ 建立对象时所指定的类称为**最远派生类**。
- ◆ 虚基类的成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的
- ◆ 在整个继承结构中，直接或间接继承虚基类的所有派生类，都必须在构造函数的成员初始化表中为虚基类的构造函数列出参数。如果未列出，则表示调用该虚基类的默认构造函数。
- ◆ 在建立对象时，只有最远派生类的构造函数调用虚基类的构造函数，其他类对虚基类构造函数的调用被忽略。

11. 多态(Polymorphism)

11.1 引言

使用多态性，您可以设计和实现易于扩展的系统，只要新类是程序通常处理的继承层次结构的一部分，就可以添加新类，而无需对程序的常规部分进行修改。程序中唯一必须更改以适应新类的部分是那些需要直接了解添加到层次结构中的新类的部分。例如，如

果我们创建继承自 Animal 类的 Tortoise 类（它可能通过爬行一英寸来响应移动消息），我们只需要编写 Tortoise 类和实例化 Tortoise 对象的模拟部分。处理每只动物的模拟部分通常可以保持不变。

11.2 多态初窥

假定我们需要设计一个游戏，此游戏需要操作多个不同类型的对象，包括类 Martian, Venutian, Plutonian, SpaceShip 和 LaserBeam。设想上述的几个类都是从基类 SpaceObject 继承而来，此基类包括成员函数 draw。每个派生类都以适合该类的方式实现此函数。一个屏幕管理程序包含一个容器（例如，vector），该容器包含指向各种类对象的 SpaceObject 指针。为了刷新屏幕，屏幕管理程序会周期性的向每个对象发送相同的信息——即 draw 函数。每个类类型的对象都会以独特的方式回应此函数。例如， Martian 类的对象会用适当数量的触角将自己画成红色， SpaceShip 对象可能会将自己画成一个银色的飞碟， LaserBeam 对象可能会在屏幕上将自己绘制为明亮的红色光束。一条发送到不同类的对象的消息（此处即为 draw 函数）会导致不同的结果。

多态屏幕管理器有助于向系统添加新类，只需对其代码进行最少的修改。

多态性使您能够处理一般性问题，并让执行时环境关注细节。您可以指示各种对象以适合这些对象的方式运行，甚至不知道它们的类型，只要这些对象属于同一继承层次结构，并且正在通过公共基类指针或公共基类引用进行访问。

多态性促进了可扩展性：为调用多态行为而编写的软件是独立于消息发送到的对象的特定类型编写的。因此，可以在不修改基本系统的情况下将可以响应现有消息的新型对象合并到这样的系统中。只有实例化新对象的客户端代码才能适应新类型。

广义的多态概念：不同类型的实体/对象对于同一消息有不同的响应，这就是多态性。

11.2.1 联编

即确定具有多态性的语句调用哪个函数的过程

1. 静态联编

在程序编译时确定调用哪个函数，例如函数重载

2. 动态联编

在程序运行时，才能够确定调用哪个函数，例如用动态联编实现的多态，也称为运行时多态

11.3 继承关系中对象的关系

通过public继承，派生类的对象可以视为是基类的对象。

11.3.1 Invoking Base-Class Functions from Derived-Class Objects

```
1 //  
2 // Created by 22364 on 2023/11/6.  
3 //  
4 #include <iostream>  
5 #include <iomanip>  
6 #include "CommissionEmployee.h"  
7 #include "BasePlusCommissionEmployee.h"  
8 using namespace std;  
9  
10 int main(){  
11     // create base-class object  
12     CommissionEmployee commissionEmployee{  
13         "Sue", "Jones", "222-22-2222", 10000, .06};  
14     // create derived-class object  
15     BasePlusCommissionEmployee basePlusCommissionEmployee{  
16         "Bob", "Lewis", "333-33-3333", 5000, .04, 300};  
17  
18     cout << fixed << setprecision(2); // set floating-point  
formatting  
19  
20     // output objects commissionEmployee and  
basePlusCommissionEmployee  
21     cout << "DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS:\n"  
22         <<commissionEmployee.toString() // base-class toString  
23         <<"\n\n"  
24         <<basePlusCommissionEmployee.toString(); // derived-class  
toString  
25  
26     // natural: aim base-class pointer at base-class object  
27     CommissionEmployee*  
commissionEmployeePtr{&commissionEmployee};  
28     cout << "\n\nCALLING TOSTRING WITH BASE-CLASS POINTER TO "  
29         << "\nBASE-CLASS OBJECT INVOKES BASE-CLASS TOSTRING  
FUNCTION:\n"  
30         <<commissionEmployeePtr->toString();// base version
```

```
31
32 // natural: aim derived-class pointer at derived-class object
33 BasePlusCommissionEmployee* basePlusCommissionEmployeePtr{
34     &basePlusCommissionEmployee}// natural
35 cout << "\n\nCALLING TOSTRING WITH DERIVED-CLASS POINTER TO "
36     << "\nDERIVED-CLASS OBJECT INVOKES DERIVED-CLASS "
37     << "TOSTRING FUNCTION:\n"
38     <<basePlusCommissionEmployeePtr->toString()// derived
39 version
40
41 // aim base-class pointer at derived-class object
42 commissionEmployeePtr = &basePlusCommissionEmployee;
43 cout << "\n\nCALLING TOSTRING WITH BASE-CLASS POINTER TO "
44     << "DERIVED-CLASS OBJECT\nINVOKES BASE-CLASS TOSTRING "
45     << "FUNCTION ON THAT DERIVED-CLASS OBJECT:\n"
46     <<commissionEmployeePtr->toString() //base version
47     <<endl;
```

DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS:

```
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06
```

```
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

CALLING TOSTRING WITH BASE-CLASS POINTER TO
BASE-CLASS OBJECT INVOKES BASE-CLASS TOSTRING FUNCTION:

```
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06
```

CALLING TOSTRING WITH DERIVED-CLASS POINTER TO
DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS TOSTRING FUNCTION:

```
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

CALLING TOSTRING WITH BASE-CLASS POINTER TO DERIVED-CLASS OBJECT
INVOKES BASE-CLASS TOSTRING FUNCTION ON THAT DERIVED-CLASS OBJECT:

```
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

在上述代码最后的一段，将派生类对象basePlusCommissionEmployee的地址赋给基类的指针commissionEmployeePtr。虽然基类的指针此时指向派生类对象，但是在使用基类指针调用成员函数toString时，调用的是基类的成员函数toString。该程序中每个**toString成员函数调用的输出显示，被调用的函数取决于指针所指的实际对象的类型，而不是指针的类型。**

11.3.2 Aiming Derived-Class Pointers at Base-Class Objects

可以将派生类对象的地址赋给基类类型的指针，但是反过来不可以。

11.3.3 Derived-Class Member-Function Calls via Base-Class Pointers

使用基类指针，编译器允许我们只调用基类里的成员函数。因此，如果基类的指针指向一个派生类对象，试图访问派生类特有的成员函数，就会出现编译错误。例如下列代码所示：

```
1 //  
2 // Created by 22364 on 2023/11/6.  
3 //  
4 #include <string>  
5 #include "CommissionEmployee.h"  
6 #include "BasePlusCommissionEmployee.h"  
7 using namespace std;  
8  
9 int main(){  
10     // create derived-class object  
11     BasePlusCommissionEmployee basePlusCommissionEmployee{  
12         "Bob", "Lewis", "333-33-3333", 5000, .04, 300};  
13  
14     // aim base-class pointer at derived-class object (allowed)  
15     CommissionEmployee*  
16     commissionEmployeePtr{&basePlusCommissionEmployee};  
17  
18     // invoke base-class member functions on derived-class  
19     // object through base-class pointer (allowed)  
20     string firstName{commissionEmployeePtr->getFirstName()};  
21     string lastName{commissionEmployeePtr->getLastName()};  
22     string ssn{commissionEmployeePtr->getSocialSecurityNumber()};  
23     double grossSales{commissionEmployeePtr->getGrossSales()};  
24     double commissionRate{commissionEmployeePtr-  
25         >getCommissionRate()};  
26  
27     // attempt to invoke derived-class-only member functions  
28     // on derived-class object through base-class pointer  
29     (disallowed)  
30     double baseSalary{commissionEmployeePtr->getBaseSalary()};  
31     commissionEmployeePtr->setBaseSalary(500);  
32 }  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
999  
1000  
1001  
1002  
1003  
1004  
1005  
1006  
1007  
1008  
1009  
1009  
1010  
1011  
1012  
1013  
1014  
1015  
1016  
1017  
1018  
1019  
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026  
1027  
1028  
1029  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079  
1079  
1080  
1081  
1082  
1083  
1084  
1085  
1086  
1087  
1088  
1089  
1089  
1090  
1091  
1092  
1093  
1094  
1095  
1096  
1097  
1098  
1099  
1099  
1100  
1101  
1102  
1103  
1104  
1105  
1106  
1107  
1108  
1109  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188  
1189  
1189  
1190  
1191  
1192  
1193  
1194  
1195  
1196  
1197  
1198  
1199  
1199  
1200  
1201  
1202  
1203  
1204  
1205  
1206  
1207  
1208  
1209  
1209  
1210  
1211  
1212  
1213  
1214  
1215  
1216  
1217  
1218  
1219  
1219  
1220  
1221  
1222  
1223  
1224  
1225  
1226  
1227  
1228  
1229  
1229  
1230  
1231  
1232  
1233  
1234  
1235  
1236  
1237  
1238  
1239  
1239  
1240  
1241  
1242  
1243  
1244  
1245  
1246  
1247  
1248  
1249  
1249  
1250  
1251  
1252  
1253  
1254  
1255  
1256  
1257  
1258  
1259  
1259  
1260  
1261  
1262  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1299  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308  
1309  
1309  
1310  
1311  
1312  
1313  
1314  
1315  
1316  
1317  
1318  
1319  
1319  
1320  
1321  
1322  
1323  
1324  
1325  
1326  
1327  
1328  
1329  
1329  
1330  
1331  
1332  
1333  
1334  
1335  
1336  
1337  
1338  
1339  
1339  
1340  
1341  
1342  
1343  
1344  
1345  
1346  
1347  
1348  
1349  
1349  
1350  
1351  
1352  
1353  
1354  
1355  
1356  
1357  
1358  
1359  
1359  
1360  
1361  
1362  
1363  
1364  
1365  
1366  
1367  
1368  
1369  
1369  
1370  
1371  
1372  
1373  
1374  
1375  
1376  
1377  
1378  
1379  
1379  
1380  
1381  
1382  
1383  
1384  
1385  
1386  
1387  
1388  
1389  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428  
1429  
1429  
1430  
1431  
1432  
1433  
1434  
1435  
1436  
1437  
1438  
1439  
1439  
1440  
1441  
1442  
1443  
1444  
1445  
1446  
1447  
1448  
1449  
1449  
1450  
1451  
1452  
1453  
1454  
1455  
1456  
1457  
1458  
1459  
1459  
1460  
1461  
1462  
1463  
1464  
1465  
1466  
1467  
1468  
1469  
1469  
1470  
1471  
1472  
1473  
1474  
1475  
1476  
1477  
1478  
1479  
1479  
1480  
1481  
1482  
1483  
1484  
1485  
1486  
1487  
1488  
1489  
1489  
1490  
1491  
1492  
1493  
1494  
1495  
1496  
1497  
1498  
1499  
1499  
1500  
1501  
1502  
1503  
1504  
1505  
1506  
1507  
1508  
1509  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548  
1549  
1549  
1550  
1551  
1552  
1553  
1554  
1555  
1556  
1557  
1558  
1559  
1559  
1560  
1561  
1562  
1563  
1564  
1565  
1566  
1567  
1568  
1569  
1569  
1570  
1571  
1572  
1573  
1574  
1575  
1576  
1577  
1578  
1579  
1579  
1580  
1581  
1582  
1583  
1584  
1585  
1586  
1587  
1588  
1589  
1589  
1590  
1591  
1592  
1593  
1594  
1595  
1596  
1597  
1598  
1599  
1599  
1600  
1601  
1602  
1603  
1604  
1605  
1606  
1607  
1608  
1609  
1609  
1610  
1611  
1612  
1613  
1614  
1615  
1616  
1617  
1618  
1619  
1619  
1620  
1621  
1622  
1623  
1624  
1625  
1626  
1627  
1628  
1629  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1669  
1670  
1671  
1672  
1673  
1674  
1675  
1676  
1677  
1678  
1679  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1699  
1700  
1701  
1702  
1703  
1704  
1705  
1706  
1707  
1708  
1709  
1709  
1710  
1711  
1712  
1713  
1714  
1715  
1716  
1717  
1718  
1719  
1719  
1720  
1721  
1722  
1723  
1724  
1725  
1726  
1727  
1728  
1729  
1729  
1730  
1731  
1732  
1733  
1734  
1735  
1736  
1737  
1738  
1739  
1739  
1740  
1741  
1742  
1743  
1744  
1745  
1746  
1747  
1748  
1749  
1749  
1750  
1751  
1752  
1753  
1754  
1755  
1756  
1757  
1758  
1759  
1759  
1760  
1761  
1762  
1763  
1764  
1765  
1766  
1767  
1768  
1769  
1769  
1770  
1771  
1772  
1773  
1774  
1775  
1776  
1777  
1778  
1779  
1779  
1780  
1781  
1782  
1783  
1784  
1785  
1786  
1787  
1788  
1789  
1789  
1790  
1791  
1792  
1793  
1794  
1795  
1796  
1797  
1798  
1799  
1799  
1800  
1801  
1802  
1803  
1804  
1805  
1806  
1807  
1808  
1809  
1809  
1810  
1811  
1812  
1813  
1814  
1815  
1816  
1817  
1818  
1819  
1819  
1820  
1821  
1822  
1823  
1824  
1825  
1826  
1827  
1828  
1829  
1829  
1830  
1831  
1832  
1833  
1834  
1835  
1836  
1837  
1838  
1839  
1839  
1840  
1841  
1842  
1843  
1844  
1845  
1846  
1847  
1848  
1849  
1849  
1850  
1851  
1852  
1853  
1854  
1855  
1856  
1857  
1858  
1859  
1859  
1860  
1861  
1862  
1863  
1864  
1865  
1866  
1867  
1868  
1869  
1869  
1870  
1871  
1872  
1873  
1874  
1875  
1876  
1877  
1878  
1879  
1879  
1880  
1881  
1882  
1883  
1884  
1885  
1886  
1887  
1888  
1889  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907  
1908  
1909  
1909  
1910  
1911  
1912  
1913  
1914  
1915  
1916  
1917  
1918  
1919  
1919  
1920  
1921  
1922  
1923  
1924  
1925  
1926  
1927  
1928  
1929  
1929  
1930  
1931  
1932  
1933  
1934  
1935  
1936  
1937  
1938  
1939  
1939  
1940  
1941  
1942  
1943  
1944  
1945  
1946  
1947  
1948  
1949  
1949  
1950  
1951  
1952  
1953  
1954  
1955  
1956  
1957  
1958  
1959  
1959  
1960  
1961  
1962  
1963  
1964  
1965  
1966  
1967  
1968  
1969  
1969  
1970  
1971  
1972  
1973  
1974  
1975  
1976  
1977  
1978  
1979  
1979  
1980  
1981  
1982  
1983  
1984  
1985  
1986  
1987  
1988  
1989  
1989  
1990  
1991  
1992  
1993  
1994  
1995  
1996  
1997  
1998  
1999  
1999  
2000  
2001  
2002  
2003  
2004  
2005  
2006  
2007  
2008  
2009  
2010  
2011  
2012  
2013  
2014  
2015  
2016  
2017  
2018  
2019  
2020  
2021  
2022  
2023  
2024  
2025  
2026  
2027  
2028  
2029  
2030  
2031  
2032  
2033  
2034  
2035  
2036  
2037  
2038  
2039  
2039  
2040  
2041  
2042  
2043  
2044  
2045  
2046  
2047  
2048  
2049  
2049  
2050  
2051  
2052  
2053  
2
```

```
===== [ 构建 | test_11_04 | Debug ] =====
\"CLion 2023.1.4\bin\cmake\win\x64\bin\cmake.exe" --build D:\clionproject\test_11_04\cmake-build-debug --
lding CXX object CMakeFiles/test_11_04.dir/main2.cpp.obj
CMakeFiles/test_11_04.dir/main2.cpp.obj
\"CLion 2023.1.4\bin\mingw\bin\g++.exe" -g -std=gnu++14 -fdiagnostics-color=always -MD -MT CMakeFiles/
project/test_11_04/main2.cpp: In function 'int main()':
project/test_11_04/main2.cpp:27:46: error: 'class CommissionEmployee' has no member named 'getBaseSalary'
    double baseSalary{commissionEmployeePtr->getBaseSalary()};
                                         ^~~~~~
project/test_11_04/main2.cpp:28:28: error: 'class CommissionEmployee' has no member named 'setBaseSalary'
    commissionEmployeePtr->setBaseSalary(500);
                                         ^~~~~~
Build stopped: subcommand failed.
```

编译器会允许指向派生类对象的基类类型指针访问派生类特有的函数--前提是显式的将基类类型的指针cast到派生类类型的指针。这种操作叫做向下转型(downcasting)。向下转型允许只能由派生类对象完成的派生类特定操作由基类类型的指针完成。但是向下转型会存在潜在的危险。后续章节会介绍如何消除这种潜在的危险。

11.4 (虚函数及虚析构函数)Virtual Functions and Virtual Destructors

11.4.1 为何虚函数有用?

假定Circle\Triangle\Rectangle\square都是基类Shape的派生类。这些类中的每个类都可能被赋予通过成员函数绘制自身的能力，但每个形状的功能却截然不同。在一个绘制形状集合的程序中，将所有的形状笼统地视为基类Shape的对象是有用的。然后，对于任意形状的绘制，我们可以简单地使用基类Shape指针来调用绘图函数，让程序根据基类Shape指针在任意时刻所指向的对象的类型，动态地确定使用哪个派生类绘图函数的(也就是说，在运行时)。这是多态行为。

通过使用虚函数，对象的类型(而不是用来调用对象的成员函数的句柄的类型)，决定了调用哪个版本的虚函数。

11.4.2 声明虚函数

为了实现上述的功能，我们在基类中将draw函数声明为虚函数，并且在每个派生类中重写draw函数以绘制相应的形状。从实现的角度看，重写一个函数无异于重新定义一个(也是一直沿用的做法)。派生类中的重写函数与基类中的重写函数具有相同的函数原型。如果我们没有将基类函数声明为虚函数，我们可以重新定义那个函数。如果我们将基类函数声明为虚函数，我们可以重写它以实现多态行为。虚函数的声明格式如下：

```
1 | virtual 函数类型 函数名();
```

不需要形参列表。

虚函数的实现要在类外实现，不能在类内实现。

一旦一个函数被声明为虚拟，从那个点开始，它在继承层次结构中一直保持虚拟，即使当派生类重写该函数时，该函数没有被明确声明为虚拟。

尽管某些函数由于在类层次结构中的一个较高的声明而隐含地是虚拟的，但为了清晰起见，在类层次结构的每个层次上都显式地声明这些函数是虚拟的。

当派生类选择不重写其基类的虚拟函数时，派生类只需继承其基类的虚拟函数实现。

初识虚函数

- ◆ 用virtual关键字说明的函数
- ◆ 虚函数是实现运行时多态性基础
- ◆ C++中的虚函数是动态绑定的函数
- ◆ 虚函数必须是非静态的成员函数，虚函数经过派生之后，就可以实现运行过程中的多态。

11.4.3 Invoking a virtual Function Through a Base-Class Pointer or Reference

如果一个程序通过基类类型的指向派生类对象的指针调用虚函数(即，`shapePtr->draw()`)或者一个指向派生类对象的基类引用(即，`shapeRef.draw()`),程序会通过对对象的类型动态的选择正确的派生类函数执行，而不是根据指针或者引用的类型。**在执行时刻**(而不是在编译时)选择合适的函数调用称为**动态绑定**。

◆ 虚表

- 每个多态类有一个虚表 (virtual table)
- 虚表中有当前类的各个虚函数的入口地址
- 每个对象有一个指向当前类的虚表的指针 (虚指针vptr)

◆ 动态绑定的实现

- 构造函数中为对象的虚指针赋值
- 通过多态类型的指针或引用调用成员函数时，通过虚指针找到虚表，进而找到所调用的虚函数的入口地址
- 通过该入口地址调用虚函数

11.4.4 通过对象的名字调用虚函数

当通过名称引用特定对象并使用点成员选择算子(例如，`squareObject.draw()`)调用一个虚函数时，函数调用在编译时(这被称为静态绑定)得到解决，所调用的虚函数是为该特定对象的类(或被继承)定义的虚函数--这不是多态行为。与虚函数的动态绑定只发生在指针和引用。

11.4.5 虚函数的注意事项

为了防止错误，将C++11的override关键字应用到每个派生类函数重写基类虚函数的原型中。这使得编译器能够检查基类是否具有具有相同签名的虚拟成员函数。如果不是，则编译器产生错误。这不仅可以确保你用合适的签名覆盖基类函数，还可以防止你意外地隐藏一个具有相同名字和不同签名的基类函数。

11.4.6 虚析构函数

使用多态性处理类层次结构的动态分配对象时，可能会出现问题。到目前为止，您已经看到了未使用关键字 virtual 声明的析构函数。如果通过将 delete 运算符应用于指向该对象的基类指针来销毁具有非虚拟析构函数的派生类对象，则 C++ 标准指定该行为未定义。**构造函数不能是虚函数。**

当我们delete一个动态分配的对象的指针时将执行析构函数。如果该指针指向继承体系中的某个类型，则有可能出现指针的静态类型与被删除对象的动态类型不符的情况。例如，如果delete一个Quote类型的指针，而指针指向了一个Bulk_quote类型的对象。如果这样的话编译器必须清楚它应该执行的是Bulk_quote的析构函数。

此问题的简单解决方案是在基类中创建一个公共虚拟析构函数。如果基类析构函数声明为虚拟析构函数，则任何派生类的析构函数也是虚拟的。例如，在类 CommissionEmployee 的定义中，我们可以按如下方式定义虚拟析构函数：

```
1 | virtual ~CommissionEmployee() {};
```

现在，如果通过将 delete 运算符应用于基类指针来显式销毁层次结构中的对象，则会根据基类指针指向的对象调用相应类的析构函数。请记住，当派生类对象被销毁时，派生类对象的基类部分也会被销毁，因此派生类和基类的析构函数都必须执行。基类析构函数在派生类析构函数之后自动执行。从现在开始，我们将在每个包含虚拟函数并需要析构函数的类中包含一个虚拟析构函数。

如果类具有虚拟函数，则始终提供虚拟析构函数，即使该类不需要虚拟析构函数。这可确保在通过基类指针删除派生类对象时，将调用自定义派生类析构函数（如果有）。

构造函数不能是虚拟的。将构造函数声明为 virtual 是编译错误。

前面的析构函数定义也可以写成如下：

```
1 | virtual ~CommissionEmployee() = default;
```

在 C++11 中，可以告诉编译器显式生成默认构造函数、复制构造函数、移动构造函数、复制赋值运算符、移动赋值运算符或析构函数的默认版本，方法是遵循特殊成员函数的原型 = default。例如，当您显式定义类的构造函数，并且仍希望编译器也生成默认构造函数时，这很有用，在这种情况下，请将以下声明添加到类定义中：

```
1 | className() = default;
```

11.4.7 override\final关键字

override

- ◆ 多态行为的基础：基类声明虚函数，派生类声明一个函数覆盖该虚函数；
- ◆ 覆盖要求：函数签名（signature）完全一致。
- ◆ 函数签名包括：函数名 参数列表 const

- ◆ C++11 引入显式函数覆盖，在编译期而非运行期捕获此类错误。
- ◆ 在虚函数显式重载中运用，编译器会检查基类是否存在一个虚拟函数，与派生类中带有声明override的虚拟函数，有相同的函数签名（signature）；若不存在，则会回报错误。

override 的价值在于：避免程序员在覆写时错命名或无虚函数导致隐藏bug。

在 C++11 之前，派生类可以重写其任何基类的虚函数。在 C++11 中，在其原型中声明为 final 的基类虚函数，如：

```
1 | virtual 函数名(参数列表) final;
```

不能在任何派生类中重写，这保证了基类的final成员函数定义将由所有基类对象以及基类的直接和间接派生类的所有对象使用。同样，在 C++11 之前，任何现有类都可以用作层次结构中的基类。从 C++11 开始，您可以将一个类声明为 final，以防止它被用作基类，如

```
1 class MyClass final { // this class cannot be a base class
2     // class body
3 }
```

尝试重写final成员函数或从final基类继承会导致编译错误。

11.5 Type Fields and switch Statements

决定对象类型的一种方式是使用switch语句检查对象中字段的值。这使我们能够区分对象类型，然后为特定对象调用适当的操作，类似于多态性。例如，在形状层次结构中，如果每个形状类(例如，Circle\Triangle\Rectangle\square)的对象都具有shapeType属性，switch语句就可以检查对象的shapeType决定调用哪个toString函数。

使用switch逻辑会使得程序面临各种潜在的问题。例如，您可能忘记在必要时包含类型测试，或者可能忘记在switch语句中测试所有可能的情况。通过添加新类型来修改基于switch的系统时，可能会忘记在所有相关的switch语句中插入新大小写。类的每次添加或删除都需要修改系统中每个关联的switch语句;跟踪这些更改可能非常耗时且容易出错。

多态编程可以消除对switch逻辑的需求。通过使用多态机制来执行等效逻辑，可以避免通常与switch逻辑相关的各种错误。

使用多态性的一个有趣的结果是程序呈现出简化的外观。它们包含较少的分支逻辑和更简单的顺序代码。

11.6 抽象类和纯虚函数

当我们把类看作一种类型时，我们假定程序会创造属于此种类型的对象。然而在某些情况下，定义从未打算从中实例化任何对象的类很有用。即，只定义类，并不为类实例化对象。这种类叫做抽象类(abstract classes)。因为这种类在继承层次结构中经常用作基类，也叫做抽象基类。可以被用来实例化对象的类叫做具体类(concrete classes)。

11.6.1 Pure virtual Functions(纯虚函数)

通过声明类的一个或多个虚拟函数是“纯”的，类是抽象的。纯虚函数是通过在其声明中放置“=0”来指定的，如：

```
1 virtual 函数类型 函数名(参数列表) = 0; // pure virtual function
```

“=0”是一个纯说明符。纯虚函数不提供实现。每个具体的派生类都必须用这些函数的具体实现重写所有基类纯虚函数;否则，派生类也是抽象的。

纯虚函数

◆ 纯虚函数是一个在基类中声明的虚函数，它在该基类中没有定义具体的操作内容，要求各派生类根据实际需要定义自己的版本，纯虚函数的声明格式为：

`virtual 函数类型 函数名(参数表) = 0;`

虚函数和纯虚函数的区别是：虚函数有实现，为派生类提供重写函数的选项；纯虚函数没有实现，要求派生类重写该派生类的函数，以便该派生类具体化；否则，派生类将保持抽象。回到我们前面的空间对象示例，基类 `SpaceObject` 具有函数 `draw` 的实现是没有意义的（因为如果没有关于正在绘制的空间对象类型的特定信息，就无法绘制通用空间对象）。定义为虚拟函数（而不是纯虚拟函数）的函数示例是返回对象名称的函数。我们可以命名一个通用的 `Space Object`（例如 "space object"），因此可以为该函数提供一个默认的实现，并且该函数不需要是纯虚的。但是，该函数仍然被声明为虚拟的，因为预期派生类将重写该函数，为派生类对象提供更具体的名称。

一个抽象类在一个类层次结构中为由它派生出来的各种类定义了一个公共接口。一个抽象类包含一个或多个具体派生类必须重写的纯虚函数。

在派生类中不能重写一个纯虚函数使得该类是抽象的。试图实例化抽象类的对象会导致编译错误。

一个抽象类至少有一个纯虚函数，一个抽象类也可以有数据成员和具体函数（concrete functions，包含构造函数和析构函数），它们受派生类继承的一般规则的约束。

抽象类的语法

◆ 带有纯虚函数的类称为抽象类：

```
class 类名
{
    virtual 类型 函数名(参数表)=0;
    //其他成员.....
}
```

虽然我们不能实例化抽象基类的对象，但是我们可以使用抽象基类来声明指针和引用，这些指针和引用可以指向从抽象类派生的任何具体类的对象。程序通常使用此类指针和引用对派生类对象进行多态操作。

11.6.2 Device Drivers: Polymorphism in Operating Systems

多态性对于实现分层的软件系统特别有效。例如，在操作系统中，每一种类型的物理设备都可能与其他类型的物理设备有很大的不同。即便如此，从设备和到设备分别读取数据或写入数据的命令可能具有一定的统一性。发送给设备驱动对象的写消息需要在该设备驱动的上下文中进行特定的解释，以及该设备驱动如何操作特定类型的设备。然而，写入调用本身与写入系统中的任何其他设备并无二致——将一定数量的字节从内存放到该设备上。面向对象的操作系统可以使用一个抽象的基类来提供一个适合所有设备驱动程序的接口。然后，通过对该抽象基类的继承，形成所有操作类似的派生类。设备驱动程序提供的(即,公共职能)功能在抽象基类中作为纯虚函数提供。这些纯虚函数的实现都是在对于特定类型设备驱动的派生类中提供的。这种架构还允许新的设备很容易地添加到系统中。用户只需在设备中插入并安装其新的设备驱动程序即可。操作系统通过其设备驱动程序与这个新设备"对话"，它具有与所有其他设备驱动程序相同的公共成员功能——设备驱动程序抽象基类中定义的那些。

11.7 RTTI(运行时类型识别)

运行时类型识别(runtime type information)(RTTI)，

C++ 的 RTTI (Run-Time Type Information) 是一种运行时类型信息机制，用于在程序运行时获取对象的类型信息。RTTI 主要包括两个关键字：typeid 和 dynamic_cast。

- typeid 运算符，用于返回表达式的类型。
- dynamic_cast 运算符，用于将基类的指针或引用安全地转换成派生类的指针或引用。

一般来说动态类型值得是某个基类指针或引用，指向一个派生类对象，且基类中有虚函数，此时会绑定对象的动态类型。

一般来说，只要有可能我们应该尽量使用虚函数。当操作被定义成虚函数时，编译器将根据对象的动态类型自动的选择正确的函数版本。然而，并非任何时候都能定义一个虚函数。假设我们无法使用虚函数，则可以使用一个RTTI运算符。

运算符dynamic_cast检查指针指向的对象的类型，然后确定该类型是否与指针正在转换的类型有is-a关系。如果是，则动态类型转换返回对象的地址。如果不是，则动态类型转换返回nullptr。

操作符typeid返回对type_info对象的引用，该对象包含关于操作数类型的信息，包括类型名称。为了使用typeid，程序必须包含头文件<typeinfo>。

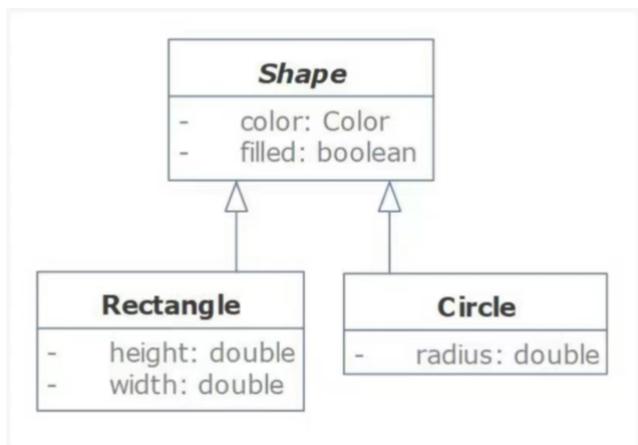
当被调用时，type_info成员函数name返回一个基于指针的字符串，该字符串包含type_info对象所表示类型的名称。

运算符dynamic_cast和typeid是C++的运行时类型信息(RTTI)特征的一部分，它允许程序在运行时确定对象的类型。

11.7.1 动态类型转换

1. 为何需要dynamic_cast?

```
1 void printObject(Shape& shape) // shape是派生类对象的引用
2 {
3     cout << "The area is "
4         << shape.getArea() << endl;
5     // 如果shape是Circle对象，就输出半径
6     // 如果shape是Rectangle对象，就输出宽高
7 }
```



如果需要修改函数，让它显示圆的半径应该怎么办？

2. Dynamic Casting Example

2.1. dynamic_cast 运算符

- (1) 沿继承层级向上、向下及侧向转换到类的指针和引用
- (2) 转指针：失败返回nullptr
- (3) 转引用：失败抛异常

2.2. 例子

先将Shape对象用dynamic_cast转换为派生类Circle对象

然后调用派生类中独有的函数

```
1 // A function for displaying a Shape object
2 void printObject(Shape &shape)
3 {
```

```

4   cout << "The area is "
5       << shape.getArea() << endl;
6   shape *p = &shape;
7   Circle *c = dynamic_cast<Circle*>(p);
8 // Circle& c = dynamic_cast<Circle&>(shape);
9 // 引用转换失败则抛出一个异常 std::bad_cast
10  if (c != nullptr) // 转换失败则指针为空
11  {
12      cout << "The radius is "
13      << p1->getRadius() << endl;
14      cout << "The diameter is "
15      << p1->getDiameter() << endl;
16  }
17 }
18

```

11.7.2 向上转换和向下转换

12. 流输入输出: A Deeper Look

12.1 Introduction

在C++ 程序中, 首选C++样式的I/O 而不是C样式的I/O。

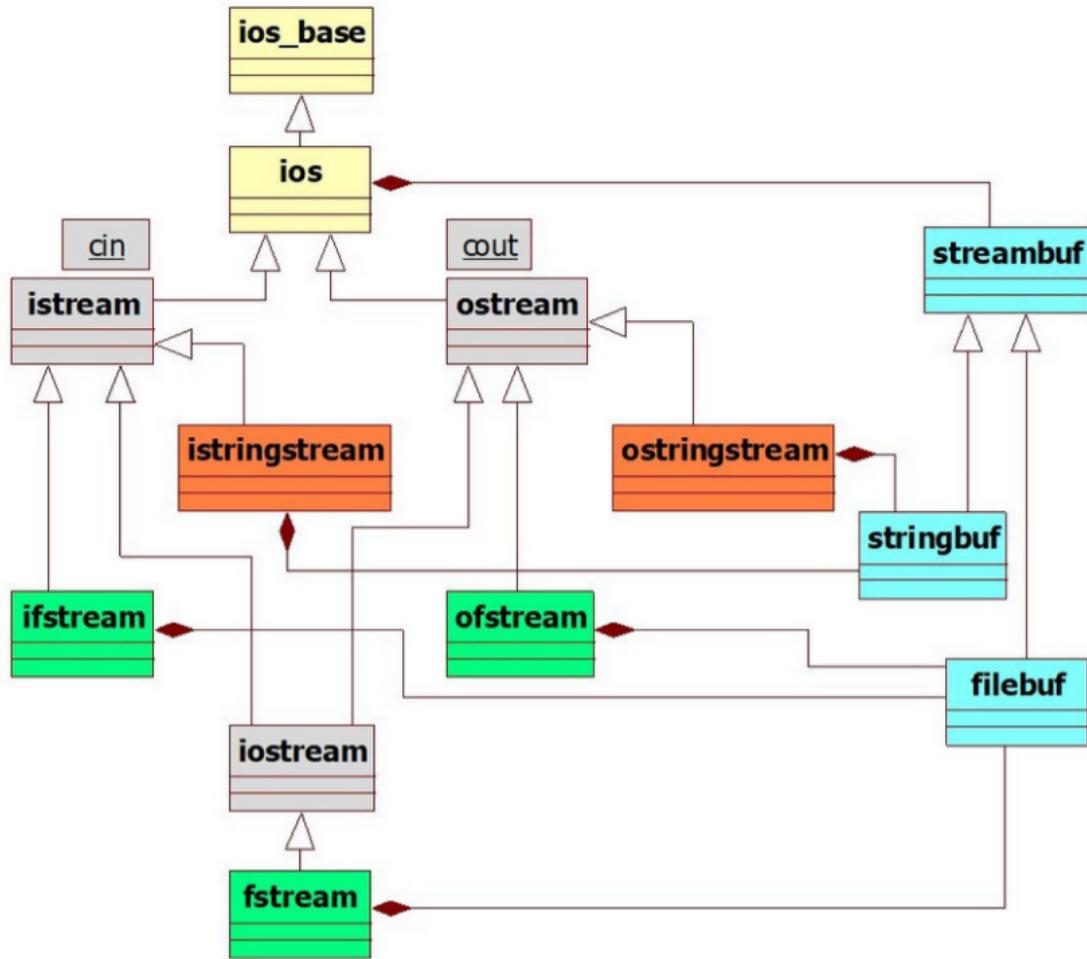
1. Comparision of File Manipulation between C and C++ (文件操作对比)

		C++	C
Header File (头文件)	file input	ifstream (<i>i: input; f:file</i>)	stdio.h
	file output	ofstream (<i>o: ouput; f:file</i>)	
	file input & output	fstream	
Read/ Write (读写 操作)	read from file (读文件)	>>; get(); get(char); get(char*); getline(); read(char*,streamsize);	fscanf(); fgets(char*, size_t , FILE*); fread(void *ptr, size, nitems, FILE *stream);
	write to file (写文件)	<<; put(char), put(int); write (const char*, streamsize); flush()	fprintf(); fwrite(const void *ptr, size, nitems, FILE *stream); fputs(const char*, FILE *);
	Status test (状态测试)	eof(); bad(); good(); fail()	feof(); ferror();

类名	说明	包含文件
抽象流基类		
ios	流基类	ios
输入流类		
istream	通用输入流类和其它输入流的基类	istream
ifstream	文件输入流类	fstream
istringstream	字符串输入流类	sstream
输出流类		
ostream	通用输出流类和其它输出流的基类	ostream
ofstream	文件输出流类	fstream
ostringstream	字符串输出流类	sstream
输入/输出流类		
iostream	通用输入/输出流类和其它输入/输出流的基类	istream
fstream	文件输入/输出流类	fstream
stringstream	字符串输入/输出流类	sstream
流缓冲区类		
streambuf	抽象流缓冲区基类	streambuf
filebuf	磁盘文件的流缓冲区类	fstream
stringbuf	字符串的流缓冲区类	sstream

12.2 流(Streams)

C++ I/O occurs in streams, which are sequences of bytes. C++ provides both “low-level” and “high-level” I/O capabilities. Low-level I/O capabilities (i.e., unformatted I/O) specify that some number of bytes should be transferred device-to-memory or memory-to-device. In such transfers, the individual byte is the item of interest. Such low-level capabilities provide high-speed, high-volume transfers but are not particularly convenient. Programmers generally prefer a higher-level view of I/O (i.e., formatted I/O), in which bytes are grouped into meaningful units, such as integers, floating-point numbers, characters, strings and user-defined types. These type-oriented capabilities are satisfactory for most I/O other than high-volume file processing.



C++的流类主要有五类：

1. 流基类 (ios_base和ios)
2. 标准输入输出流类 (istream/ostream/iostream)
3. 字符串流类 (istringstream/ostringstream)
4. 文件流类 (ifstream/ofstream/fstream)
5. 缓冲区类 (streambuf/stringbuf/filebuf)

12.2.1 Classic Streams vs. Standard Streams

以前，C++经典流库(classic stream libraries)只支持char类型基础的I/O操作。因为char类型只占用一个字节，它只能表示有限的一组字符(例如ASCII码)。然而许多语言使用的字母表中包含的字符数量多于单个字节char所能表示的数量。The ASCII character set does not provide these characters, but the Unicode® character set does. Unicode is an extensive international character set that represents the majority of the world's languages, mathematical symbols and much more. C++ provides Unicode support via the types wchar_t (the original C++ type for processing Unicode) and C++11 types char16_t and char32_t.

此外，标准流库(standard stream libraries)被实现为类模板式的类数组和向量。

12.2.2 iostream Library Headers

The <iostream> header defines the `cin`, `cout`, `cerr` and `clog` objects, which correspond to the standard input stream, the standard output stream, the unbuffered standard error stream and the buffered standard error stream, respectively. The <iomanip> header declares the parameterized stream manipulators such as `setprecision` and `setw` —for formatted I/O.

12.2.3 Stream Input/Output Classes and Objects

The iostream library defines each alias with the `typedef` specifier, which you'll sometimes use to create more readable type names. For example, the following statement defines the alias `CardPtr` as a synonym for type `Card*`:

```
1 | typedef Card* CardPtr;
```

12.3 流输出(Stream Output)

`cout`与`cin`本质上是对象。

格式化和非格式化的输出能力由`ostream`提供。 Capabilities include output of standard data types with the stream insertion operator (`<<`); output of characters via the `put` member function; unformatted output via the `write` member function; output of integers in decimal, octal and hexadecimal formats; output of floatingpoint values with various precision, with forced decimal points, in scientific notation (e.g., `1.234567e-03`) and in fixed notation (e.g., `0.00123457`); output of data justified in fields of designated widths; output of data in fields padded with specified characters; and output of uppercase letters in scientific notation and hexadecimal notation.

C++中最重要的三个输出流：`ostream`\`ofstream`\`ostringstream`还有三个预定义好的输出流对象：`cout`\`cerr`\`clog`

预先定义的输出流对象

- ◊ `cout` 标准输出。
- ◊ `cerr` 标准错误输出，没有缓冲，发送给它的内容立即被输出。
- ◊ `clog` 类似于`cerr`，但是有缓冲，缓冲区满时被输出。

构造输出流对象

- ◊ `ofstream`类支持磁盘文件输出
- ◊ 如果在构造函数中指定一个文件名，当构造这个文件时该文件是自动打开的
 - `ofstream myFile("filename");`
- ◊ 可以在调用默认构造函数之后使用`open`成员函数打开文件
 - `ofstream myFile; //声明一个静态文件输出流对象`
 - `myFile.open("filename"); //打开文件，使流对象与文件建立联系`
- ◊ 在构造对象或用`open`打开文件时可以指定模式
 - `ofstream myFile("filename", ios_base::out | ios_base::binary);`

文件输出流成员函数

- ◊ `open`函数
 - 把流与一个特定的磁盘文件关联起来；
 - 需要指定打开模式。
- ◊ `put`函数
 - 把一个字符写到输出流中。
- ◊ `write`函数
 - 将内存中的一块内容写到一个文件输出流中。
- ◊ `seekp`和`tellp`函数
 - 操作文件流的内部指针。
- ◊ `close`函数
 - 关闭与一个文件输出流关联的磁盘文件。
- ◊ 错误处理函数
 - 在写到一个流时进行错误处理。

12.3.1. Output of char* Variables

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     const char* const word{"again"};
6
7     cout<<"Value of word is: "<<word
8         <<"\nValue of static_cast<const void*>(word) is:"
9         <<static_cast<const void*>(word)<<endl;
10 }
```

在上述代码中，char类型的指针指向字符串again的首字符地址。

在 C++ 程序中，字符串可以存储在字符数组中，我们可以用指针访问数组，自然也就可以用指针访问字符串。

字符串指针本质是一个 char 类型的指针，然后将它指向一个字符串的开头，即字符串中的第一个字符。

12.3.2 C风格字符串

C 语言约定：如果字符型 (char) 数组的末尾包含了空字符\0 (也就是 0) ，那么该数组中的内容就是一个字符串。 ↵

这是一个字符数组，不是字符串。

w	w	w	.	f	r	e	e	c	p	l	u	s	.	n	e	t
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

这是一个字符数组，也是一个字符串。

w	w	w	.	f	r	e	e	c	p	l	u	s	.	n	e	t	\0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

空字符



用字符数组存储字符串（C风格字符串）

◆ 例如

```
char str[8] = { 'p', 'r', 'o', 'g', 'r', 'a', 'm', '\0' };
char str[8] = "program";
char str[] = "program";
```

p	r	g	g	r	a	m	\0
---	---	---	---	---	---	---	----

用字符数组表示字符串的缺点

- ◆ 执行连接、拷贝、比较等操作，都需要显式调用库函数，很麻烦
- ◆ 当字符串长度很不确定时，需要用new动态创建字符数组，最后要用delete释放，很繁琐
- ◆ 字符串实际长度大于为它分配的空间时，会产生数组下标越界的错误

字符串指针

用指针指向字符串，和指向数组没有什么区别，要么在定义指针后给指针赋值，要么在定义时就初始化。

例如：

```
1 char str[] = "http://c.biancheng.net";
2 char* p = str; // 初始化字符指针，令其指向 str 数组中的字符串
3
4 char* q; // 定义一个字符指针
5 q = str; // 令其指向字符串的开头
```

实际场景中，我们还会经常见到如下定义的字符串指针：

```
1 char *p = "http://c.biancheng.net";
```

字符串“Hello World!”存放在内存中的常量区，指针 p 指向常量区中的这个字符串。

注意，常量区的内容是不允许修改的，因此对于上面定义的字符串 "<http://c.biancheng.net>"，不可以通过指针 p 修改它。

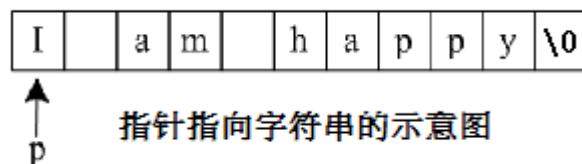
字符指针指向字符串的方法：

由于字符串在内存中连续存储的特点，可以使用指针进行操作，并且指针必须是字符型的。通常将指针指向字符串的首地址，利用指针的后移可以指向后续字符。字符指针指向字符串一般有三种的方法：

1. 为字符指针初始化字符串首地址

例如：char *p = "I am happy";

这时字符串常量 "I am happy" 在内存中占用11个字节，以上初始化过程是将这段存储空间的首地址赋给字符型指针p，我们称指针p指向字符串"I am happy"，存储示意图如下所示：



字符串指针变量的定义说明与指向字符变量的指针变量说明只能按对指针变量的赋值不同来区别。

例如：

char c,*p=&c;

表示p是一个指向字符变量c的指针变量。

而：

char *s="C Language";

则表示s是一个指向字符串的指针变量。把字符串的首地址赋予s。

2. 为字符指针赋值字符串首地址

例如：

```
1 | char *p;  
2 | p = "I am happy";
```

以上赋值语句“p = "I am happy";”是将字符串存储空间的首地址赋给指针p，然后p便指向字符串。注意不是将字符串赋给指针p。

3. 让指针指向存放字符串的数组

```
1 | char a[20] = "I am happy", *p = a;
```

以上定义了字符数组a并为之初始化字符串"I am happy", 然后定义了字符指针p, 将p初始化为指向数组a的首地址, 即指向了字符串的第一个字符 'I'。

一般来说, 如果给cout提供一个指针, 它将打印地址。但如果指针的类型为char*, cout将显示指向的字符串。如果要显示字符串的地址, 则必须将这种指针强制转换成另一种指针类型。例如:

```
1 | char ch = 'A';
2 | cout << &ch; // A
```

对于上述代码, 从类型的角度而言, &ch的类型为char*, 而字符串的类型为const char*, 字符数组退化后的类型为char*, 因此类型的角度而言无法区分它指的是字符串的地址还是字符的地址, 统一按字符串处理。

字符串的类型实际上是由常量字符构成的数组(array)。而字符数组的类型为const char**, 所以字符串的类型为const char**

尽管C++支持C风格字符串, 但在C++程序中最好不要使用C风格的字符串。因为使用起来不方便, 且容易引发程序漏洞。C风格字符串不是一种类型, 而是为了表达和使用字符串而形成的一种约定俗成的写法。按照此习惯书写的字符串存放在字符数组中并以空字符结束。

比较字符串

比较两个C风格字符串的方法和之前学习过的比较标准库string对象的方法大相径庭。比较标准库string对象的时候, 用的是普通的关系运算符和相等性运算符:

```
string s1 = "A string example";
string s2 = "A different string";
if (s1 < s2) // false: s2 小于 s1
```

如果把这些运算符用在两个C风格字符串上, 实际比较的将是指针而非字符串本身:

```
const char ca1[] = "A string example";
const char ca2[] = "A different string";
if (ca1 < ca2) // 未定义的: 试图比较两个无关地址
```

12.3.2 Character Output Using Member Function put

basic_ostream的成员函数put, 一次输出一个字符。例如:

```
1 | cout.put('A');
```

会显示字符'A'。The put function also may be called with a numeric expression that represents an ASCII value, as in the following statement, which also outputs A:

```
1 | cout.put(65);
```

12.4 流输入(Stream Input)

The stream extraction operator (>>) normally skips white-space characters (such as blanks, tabs and newlines) in the input stream.

重要的输入流类：

- ◆ `istream`类最适合用于顺序文本模式输入。`cin`是其实例
- ◆ `ifstream`类支持磁盘文件输入
- ◆ `istringstream`

构造输入流对象

- ◆ 如果在构造函数中指定一个文件名，在构造该对象时该文件便自动打开。
`ifstream myFile("filename");`
- ◆ 在调用默认构造函数之后使用`open`函数来打开文件。
`ifstream myFile; //建立一个文件流对象`
`myFile.open("filename"); //打开文件"filename"`
- ◆ 打开文件时可以指定模式
`ifstream myFile("filename", ios_base::in | ios_base::binary);`

输入流相关函数

- ◆ `open`函数把该流与一个特定磁盘文件相关联。
- ◆ `get`函数的功能与提取运算符 (>>) 很相像，主要的不同点是`get`函数在读入数据时包括空白字符。
- ◆ `getline`的功能是从输入流中读取多个字符，并且允许指定输入终止字符，读取完成后，从读取的内容中删除终止字符。
- ◆ `read`成员函数从一个文件读字节到一个指定的内存区域，由长度参数确定要读的字节数。当遇到文件结束或者在文本模式文件中遇到文件结束标记字符时结束读取。

字符串输入流 (`istringstream`)

- ◊ 用于从字符串读取数据
- ◊ 在构造函数中设置要读取的字符串
- ◊ 功能
 - 支持`ifstream`类的除`open`、`close`外的所有操作
- ◊ 典型应用
 - 将字符串转换为数值

12.4.1 `get` and `getline` 成员函数

>>运算符用空格分隔数据，即，在使用此运算符读取数据时，遇到空格就停止读取。

不带参数的 `get` 成员函数从指定的流中输入一个字符（包括空格字符和其他非图形字符，例如表示文件末尾的键序列），并将其作为函数调用的值返回。当在流上遇到文件末尾时，此版本的 `get` 将返回 `EOF`。`EOF` 通常具有值 `-1`，并在标头中定义，该标头通过流库标头间接包含在代码中`<iostream>`。

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int character; // **use int, because char cannot represent
6     // EOF**
7
8     // prompt user to enter line of text
9     cout << "Before input, cin.eof() is " << cin.eof()
10    << "\nEnter a sentence followed by Enter and end-of-
11    file:\n";
12
13    // use get to read each character; use put to display it
14    while((character=cin.get())!=EOF){
15        cout.put(character);
16
17        // display end-of-file character
18        cout << "\nEOF in this system is: " << character
19        << "\nAfter input of EOF, cin.eof() is " << cin.eof() <<
20        endl;
```

```

Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions
Testing the get and put member functions
^Z

EOF in this system is: -1
After input of EOF, cin.eof() is 1

```

使用int类型的值，因为在许多平台上，char只能表示非负值，但是EOF为-1。

此外，我们还可以用输入输出流对象的一些成员函数来实现输入和输出。

- `cout.put()` 输出单个字符，可以连续输出
- `cin.get()` 读入一个字符（包括空白字符），返回读入成功的字符，如遇到文件结束符，返回 EOF
- `cin.get(ch)` 读入一个字符并赋值给变量 ch，成功读入则返回真
- `cin.get(字符数组或指针, 字符个数 n, 终止字符)` 读入 n-1 个字符，如遇到终止字符则提前结束
- `cin.getline(字符数组或指针, 字符个数 n, 终止字符)` 与上面的 `cin.get` 类似，但是遇到终止字符时，**字符指针会移到该终止字符后面，而 `cin.get` 则会停留在原位置**
- `cin.eof()` 如果到达文件末尾（遇文件终止符）返回真，否则返回假
- `cin.peek()` 返回当前指针指向的字符，但只是观测，指针仍然停留在当前位置
- `cin.putback(ch)` 将字符 ch 返回到输入流，插入到当前指针位置
- `cin.ignore(n, 终止字符)` 跳过输入流中 n 个字符，若遇到终止符提前结束，此时指向终止字符后面一个位置

get()函数是cin输入流对象的成员函数，`cin.get()`从流中读取并取走一个字符，有三种形式：

1. 无参数的；
2. 有一个参数的；

3. 有3个参数的。

1. 无参数的

其调用形式为

```
1 | cin.get()
```

用来从指定的输入流中提取一个字符（包括空白字符），函数的返回值就是读入的字符。若遇到输入流中的文件结束符，则函数值返回文件结束标志EOF(End Of File)，一般以-1代表EOF，用-1而不用0或正值，是考虑到不与字符的ASCII代码混淆，但不同的C++系统所用的EOF值有可能不同。

```
1 | #include <iostream>
2 | using namespace std;
3 |
4 | int main()
5 | {
6 |     char c;
7 |     cout << "enter a sentence:" << endl;
8 |     while ((c = cin.get()) != EOF){
9 |         cout.put(c);
10 |     }
11 |
12 |     cout << "end" << endl;
13 |
14 |     return 0;
15 | }
```

2. 有一个参数的

```
1 | cin.get(ch)
```

其作用是从输入流中读取一个字符，赋给字符变量ch。如果读取成功则函数返回true，如失败(遇文件结束符)则函数返回false (文件结束符为 $ctrl + z$)。上面的例子可以改写如下：

```
1 | #include <iostream>
2 | using namespace std;
3 |
4 | int main()
5 | {
```

```
6     char c;
7     cout << "enter a sentence:" << endl;
8
9     while (cin.get(c)) //读取一个字符赋给字符变量c,如果读取成
10        //功, cin.get(c)为真
11        //读取字符为文件结束符 (ctrl + z) 时,
12        //cin.get(c)为假
13    {
14        cout.put(c);
15    }
16    cout << "end" << endl;
17
18    return 0;
19 }
```

3. 有三个参数的

```
1     cin.get(字符数组, 字符个数n, 终止字符)
2 或
3     cin.get(字符指针, 字符个数n, 终止字符)
```

其作用是从输入流中读取n-1个字符，赋给指定的字符数组(或字符指针指向的数组)，如果在读取n-1个字符之前遇到指定的终止字符，则提前结束读取。如果读取成功则函数返回true(真)，如失败(遇文件结束符)则函数返回false(假)。如果在函数原型中省略终止字符，默认为'\n'。

具有三个参数 (字符数组、大小限制和分隔符 (具有默认值换行符)) 的成员函数 get 从输入流中读取字符，最多读 (大小限制数目-1) 个字符，或直到读取分隔符。输入字符串以 null 字符结尾。分隔符不放置在字符数组中，而是保留在输入流中。

将上例改写如下：

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     char ch[20];
6     cout << "enter a sentence:" << endl;
7     cin.get(ch, 10, '\n');//指定换行符为终止字符
8     cout << ch << endl;
9     system("pause");
10    return 0;
11 }
```

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     // create two char arrays, each with 80 elements
7     const int SIZE{80};
8     char buffer1[SIZE];
9     char buffer2[SIZE];
10
11     // use cin to input characters into buffer1
12     cout << "Enter a sentence:\n";
13     cin >> buffer1;
14
15     // display buffer1 contents
16     cout << "\nThe string read with cin was:\n" << buffer1 <<
17     "\n\n";
18
19     // use cin.get to input characters into buffer2
20     cin.get(buffer2, SIZE);
21
22     // display buffer2 contents
23     cout << "The string read with cin.get was:\n" << buffer2 <<
24     endl;
25 }
```

`cin` 这个函数输入取数据的过程：

```
1 // cin如何读取
2     int a, b;
3     cin >> a >> b;
```

大家肯定知道，假定我们在这里输入3 5 8 9 11，a最终会读取的数字是3。但是实际上cin是将3 5 8 9 11都放入了缓冲区，等待a去读取，a从头开始读，读到int类型的数据，并将3拿出来赋给a,这时候缓冲区内还剩下：5 8 9 11。注意：这里3后面的' '是存在的，缓冲区内的字符会一直留着直到后面被读取。然后接着b再读取，读到的是数字5，然后5便赋给b,剩下的字符留在缓冲区等待程序后续操作。

成员函数getline的操作类似于三参数get成员函数。getline函数从输入流中删除分隔符，但不将其存储在字符串中。

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     const int SIZE{80};
6     char buffer[SIZE]; // create array of 80 characters
7
8     // input characters in buffer via cin function getline
9     cout << "Enter a sentence:\n";
10    cin.getline(buffer, SIZE);
11
12    cout << "\nThe sentence entered is:\n" << buffer << endl;
13 }
```

12.4.2 istream Member Functions peek, putback and ignore

istream的成员函数ignore读取并舍弃字符。接收两个实参：

- 1.指定字符数-默认值为1
- 2.停止忽略字符的分隔符 - 默认分隔符为EOF。

该函数将丢弃指定数量的字符，如果在输入流中遇到分隔符，则丢弃更少的字符。

它的原型是：

```
1 istream & ignore(int n =1, int delim = EOF);
```

此函数的作用是跳过输入流中的 n 个字符，或跳过 delim 及其之前的所有字符，哪个条件先满足就按哪个执行。两个参数都有默认值，因此 `cin.ignore()` 就等效于 `cin.ignore(1, EOF)`，即跳过一个字符。

例如：

```
1 | ignore(6, '\n');
```

这里的意思是说从缓冲区的第一个字符开始读，读到第6个字符，这些字符我们就全部从缓冲区里舍去，假如这段字符长这样hello c++，那么现在缓冲区里还剩下c++，这里的字符是一个一个来的，假如是数字，例如：13 1 89 72，调用了`cin.ignore(6, '\n')`之后呢，缓冲区里还剩下9 72，没错，89变成了9，就是因为舍去的是前六位字符而不是数字，所以这里也是很多人容易错的地方，觉得是舍弃6个整数。

接着，看'\n'，这个的意思就简单了，只要你输入了'\n'，也就是我们键入回车的时候，`ignore()`便不再从缓冲区里舍弃字符了，也就是你回车之前的所有字符全部从缓冲区里拿出来。这个'\n'也可以定义为其他字符，只要遇到这个字符`ignore()`便停止他的行为。

```
1 | //ignore()用法
2 | #include <iostream>
3 |
4 | using namespace std;
5 |
6 | int main()
7 | {
8 |     int a, b;
9 |     cin >> a;
10 |    //从缓冲区舍弃字符，到第五个停止，如果遇到换行符，舍弃换行符之前的所有
11 |    //字符
12 |    cin.ignore(5, '\n');
13 |    cin >> b;
14 |    cout << "a is : " << a << endl;
15 |    cout << "b is : " << b << endl;
16 |    return 0;
17 | }
```

输入 25 86 91 12，首先 25 被 a 从缓冲区中拿出来，此时缓冲区剩下 86 91 12，然后开始舍弃，数到第五位，也就是 91 中 9 的位置，9 和它之前的数据被全部舍弃，b 再进去读取，便读取到 1，输出正确。

```
D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
25 86 91 12
a is : 25
b is : 1
```

这里我们在数字8后键入了回车，按照前面所说，`a`拿走了`25`，回车将`8`以及它之前的数据全部丢掉，再给`b`进去读取，因此直接读取`96`

```
D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
25 8
96 12
a is : 25
b is : 96
```

`cin.ignore()`,大可以把`cin.ignore()`理解为`cin.ignore(1, '\n')`,就是舍弃掉第一个字符或者回车之前的数据，我们可以用它舍弃掉缓冲区里的空格字符或者存在的换行符。

```
1 // 代码差不多
2 #include <iostream>
3
4 using namespace std;
5
6 int main()
7 {
8     int a, b;
9     cin >> a;
10    cin.ignore(); //唯一的区别在这！
11    cin >> b;
12    cout << "a is : " << a << endl;
13    cout << "b is : " << b << endl;
14    return 0;
15 }
```

`putback`成员函数将`get`函数从输入流中获取的上一个字符放回该流中。

`peek`成员函数返回输入流的下一个字符，但并不会将此字符从流中移除。

12.5 Unformatted I/O Using `read`, `write` and `gcount`

未格式化的输入/输出使用`istream`的`read`函数和`ostream`的`write`成员函数。`read`将字节放入`char`类型的数组中。`write`将`char`数组中的字节输出。例如：

```
1 | char buffer[] {"HAPPY BIRTHDAY"};
2 | cout.write(buffer, 10);
```

输出buffer中的前十个字节(包括空字符),

```
1 | cout.write("ABCDEFGHIJKLMNPQRSTUVWXYZ", 10);
```

上述代码将会输出字母表的前十个字母。

成员函数gcount报告上次输入操作读取的字符数量。

```
1 | #include <iostream>
2 | using namespace std;
3 |
4 | int main(){
5 |     const int SIZE{80};
6 |     char buffer[SIZE];
7 |
8 |     // use function read to input characters into buffer
9 |     cout << "Enter a sentence:\n";
10 |    cin.read(buffer, 20);
11 |
12 |    // use functions write and gcount to display buffer
13 |    // characters
14 |    cout << "\nThe sentence entered was:\n";
15 |    cout.write(buffer, cin.gcount());
16 |    cout << endl;
17 | }
```

运行结果为:

```
D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
Enter a sentence:
Using the read, write, and gcount member functions

The sentence entered was:
Using the read, writ
```

12.6 Stream Manipulators(操纵符): 格式化输出

操纵符 (manipulator)

- 插入运算符与操纵符一起工作
 - 控制输出格式。
- 很多操纵符都定义在
 - ios_base类中 (如hex()) 、 <iomanip>头文件 (如setprecision()) 。
- 控制输出宽度
 - 在流中放入setw操纵符或调用width成员函数为每个项指定输出宽度。
- setw和width仅影响紧随其后的输出项，但其它流格式操纵符保持有效直到发生改变。
- dec、 oct和hex操纵符设置输入和输出的默认进制。

setiosflags的参数 (流的格式标识)

- ios_base::skipws 在输入中跳过空白。
- ios_base::left 左对齐值，用填充字符填充右边。
- ios_base::right 右对齐值，用填充字符填充左边 (默认对齐方式)。
- ios_base::internal 在规定的宽度内，指定前缀符号之后，数值之前，插入指定的填充字符。
- ios_base::dec 以十进制形式格式化数值 (默认进制)。
- ios_base::oct 以八进制形式格式化数值。
- ios_base::hex 以十六进制形式格式化数值。
- ios_base::showbase 插入前缀符号以表明整数的数制。
- ios_base::showpoint 对浮点数值显示小数点和尾部的0。
- ios_base::uppercase 对于十六进制数值显示大写字母A到F，对于科学格式显示大写字母E。
- ios_base::showpos 对于非负数显示正号 (“+”)。
- ios_base::scientific 以科学格式显示浮点数值。
- ios_base::fixed 以定点格式显示浮点数值 (没有指数部分)。
- ios_base::unitbuf 在每次插入之后转储并清除缓冲区内容。

12.6.1 Integral Stream Base: dec, oct, hex and setbase

```
1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 int main(){
6     int number;
7
8     cout<<"Enter a decimal number: ";
```

```

9  cin >> number; // input number
10
11 // use hex stream manipulator to show hexadecimal number
12 cout << number << " in hexadecimal is: " << hex << number <<
13 "\n";
14
15 // use oct stream manipulator to show octal number
16 cout << dec << number << " in octal is: " << oct << number <<
17 "\n";
18
19 // use setbase stream manipulator to show decimal number
20 cout << setbase(10) << number << " in decimal is: " << number
21 << endl;
22
23 }
```

```

D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
Enter a decimal number:23
23 in hexadecimal is: 17
23 in octal is: 27
23 in decimal is: 23
```

12.6.2 设置浮点数精度(precision, setprecision)

使用setprecision流操作符或者ostream的成员函数precision控制浮点数的精度。使用setprecision流操作符会改变随后操作的浮点数的精度，直到下一次使用setprecision。使用成员函数precision会返回当前使用的浮点数的精度。头文件为<iomanip>

setprecision(0)的实际效果取决于编译器，不同的编译器实现是不同的。

精度

- 如果不指定fixed或scientific，精度值表示有效数位数。
- 如果设置了ios_base::fixed或ios_base::scientific精度值表示小数点之后的位数。

```

1 #include <iostream>
2 #include <iomanip>
3 #include <cmath>
4 using namespace std;
5
```

```
6 int main(){
7     double root2{sqrt(2.0)}; // calculate square root of 2
8
9     cout << "Square root of 2 with precisions 0-9.\n"
10    << "Precision set by ostream member function
precision:\n";
11    cout << fixed; // use fixed-point notation
12
13    // display square root using ostream function precision
14    for (int places{0}; places <= 9; ++places){
15        cout.precision(places);
16        cout << root2 << "\n";
17    }
18
19    cout << "\nPrecision set by stream manipulator
setprecision:\n";
20
21    // set precision for each digit, then display square root
22    for (int places{0}; places <= 9; ++places){
23        cout << setprecision(places) << root2 << "\n";
24    }
25 }
```

运行结果如下：

```
Square root of 2 with precisions 0-9.  
Precision set by ostream member function precision:  
1  
1.4  
1.41  
1.414  
1.4142  
1.41421  
1.414214  
1.4142136  
1.41421356  
1.414213562
```

```
Precision set by stream manipulator setprecision:  
1  
1.4  
1.41  
1.414  
1.4142  
1.41421  
1.414214  
1.4142136  
1.41421356  
1.414213562
```

12.6.3 Field Width (width, setw)

istream和ostream类的成员函数width()设置field width(即，应输出值的字符位置数或应输入的最大字符数)。

If values output are narrower than the field width, fill characters are inserted as padding. A value wider than the designated width will not be truncated—the full number will be printed. The width function with no argument returns the current setting.

宽度设置仅适用于下一次插入或提取 (即宽度设置not sticky) ;之后，将宽度隐式设置为0 (即，输入和输出将使用默认设置执行)。假设宽度设置适用于所有后续输出是一个逻辑错误。

如果输出的数值占用的宽度超过setw(int n)设置的宽度，则按照实际宽度输出。

```
1 float f=0.364823;  
2 std::cout<<std::setw(3)<<f<<std::endl;
```

上述代码的输出结果为：

```
D:\Code\cppProject\test240415\cmake-build-debug\test240415.exe
0.364823
```

cout.width()默认为右对齐，且只对后面第一个输出有作用。

cin.width()遇到空格自动停止接收。例如对于下列代码，输入This is a test of the width member function。cin.width(5)，遇到空格停止读入，所以只读入了This，使用cout.width(widthValue++)输出，widthValue先使用再递增，所以此时为cout.width(4)。

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int widthvalue{4};
6     char sentence[10];
7
8     cout << "Enter a sentence:\n";
9     cin.width(5); // input only 5 characters from sentence
10
11     // set field width, then display characters based on that
12     // width
13     while(cin>>sentence){
14         cout.width(widthvalue++);
15         cout<<sentence<<"\n";
16         cin.width(5);
17     }
18 }
```

```
D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
Enter a sentence:
This is a test of the width member function
This
    is
        a
    test
        of
    the
    width
        h
    member
        func
    tion
    
```

12.6.4 User-Defined Output Stream Manipulators

```
1 #include <iostream>
2 using namespace std;
3
4 // bell manipulator (using escape sequence \a)
5 ostream& bell(ostream& output){
6     return output << '\a'; // issue system beep
7 }
8
9 // carriageReturn manipulator (using escape sequence \r)
10 ostream& carriageReturn(ostream& output){
11     return output << '\r'; // issue carriage return
12 }
13
14 // tab manipulator (using escape sequence \t)
15 ostream& tab(ostream& output) {
16     return output << '\t'; // issue tab
17 }
18
19 // endLine manipulator (using escape sequence \n and flush stream
20 // manipulator to simulate endl)
21 ostream& endLine(ostream& output){
```

```
22     return output << '\n' << flush; // issue endl-like end of
23     line
24
25 int main(){
26     // use tab and endl manipulators
27     cout << "Testing the tab manipulator:" << endl
28         <<'a'<<tab<<'b'<<tab<<'c'<<endl;
29
30     cout << "Testing the carriageReturn and bell manipulators:"
31         <<endl<<".....";
32
33     cout<<bell;//use bell manipulator
34
35     // use carriageReturn and endl manipulators
36     cout << carriageReturn<< "----" << endl;
37 }
```

```
D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
Testing the tab manipulator:
a      b      c
Testing the carriageReturn and bell manipulators:
-----.

进程已结束，退出代码为 0
```

上述代码创建了自己定义的流操作符。

对于输出流操作符，返回类型和形参必须是ostream&类型。

12.7 格式化输出(Stream Format States and Stream Manipulators)

Manipulator	Description
<code>skipws</code>	<i>Skip white-space characters</i> on an input stream. This setting is reset with stream manipulator <code>noskipws</code> .
<code>left</code>	<i>Left justify</i> output in a field. <i>Padding</i> characters appear to the <i>right</i> if necessary.
<code>right</code>	<i>Right justify</i> output in a field. <i>Padding</i> characters appear to the <i>left</i> if necessary.
<code>internal</code>	Indicate that a number's <i>sign</i> should be <i>left justified</i> in a field and a number's <i>magnitude</i> should be <i>right justified</i> in that same field (i.e., <i>padding</i> characters appear <i>between</i> the sign and the number).
<code>boolalpha</code>	Specify that <i>bool values</i> should be displayed as the word <code>true</code> or <code>false</code> . The manipulator <code>noboolalpha</code> sets the stream back to displaying bool values as 1 (true) and 0 (false).
Manipulator	Description
<code>dec</code>	Specify that integers should be treated as <i>decimal</i> (base 10) values.
<code>oct</code>	Specify that integers should be treated as <i>octal</i> (base 8) values.
<code>hex</code>	Specify that integers should be treated as <i>hexadecimal</i> (base 16) values.
<code>showbase</code>	Specify that the <i>base</i> of a number is to be output <i>ahead</i> of the number (a leading 0 for octals; a leading 0x or 0X for hexadecimals). This setting is reset with stream manipulator <code>noshowbase</code> .
<code>showpoint</code>	Specify that floating-point numbers should be output with a <i>decimal point</i> . This is used normally with <code>fixed</code> to <i>guarantee</i> a certain number of digits to the <i>right</i> of the decimal point, even if they're zeros. This setting is reset with stream manipulator <code>noshowpoint</code> .
<code>uppercase</code>	Specify that <i>uppercase letters</i> (i.e., X and A through F) should be used in a <i>hexadecimal</i> integer and that <i>uppercase E</i> should be used when representing a floating-point value in <i>scientific notation</i> . This setting is reset with stream manipulator <code>noupper-case</code> .
<code>showpos</code>	Specify that <i>positive</i> numbers should be preceded by a plus sign (+). This setting is reset with stream manipulator <code>noshowpos</code> .
<code>scientific</code>	Specify output of a floating-point value in <i>scientific notation</i> .
<code>fixed</code>	Specify output of a floating-point value in <i>fixed-point notation</i> with a specific number of digits to the <i>right</i> of the decimal point.

Fig. 13.10 | Format state stream manipulators from `<iostream>`. (Part 2 of 2.)

12.7.1 Trailing Zeros and Decimal Points (showpoint)

流操作符`showpoint`是一个sticky setting，强迫浮点数数字以其小数点和尾随零输出。例如，浮点数79.0输出为79，在不使用`showpoint`的情况下。或者使用`showpoint`输出79.00000。重置`showpoint`设置，使用流操作符`noshowpoint`。默认的浮点数精度为6，不使用`fixed`和`scientific`操作符，精度表示的是有效数字，而不是小数点以后的。例如9.9000，输出9.9。9.990输出9.99。例如下列程序：

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     // display double values with default stream format
6     cout << "Before using showpoint"
7     <<"\n9.9900 prints as: " << 9.9900
8     << "\n9.9000 prints as: " << 9.9000
9     <<"\n9.0000 prints as: " << 9.0000;
10
11    // display double value after showpoint
12    cout<<showpoint
13    <<"\n\nAfter using showpoint"
14    <<"\n9.9900 prints as: " << 9.9900
15    <<"\n9.9000 prints as: " << 9.9000
16    <<"\n9.0000 prints as: " << 9.0000 << endl;
17 }

```

```

D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
Before using showpoint
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9

After using showpoint
9.9900 prints as: 9.99000
9.9000 prints as: 9.90000
9.0000 prints as: 9.00000

```

进程已结束，退出代码为 0

12.7.2 Justification (left, right and internal)

使用流操作符left或者right可以让要输出的内容在指定的宽度内左对齐或者右对齐。

```

1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4 int main() {
5     int x{12345};
6

```

```

7   // display x right justified (default)
8   cout << "Default is right justified:\n\"" << setw(10) << x <<
9   "\\"";
10  // use left manipulator to display x left justified
11  cout << "\n\nuse left to left justify x:\n\""
12    << left << setw(10) << x << "\\"";
13
14  // use right manipulator to display x right justified
15  cout << "\n\nuse right to right justify x:\n\""
16    << right << setw(10) << x << "\\" << endl;
17 }

```

```

D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
Default is right justified:
"      12345"

Use left to left justify x:
"12345      "

Use right to right justify x:
"      12345"

进程已结束，退出代码为 0

```

流操作符internal指出数字的正负号(或者使用流操作符showbase显示数字使用的进制)应该在指定的输出宽度的最左边输出，数字的数量应该靠右显示。中间的空白区域使用填充字符填充，默认情况下为空格。showpos操作符使得正数的符号+强制显示。例如下述代码：

```

1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 int main(){
6 // display value with internal spacing and plus sign
7     cout<<internal<<showpos<<setw(10)<<123<<endl;
8 }

```

```
D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
+      123
```

将123换成-123后，输出结果为：

```
D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
-      123
```

对于指定的宽度10，数字-123，-号在最左面显示，123在最右边显示，中间以空格字符填充。

12.7.3 设置填充字符(fill, setfill)

对于指定的字段宽度，使用fill成员函数指定填充字符。默认情况下的填充字符为空格。函数返回先前的填充字符。setfill操作符也可以指定填充字符。

```
1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 int main() {
6     int x{10000};
7
8     // display x
9     cout << x << " printed as int right and left justified\n"
10    << "and as hex with internal justification.\n"
11    << "Using the default pad character (space):\n";
12
13    // display x
14    cout << setw(10) << x << "\n";
15
16    // display x with left justification
17    cout << left << setw(10) << x << "\n";
18
19    // display x with base as hex with internal justification
20    cout << showbase << internal << setw(10) << hex << x <<
21    "\n\n";
22
23    cout << "Using various padding characters:" << endl;
24
25    // display x using padded characters (right justification)
```

```

25     cout<<right;
26     cout.fill('*');
27     cout << setw(10) << dec << x << "\n";
28
29     // display x using padded characters (left justification)
30     cout << left << setw(10) << setfill('%') << x << "\n";
31
32     // display x using padded characters (internal justification)
33     cout << internal << setw(10) << setfill('A') << hex << x <<
34     endl;
35 }
```

D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
10000 printed as int right and left justified
and as hex with internal justification.

Using the default pad character (space):

```

10000
10000
0x    2710
```

Using various padding characters:

```

*****10000
10000%%%%%
0x^^^^2710
```

进程已结束，退出代码为 0

12.7.4 Integral Stream Base (dec, oct, hex, showbase)

C++提供流操作符dec,hex和oct让整数以十进制、十六进制和八进制的形式显示。如果没有指定以哪种进制显示，默认以十进制显示。以0开头的数字为八进制，以0x或者0X开头的数字为十六进制。

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int x{100};
6
7     // use showbase to show number base
```

```

8     cout << "Printing octal and hexadecimal values with
9     showbase:\n"
10
11     <<showbase;
12
13     cout << x << endl; // print decimal value
14     cout << oct << x << endl; // print octal value
15     cout << hex << x << endl; // print hexadecimal value
16 }
```

```

D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
Printing octal and hexadecimal values with showbase:
100
0144
0x64
```

12.7.5 Floating-Point Numbers; Scientific and Fixed Notation (scientific, fixed)

浮点值可以四舍五入到若干位有效数或精度，这是出现在小数点前后的总位数。可以通过使用 `setprecision` 操作符来控制显示浮点数值的有效数的数量。

```

1 // This program demonstrates how the setprecision manipulator
2 // affects the way a floating-point value is displayed.
3 #include <iostream>
4 #include <iomanip> // Header file needed to use setprecision
5 using namespace std;
6
7 int main()
8 {
9     double number1 = 132.364, number2 = 26.91;
10    double quotient = number1 / number2;
11    cout << quotient << endl;
12    cout << setprecision(5) << quotient << endl;
13    cout << setprecision(4) << quotient << endl;
14    cout << setprecision(3) << quotient << endl;
15    cout << setprecision(2) << quotient << endl;
16    cout << setprecision(1) << quotient << endl;
17    return 0;
18 }
```

程序输出结果：

```
4.91877
4.9188
4.919
4.92
4.9
5
```

程序中的第一个值显示在第 11 行，没有设置 `setprecision` 操作符（默认情况下，系统使用 6 个有效数显示浮点值）。后续的 `cout` 语句打印相同的值，但四舍五入为 5、4、3、2 和 1 个有效数。

请注意，与 `setw` 不同的是，`setprecision` 不计算小数点。例如，当使用 `setprecision(5)` 时，输出包含 5 位有效数，但是需要 6 个位置来显示 4.9188。如果一个数字的值可以由少于 `setprecision` 指定的精度位数来表示，则操作符将不起作用。

使用流操作符 `scientific` 和 `fixed` 可以控制浮点数的输出格式。`scientific` 强制浮点数以科学计数法的形式显示。

`setprecision` 函数指明了浮点数应该显示的小数点后的位数，使用此函数，需要头文件。`setprecision` 是一个 parameterized stream manipulator。而 `endl` 是一个 nonparameterized stream manipulator。

`fixed` 指明浮点数的值应该显示为定点数，而不是科学计数法。`fixed` 与 `setprecision(n)` 连用可以控制小数点后的位数。

当使用 `fixed` 和 `setprecision` 时，打印的值为四舍五入到由 `setPrecision` 的参数指示的小数位数，但内存中的值保持不变，例如 87.946 和 67.543 的输出分别为 87.95 和 67.54，也可以使用 `showpoint` 和 `fixed` 实现上述功能，如果没有 `fixed`，后面的零不会打印。例如 7.3000 会打印为 7.3。

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     double x{0.001234567};
6     double y{1.946e9};
7
8     // display x and y in default format
9     cout << "Displayed in default format:\n" << x << '\t' << y;
```

```

10
11     // display x and y in scientific format
12     cout << "\n\nDisplayed in scientific format:\n"
13         <<scientific << x << '\t' << y;
14
15     // display x and y in fixed format
16     cout << "\n\nDisplayed in fixed format:\n"
17         << fixed<< x << '\t' << y << endl;
18 }
```

```

D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
Displayed in default format:
0.00123457      1.946e+09

Displayed in scientific format:
1.234567e-03    1.946000e+09

Displayed in fixed format:
0.001235        1946000000.000000
```

12.7.6 Uppercase/Lowercase Control (uppercase)

uppercase流操作符可以让十六进制或者科学计数法显示数字时，使得原本是小写的字母变为大写。下例为未使用uppercase操作符之前：

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     cout << "Printing uppercase letters in scientific\n"
6         << "notation exponents and hexadecimal values:\n";
7
8 // use std::uppercase to display uppercase letters; use std::hex
9 // and
10    // std::showbase to display hexadecimal value and its base
11    cout << 4.345e10 << "\n"
12        << hex << showbase << 123456789 << endl;
13 }
```

```
D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345e+10
0x75bcd15
```

在使用uppercase操作符后：

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     cout << "Printing uppercase letters in scientific\n"
6         << "notation exponents and hexadecimal values:\n";
7
8 // use std::uppercase to display uppercase letters; use std::hex
9 // and
10 // std::showbase to display hexadecimal value and its base
11 cout << uppercase << 4.345e10 << "\n"
12     << hex << showbase << 123456789 << endl;
13 }
```

```
D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+10
0X75BCD15
```

12.7.7 Specifying Boolean Format (boolalpha)

C++提供了数据类型bool，其值可以是false的，也可以是true的，作为旧式的0表示假，任何非零值表示真的首选替代。布尔变量默认输出为0或1。可以使用流操作符boolalpha让输出流将bool值输出为字符串false或者true。使用noboolalpha使得bool值输出为整数(即，默认设置)。

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     bool booleanvalue{true};
6
7     // display default true booleanvalue
```

```

8  cout << "booleanvalue is " << booleanvalue;
9
10 // display booleanvalue after using boolalpha
11 cout << "\nbooleanvalue (after using boolalpha) is "
12     <<boolalpha<<booleanvalue;
13
14 cout << "\n\nswitch booleanvalue and use noboolalpha\n";
15 booleanvalue = false; // change booleanvalue
16 cout << noboolalpha; // use noboolalpha
17
18 // display default false booleanvalue after using noboolalpha
19 cout << "\nbooleanvalue is " << booleanvalue;
20
21 // display booleanvalue after using boolalpha again
22 cout << "\nbooleanvalue (after using boolalpha) is "
23     <<boolalpha<<booleanvalue<<endl;
24
25 }

```

```

D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
booleanValue is 1
booleanValue (after using boolalpha) is true

switch booleanValue and use noboolalpha

booleanValue is 0
booleanValue (after using boolalpha) is false

```

12.7.8 Setting and Resetting the Format State via Member Function flags

在对要输出流进行了几次操作符的改动后，怎么才能恢复到默认格式呢？无实参的flag成员函数将当前的设置格式作为fmtflags格式返回，此格式代表格式状态。有实参的flags函数，以fmtflags格式为实参，将格式状态设置为fmtflags指明的格式，并返回先前的状态设置。

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){

```

```

5  int integervalue{1000};
6  double doublevalue{0.0947628};

7
8  // display flags value, int and double values (original
9  // format)
10 cout << "The value of the flags variable is: " <<
11 cout.flags()
12     <<"\nPrint int and double in original format:\n"
13     <<integervalue << '\t' << doublevalue;
14
15 // use cout flags function to save original format
16 ios_base::fmtflags originalFormat{cout.flags()};
17 cout << showbase << oct << scientific; // change format
18
19 // display flags value, int and double values (new format)
20 cout << "\n\nThe value of the flags variable is: "
21 <<cout.flags()
22     <<"\nPrint int and double in a new format:\n"
23     <<integervalue << '\t' << doublevalue;
24
25 cout.flags(originalFormat); // restore format
26
27 // display flags value, int and double values (original
28 // format)
29 cout << "\n\nThe restored value of the flags variable is: "
30     <<cout.flags()<<"\nPrint values in original format
31 again:\n"
32     <<integervalue << '\t' << doublevalue << endl;
33 }
```

D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe

The value of the flags variable is: 4098

Print int and double in original format:

1000 0.0947628

The value of the flags variable is: 011500

Print int and double in a new format:

01750 9.476280e-02

The restored value of the flags variable is: 4098

Print values in original format again:

1000 0.0947628

12.8 Stream Error States(流的错误状态)

每个流对象都包含一组状态比特以表示流的状态---sticky format settings,error indicators等。以下述代码为例：

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int integervalue;
6
7     // display results of cin functions
8     cout << "Before a bad input operation:"
9     <<"\ncin.rdstate(): " <<cin.rdstate()
10    <<"\n cin.eof(): " <<cin.eof()
11    <<"\n cin.fail(): " <<cin.fail()
12    <<"\n cin.bad(): " <<cin.bad()
13    <<"\n cin.good(): " <<cin.good()
14    <<"\n\nExpects an integer, but enter a character: ";
15
16    cin >> integervalue; // enter character value
17
18    // display results of cin functions after bad input
19    cout << "\nAfter a bad input operation:"
20    <<"\ncin.rdstate(): " <<cin.rdstate()
21    <<"\n cin.eof(): " <<cin.eof()
22    <<"\n cin.fail(): " <<cin.fail()
23    <<"\n cin.bad(): " <<cin.bad()
24    <<"\n cin.good(): " <<cin.good();
25
26    cin.clear(); // clear stream
27
28    // display results of cin functions after clearing cin
29    cout << "\n\nAfter cin.clear()" << "\ncin.fail(): "
30    <<cin.fail()
31    <<"\ncin.good(): " <<cin.good()<<endl;
```

```
D:\clionproject\test_11_09\cmake-build-debug\test_11_09.exe
Before a bad input operation:
cin.rdstate(): 0
cin.eof(): 0
cin.fail(): 0
cin.bad(): 0
cin.good(): 1

Expects an integer, but enter a character:A

After a bad input operation:
cin.rdstate(): 4
cin.eof(): 0
cin.fail(): 1
cin.bad(): 0
cin.good(): 0

After cin.clear()
cin.fail(): 0
cin.good(): 1
```

rdstate成员函数： 成员函数rdstate返回流中的错误状态。例如，通过调用cout.rdstate返回流的状态，然后通过switch语句检查eofbit、badbit、failbit 和 goodbit来检查这些状态。检测流状态的首选方法是使用成员函数eof, fail, bad 和 good， 使用这些函数不要求了解具体的状态位。

eof成员函数判定是否在流中遇到了end-of-file。在此例中，一开始没有输入任何内容，函数返回0(false)。此函数检查流的eofbit数据成员的值，该值在输入流遇到文件结束符后被设置为真。 Returns **true** if the associated stream has reached end-of-file. Specifically, returns **true** if **eofbit** is set in **rdstate()** .此函数仅报告最近 I/O 操作设置的流状态;它不检查关联的数据源。例如，最近的I/O操作是get(),get()函数返回文件的最后一个字符，eof()返回false。下一个get()函数从输入流中不会提取到任何字符，并且会设置eofbit.只有这样，eof()才会返回true。

```
1 #include <cstdlib>
2 #include <fstream>
3 #include <iostream>
4
5 int main()
```

```

6  {
7      std::ifstream file("test.txt");
8      if (!file) // operator! is used here
9      {
10         std::cout << "File opening failed\n";
11         return EXIT_FAILURE;
12     }
13
14     // typical C++ I/O loop uses the return value of the I/O
15     // function
16     // as the loop controlling condition, operator bool() is used
17     // here
18     for (int n; file >> n;)
19         std::cout << n << ' ';
20     std::cout << '\n';
21
22     if (file.bad())
23         std::cout << "I/O error while reading\n";
24     else if (file.eof())
25         std::cout << "End of file reached successfully\n";
26     else if (file.fail())
27         std::cout << "Non-integer data encountered\n";
28 }
```

fail成员函数判定一个流操作是否失败。此函数检查流的failbit数据成员，Returns **true** if an error has occurred on the associated stream.当在流中发生格式错误时，failbit位将被设置。例如程序要求输入整数，但是在输入流中有非整数的字符的情况。在遇到这种错误时，这些字符不会丢失。成员函数fail将报告流操作失败了，通常这种错误是可以恢复的。

```

1 #include <cstdlib>
2 #include <fstream>
3 #include <iostream>
4
5 int main()
6 {
7     std::ifstream file("test.txt");
8     if (!file) // operator! is used here
9     {
10         std::cout << "File opening failed\n";
11         return EXIT_FAILURE;
12     }
```

```

13
14     // typical C++ I/O loop uses the return value of the I/O
15     // function
16     // as the loop controlling condition, operator bool() is used
17     // here
18     for (int n; file >> n;)
19         std::cout << n << ' ';
20     std::cout << '\n';
21
22     if (file.bad())
23         std::cout << "I/O error while reading\n";
24     else if (file.eof())
25         std::cout << "End of file reached successfully\n";
26     else if (file.fail())
27         std::cout << "Non-integer data encountered\n";
28 }
```

ios_base::iostate flags			basic_ios accessors					
eofbit	failbit	badbit	good()	fail()	bad()	eof()	operator bool	operator!
false	false	false	true	false	false	false	true	false
false	false	true	false	true	true	false	false	true
false	true	false	false	true	false	false	false	true
false	true	true	false	true	true	false	false	true
true	false	false	false	false	false	true	true	false
true	false	true	false	true	true	true	false	true
true	true	false	false	true	false	true	false	true
true	true	true	false	true	true	true	false	true

bad成员函数: 判定流操作是否失败。当发生数据丢失时, 将会设置badbit位。成员函数bad将报告流操作是否失败了。一般情况下, 这种严重的错误是不能修复的。

good成员函数: 如果流中的eofbit、failbit 和 badbit位都没有被设置, 那么goodbit位将被设置, 即如果函数eof, fail 和 bad都返回false值, 则成员函数good返回true值。I/O操作只在“好的”流中才能进行。

clear成员函数: clear成员函数将流的状态重置为“好的”, 使得流可以继续执行I/O操作。clear函数的默认参数是goodbit, 所以语句cin.clear()清空了cin, 并且为该流设置goodbit位。语句cin.clear(ios::failbit)则为流设置failbit位。

如果failbit位 和 badbit位其中至少一个被设置, 则basic_ios的成员函数operator!返回true; operator void*返回false值(0)

重载! 和bool

重载运算符可用于测试流在条件下的状态。operator!成员函数返回true如果badbit和failbit其一为true,或者两者均为true。如果badbit为真, failurebit为真或两者都为真, 则operator bool成员函数返回false。当在选择语句或迭代语句的控制下测试真/假条件时, 这些函数在I/O处理中非常有用。例如下列语句:

```
1 if(!cin){  
2     //process invalid input stream  
3 }
```

如果cin的流由于输入失败而无效, 则执行代码。或者:

```
1 while(cin>>variableName){  
2     //process valid input  
3 }
```

只要每个输入操作都成功, 就可以执行循环, 并在输入失败或遇到end-of-file时终止循环。

13. 文件处理

13.1 引言

内存中数据的存储是暂时的。文件用于数据的持久化- -数据的永久保留。计算机将文件存储在二次存储设备上, 如硬盘、CD、DVD、闪存和磁带。

13.2 文件和流(Files and Streams)

C++将文件简单的看作一系列字节。



Fig. 14.1 | C++'s simple view of a file of n bytes.

每个文件以end-of-file标记或在操作系统维护的管理数据结构中记录的特定字节数结束。当一个文件打开, 就创建了一个对象, 流就和此对象联系起来。流和对象之间的联系为程序和文件提供了交流通道。

为了文件处理, C++的头文件<iostream><fstream>必须包含。

Header <fstream> includes the definitions for the stream class templates

- `basic_ifstream`—a subclass of `basic_istream` for file input
- `basic_ofstream`—a subclass of `basic_ostream` for file output
- `basic_fstream`—a subclass of `basic_iostream` for file input *and* output.

Each has a predefined specialization for `char` I/O. In addition, the <fstream> library provides `typedef` aliases for these template specializations:

- `ifstream` is an alias for `basic_ifstream<char>`
- `ofstream` is an alias for `basic_ofstream<char>`
- `fstream` is an alias for `basic_fstream<char>`.

头文件fstream定义了三个类型来支持文件IO: ifstream从一个给定文件读取数据, ofstream向一个给定的文件写入数据, fstream可以读写给定文件。要在 C++ 中进行文件处理, 必须在 C++ 源代码文件中包含头文件 和。

每当我们想要读写一个文件时, 可以定义一个文件流对象, 并将对象与文件关联起来。

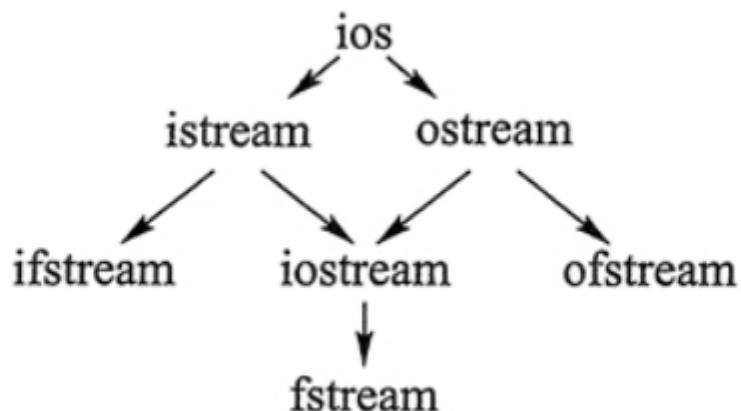


图1: C++类库中的流类

两个重要的输入/输出流

- ◊ 一个iostream对象可以是数据的源或目的。
- ◊ 两个重要的I/O流类都是从iostream派生的, 它们是fstream和 stringstream。这些类继承了前面描述的istream和ostream类的功能。

fstream类

- ◊ `fstream`类支持磁盘文件输入和输出。
- ◊ 如果需要在同一个程序中从一个特定磁盘文件读并写到该磁盘文件，可以构造一个`fstream`对象。
- ◊ 一个`fstream`对象是有两个逻辑子流的单个流，两个子流一个用于输入，另一个用于输出。

stringstream类

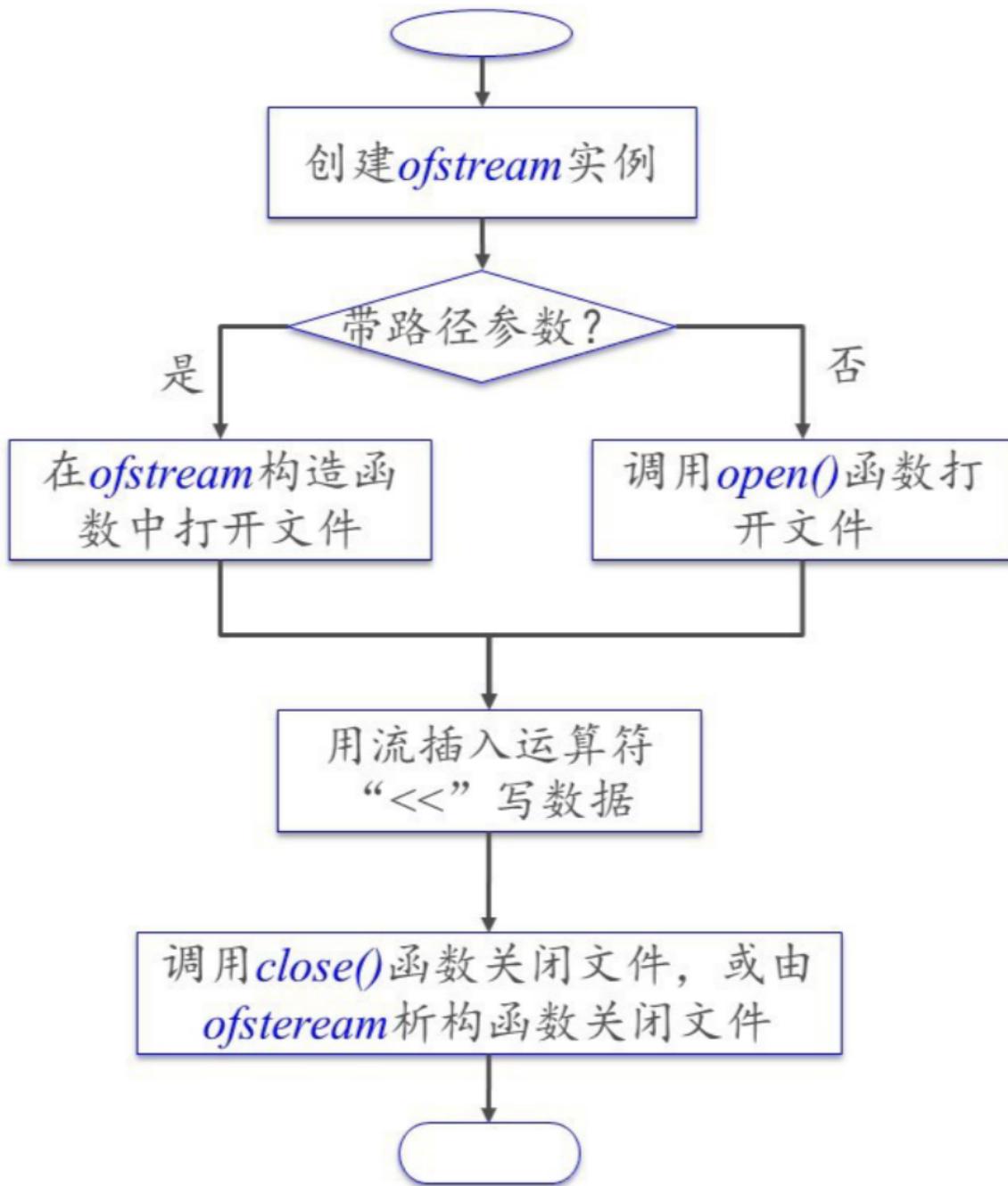
`stringstream`类支持面向字符串的输入和输出

可以用于对同一个字符串的内容交替读写，同样是由两个逻辑子流构成。

13.3 向文件写入数据

`ofstream`类可以向文本文件写入数据。

输出数据的流程：



当文件已经存在的情况下，已有的内容会被清除。

13.4 ofstream类使用的示例

C++不对文件施加任何结构。因此，在C++中不存在类似于记录(可以用几个相关的字段组成一条记录)的概念。必须构造满足应用需求的文件。下面的例子展示了如何在一个文件上施加一个简单的记录结构。

```
1 | #include <iostream>
```

```

2 #include <iostream>
3 #include <fstream>
4 #include <cstdlib>
5 using namespace std;
6
7 int main() {
8     // ofstream constructor opens file
9     ofstream outClientFile{"clients.txt", ios::out};
10
11    // exit program if unable to create file
12    if (!outClientFile) { // overloaded ! operator
13        cerr << "File could not be opened" << endl;
14        exit(EXIT_FAILURE);
15    }
16
17    cout << "Enter the account, name, and balance.\n"
18        << "Enter end-of-file to end input.\n? ";
19
20    int account; // the account number
21    string name; // the account owner's name
22    double balance; // the account balance
23
24    // read account, name and balance from cin, then place in
25    // file
26    while (cin >> account >> name >> balance) {
27        outClientFile << account << ' ' << name << ' ' << balance
28        << endl;
29        cout << "? ";
30    }
31
32 }

```

上述代码，创建一个顺序文件，该文件可能用于应收账款系统，以帮助管理其信贷客户欠公司的钱。

13.4.1 文件打开模式

上述代码将数据写入文件，因此我们通过创建ofstream对象打开文件进行输出。向对象的构造函数传入了两个实参，文件名字和文件打开方式(file-open mode)。对于一个ofstream对象，文件打开模式可以是ios::out(the default)，用以向文件输出数据。或者ios::app,用以在文件末尾增加数据(不会修改文件中原先存在的任何数据)。在上述代码中，创造了ofstream的对象-outClientFile，和文件clients.txt联系起来。由于没有指定

文件的存储路径，最终文件会和程序存放在同一路径。ofstream的构造函数打开文件---这与文件建立了一条通信线路。因为ios::out是默认的，所以：

```
1 | ofstream outClientFile("clients.txt", ios::out);
```

也可以写为：

```
1 | ofstream outClientFile("clients.txt");
```

现存的文件由模式ios::out打开会被截断---即文件中的所有数据均被丢弃。如果文件不存在，则ofstream对象会创造一个文件(文件名即为指定的名字)。

Mode	Description
ios::app	<i>Append</i> all output to the end of the file.
ios::ate	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written <i>anywhere</i> in the file.
ios::in	Open a file for <i>input</i> .
ios::out	Open a file for <i>output</i> .
ios::trunc	<i>Discard</i> the file's contents (this also is the default action for ios::out).
ios::binary	Open a file for binary, i.e., <i>nontext</i> , input or output.

Fig. 14.3 | File-open modes.

上述几种打开模式可以组合在一起，使用位或运算符。

```
1 | stream.open("name.txt", ios::out|ios::app);
```

13.4.2 通过open函数打开文件

也可以创造一个ofstream对象而不需要打开特定的文件---这种情况下，文件可以在后续的操作中附加到对象上。例如，下列的语句：

```
1 | ofstream outClientFile;
```

创造了对象，但并没有和文件联系起来。ofstream的成员函数open打开一个文件，并将其附加在一个现存的ofstream对象上，如下所示：

```
1 | outClientFile.open("clients.txt", ios::out);
```

上述语句中的ios::out也是默认的，即也可以省略。

13.4.3 测试文件是否成功打开

在创建了ofstream对象后并试图打开它，if语句使用了重载的ios的成员函数operator!来判定打开操作是否成功。operator!函数返回true如果failbit或者badbit被流设置---这种情况下，由于打开操作失败，两者其一或者两者均被设置。可能的原因是：

- 1.试图打开一个不存在的文件以读取
- 2.试图打开一个在无权访问的文件夹中的文件以读写。
- 3.为了写入打开文件，但没有可用的硬盘空间。

The argument Passing to exit is returned to the environment from which the program was invoked.EXIT_SUCCESS (defined in <cstdlib>) to exit indicates that the program terminated normally; passing any other value (in this case EXIT_FAILURE) indicates that the program terminated due to an error.

13.4.4 Overloaded bool Operator

上述代码在while语句中，隐式的对cin调用成员函数operator bool。只要failbit和badbit均未被设置，while的条件就会保持true的状态。

13.4.5 Processing Data

如果上述程序成功打开文件，程序就开始处理数据。When end-of-file is encountered or bad data is entered, operator bool returns false and the while statement terminates.

13.4.6 Closing a File

对于上述程序，一旦用户输入end-of-file指示符，main函数终止。此操作隐式的调用了outClientFile的析构函数，此析构函数用来关闭文件。也可以显示的关闭，使用成员函数close：

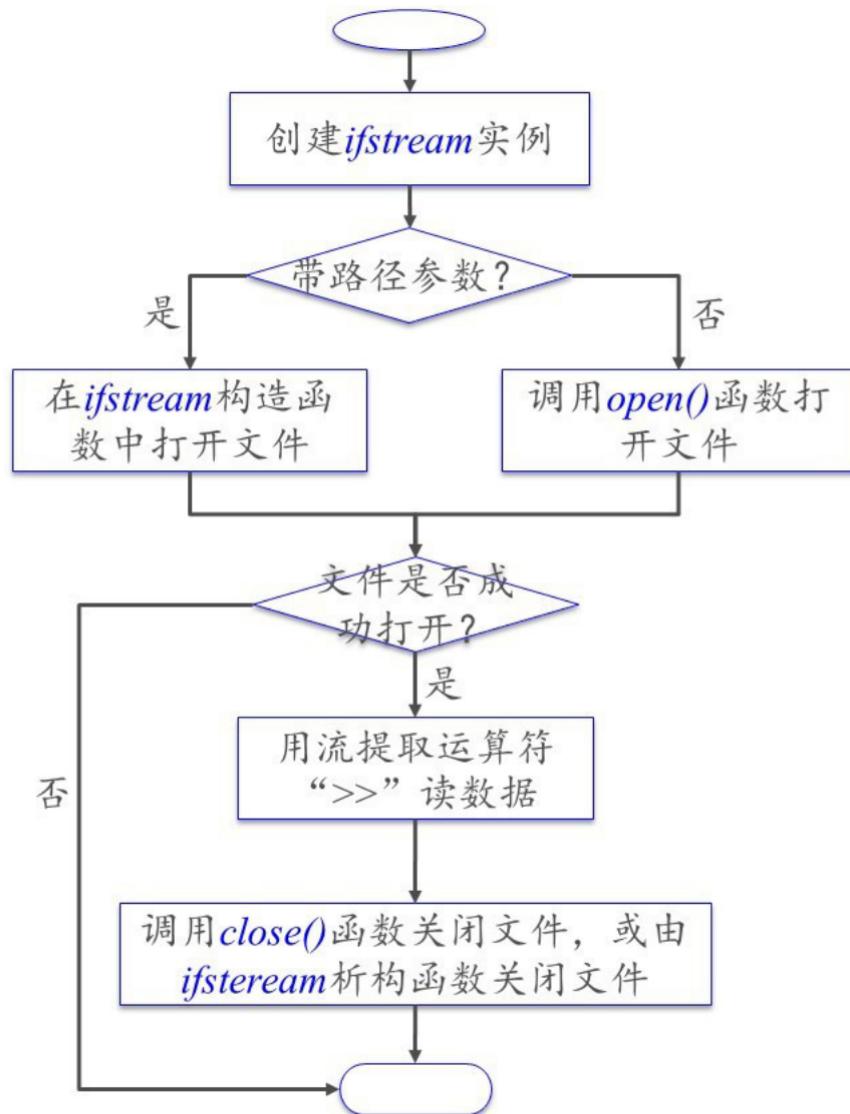
```
1 | outClientFile.close();
```

Always close a file as soon as it's no longer needed in a program.

13.5 从文件中读取数据

13.5.1 基本概念

`ifstream`类可以从文本文件中读取数据。需要检测文件是否成功打开。



若要正确读取数据，必须确切了解数据的存储格式。

To read data correctly, you need to know exactly how data is stored.(若想正确读出数据，必须确切了解数据的存储格式)

For example, the program in previous would not work if the score is a `int` value.

```
// 正确做法:  
iinput >> stringName >> doubleScore;
```

LiLei 90.5
HanMeimei 85

```
// 错误做法  
string name{};  
int score{ 0 };  
input >> name >> score; //score: 90  
cout << name << " " << score << endl;  
input >> name >> score;  
cout << name << " " << score << endl;
```

13.5.2 检测文件是否成功打开:

1.可能出现的错误：

读文件时，文件不存在；写文件时介质只读

2.检测文件是否成功打开的方法

`open()`之后马上调用`fail()`函数，`fail()`函数返回`true`，文件未打开

```
1 ofstream output("hello.txt");  
2 if(output.fail()){  
3     cout<<"(can't open file \"hello.txt\")";  
4 }
```

13.5.3 检测文件是否到达末尾

使用eof()函数检查是否到达文件末尾。

```
1 ifstream in("hello.txt");
2 while(in.eof()==false){
3     cout<<static_cast<char>(in.get());
4 }
```

13.5.3 Opening a File for Input

```
1 #include <iostream>
2 #include <fstream>
3 #include <iomanip>
4 #include <string>
5 #include <cstdlib>
6 using namespace std;
7
8 void outputLine(int, const string&, double);
9
10 int main() {
11     // ifstream constructor opens the file
12     ifstream inClientFile("clients.txt", ios::in);
13
14     // exit program if ifstream could not open file
15     if(!inClientFile){
16         cerr<<"File could not be opened" << endl;
17         exit(EXIT_FAILURE);
18     }
19
20     cout << left << setw(10) << "Account" << setw(13)
21         <<"Name" << "Balance\n" << fixed << showpoint;
22
23     int account;
24     string name;
25     double balance;
26
27     // display each record in file
28     while(inClientFile>>account>>name>>balance){
29         outputLine(account, name, balance);
30     }
31 }
```

```
32
33 // display single record from file
34 void outputLine(int account, const string& name, double balance){
35     cout << left << setw(10) << account << setw(13) << name
36     << setw(7) << setprecision(2) << right << balance << endl;
37 }
```

```
1 ifstream inClientFile("clients.txt");
```

上述语句打开文件。

13.5.3 File-Position Pointers

程序通常从文件的开头按顺序读取，并连续读取所有数据，直到找到所需的数据。在程序执行期间，可能需要（从头开始）按顺序处理文件多次。istream and ostream提供了成员函数seekg和seekp。seekg重新定位文件位置指针（文件中要读取或写入的下一个字节的字节号）。每个 istream 对象都有一个 get 指针，该指针指示文件中下一个输入的字节号，每个 ostream 对象都有一个 put 指针，该指针指示文件中应放置下一个输出的字节号。例如：

```
1 inClientFile.seekg(0);
```

将文件位置指针定位到文件的开始(位置0),并将文件附加给对象。seekg的成员函数的参数是整数。该函数还可以有第二个参数，可以指定第二个参数来指示搜索方向，该参数可以是 ios::beg (默认值) 表示相对于流的开头进行定位，ios::cur 表示相对于流中的当前位置进行定位，或者 ios::end 表示相对于流的末尾向后定位。文件位置指针是一个整数值，用于将文件中的位置指定为距文件起始位置的字节数（也称为与文件开头的偏移量）。定位 file-position 指针的一些示例包括：

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg(n);
// position n bytes in fileObject
fileObject.seekg(n, ios::cur);
// position n bytes back from end of fileObject
fileObject.seekg(n, ios::end);
// position at end of fileObject
fileObject.seekg(0, ios::end);
```

ostream的成员函数seekp也可以执行相同的操作。成员函数tellg和tellp可以分别返回文件位置指针当前的位置。下列的语句将文件位置指针赋给long类型的变量location：

```
1 location = fileObject.tellg();
```

13.6 C++14: Reading and Writing Quoted Text

C++14's new stream manipulator—quoted (header <iomanip>)—enables a program to read quoted text from a stream, including any white space characters in the quoted text, and discards the double quote delimiters.

13.7 Updating Sequential Files

如第 14.3 节所示，格式化并写入顺序文件的数据不能被修改，否则会有破坏文件中其他数据的风险。例如，如果需要将名称“White”更改为“Worthington”，则在不损坏文件的情况下无法覆盖旧名称。White 的记录被写入文件，如下所示：

```
1 | 300 white 0.00
```

如果使用较长的名称从文件中的同一位置开始重写此记录，则该记录将为：

```
1 | 300 Worthington 0.00
```

新记录包含的字符比原始记录多 6 个字符。因此，“Worthington”中“h”之外的字符将覆盖文件中下一个顺序记录的 0.00 和开头。问题在于，在使用流插入运算符 << 和流提取运算符 >> 的格式化输入/输出模型中，字段（以及记录）的大小可能会有所不同。例如，值 7、14、-117、2074 和 27383 都是整数，它们在内部存储相同数量的“原始数据”字节（在 32 位计算机上通常为 4 个字节，在某些 64 位计算机上可能为 8 个字节）。但是，当输出为格式化文本（字符序列）时，这些整数会根据其实际值成为不同大小的字段。因此，格式化的输入/输出模型通常不用于就地更新记录。第 14.7-14.11 节介绍如何使用固定长度的记录执行就地更新。

这样的更新是可以完成的，但很尴尬。例如，要进行前面的名称更改，可以将顺序文件中 300 White 0.00 之前的记录复制到新文件，然后将更新的记录写入新文件，并将 300 White 0.00 之后的记录复制到新文件。然后可以删除旧文件并重命名新文件。这需要处理文件中的每条记录以更新一条记录。但是，如果在一次文件传递中更新了许多记录，则此技术是可以接受的。

13.8 随机访问文件(Random-Access Files)

随机访问意味着可以读写文件的任意位置。

要实现随机访问文件，需要：

1. 我们能知道文件定位器在什么位置
2. 我们能在文件中移动文件定位器

	For reading (读文件时用)	For writing(写文件时用)
获知文件定位器指到哪里	tellg(); tell是获知, g是get表示读文件	tellp(); tell是获知, p是put表示写文件
移动文件定位器到指定位置	seekg(); seek是寻找, g是get表示读文件	seekp(); seek是寻找, p是put表示写文件

可以将数据插入到随机访问文件中，而不会破坏文件中的其他数据。以前存储的数据也可以更新或删除，而无需重写整个文件。在以下各节中，我们将介绍如何创建随机访问文件，将数据输入到文件中，按顺序和随机读取数据，更新数据并删除不再需要的数据。

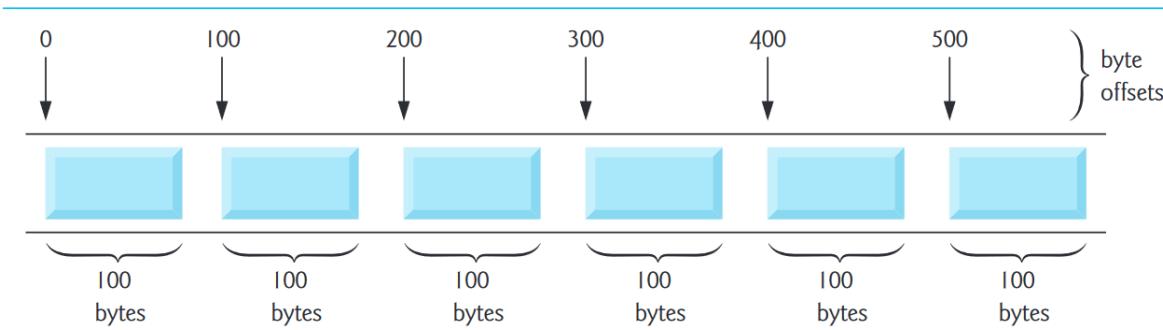


Fig. 14.7 | C++ view of a random-access file.

13.8.1 seek的用法

2.1. seek的原型

```

1 | xxx_stream& seekg/seekp( pos_type pos );
2 | xxx_stream& seekg/seekp( off_type off, std::ios_base::seekdir
  |   dir );

```

seekdir 文件定位方向类型	解释
std::ios_base::beg	流的开始； beg = begin
std::ios_base::end	流的结尾
std::ios_base::cur	流位置指示器的当前位置； cur = current

例子	解释
seekg(42L);	将文件位置指示器移动到文件的第42字节处
seekg(10L, std::ios::beg);	将文件位置指示器移动到从文件开头算起的第10字节处
seekp(-20L, std::ios::end);	将文件位置指示器移动到从文件末尾开始，倒数第20字节处
seekp(-36L, std::ios::cur);	将文件位置指示器移动到从当前位置开始，倒数第36字节处

13.8.2 文件指针

1. File Positioner (文件位置指示器)

1.1. file positioner (fp):

A file consists of a sequence of bytes. (文件由字节序列构成)

File positioner is a special *marker* that is positioned at one of these bytes. (一个特殊标记指向其中一个字节)

1.2. A read or write operation takes place at the location of the file positioner. (读写操作都是从文件位置指示器所标记的位置开始)

When a file is opened, the fp is set at the beginning. (打开文件, fp指向文件头)

When you read or write data to the file, the file pointer moves forward to the next data item. (读写文件时, 文件位置指示器会向后移动到下一个数据项)

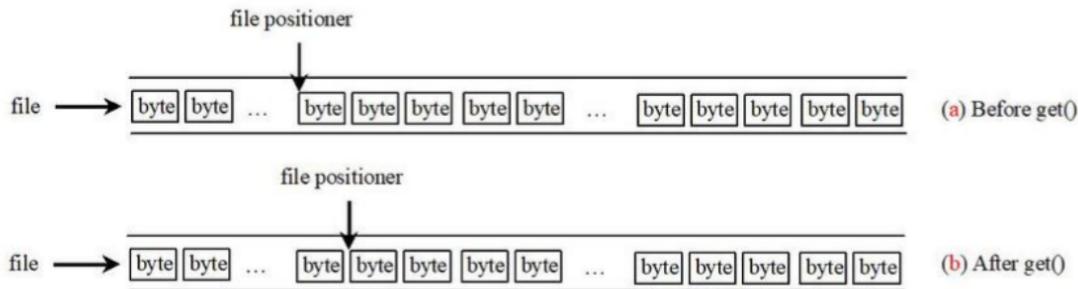
1.3. File Positioner(文件位置指示器)的其它说法

File Pointer(文件指针): 易与C语言的FILE*混淆

File Cursor(文件光标): 借用数据库中的“光标”概念

2.Example of File Positioner

调用aFileStream.get()函数, 会导致 $fp = fp + 1$;



13.9 Creating a Random-Access File

`ostream` 成员函数 `write` 从内存中的特定位置开始, 将固定数量的字节写入到指定的流。当流与文件关联时, 函数写入将数据写入 `put file-position` 指针指定的文件中的位置。`istream` 成员函数 `read` 将输入从指定流读取到内存中从指定地址开始的区域的固定字节数。如果流与文件关联, 则函数在“get”文件位置指针指定的文件中的位置读取输入字节。

13.9.1 Writing Bytes with `ostream` Member Function `write`

将整数写入文件使用下列语句:

```
1 | outFile << number;
```

13.10 向字符串输出

字符串输出流 (`ostringstream`)

- ◊ 用于构造字符串
- ◊ 功能
 - 支持`ofstream`类的除`open`、`close`外的所有操作
 - `str`函数可以返回当前已构造的字符串
- ◊ 典型应用
 - 将数值转换为字符串

13.11 C++17引入的文件系统库

13.11.1 简介

文件系统库放在名字空间`std::filesystem`中。标准库的`filesystem`提供在文件系统与其组件，例如路径、常规文件与目录上进行操作的方法。

文件：持有数据的文件系统对象，能被写入或读取。文件有名称和属性，属性之一是文件类型

路径：标识文件所处位置的一系列元素，可能包含文件名。

路径类：

```
1 namespace fs=std::filesystem;
2 fs::path p{"checkPath.cpp"};
```

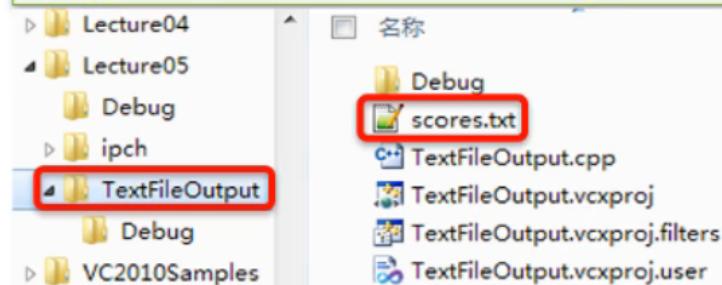
路径的概念：

绝对路径：包含完整的路径和驱动器符号

相对路径：不包含驱动器以及/符号。文件存在于相对于当前路径的位置。

```
std::filesystem::path p("./scores.txt");
```

Current path:
D:\cpp\src\Lecture05\TextFileOutput



OS Type	Absolute path	Directory path
Windows(case insensitive)	c:\example\scores.txt	c:\example
Unix/Linux(case sensitive)	/home/cyd/scores.txt	/home/cyd

Differences between Windows and Linux(两种操作系统的不同)

	Windows	Linux	C++	java
行结束字符	\r\n	\n	-	System.getProperty("line.separator");
路径名分隔符	\	/	std::filesystem::path::preferred_separator	java.io.File.separator
路径名	a:\b\c 或 \\host\b\c	/a/b/c	std::filesystem::path	-

为名字空间起个别名

```
namespace fs = std::filesystem;
```

```
// The directory separator for Windows is a backslash (\), which needs special treat
namespace fs = std::filesystem;
fs::path p1("d:\\cpp\\hi.txt"); // 字符串中的反斜杠要被转义
fs::path p2("d:/cpp/hi.txt"); // Windows也支持正斜杠
fs::path p3(R"(d:\cpp\hi.txt)"); // 使用原始字符串字面量
```

使用std::filesystem::path::prefered_separator,可以得知当前系统使用哪一种路径名分隔符。

13.11.2 路径类及操作

1. Members functions of path class(path类的成员函数)

部分重要的成员函数			说明
连接	+path(string)		构造函数
	+assign(string): path&		为路径对象赋值
修改器	+append(type p): path&		将p追加到路径后。type是string、path或const char*。等价于 /= 运算符；自动添加目录分隔符
	+concat(type p): path&		将p追加到路径后。type是string、path或const char*。等价于+=运算符；不自动添加目录分隔符
分解	+clear(): void		清空存储的路径名
	+remove_filename(): path&		从给定的路径中移除文件名
	+replace_filename(const path& replacement): path&		以 replacement 替换文件名
查询	+root_name(): path		返回通用格式路径的根名
	+root_directory(): path		返回通用格式路径的根目录
	+root_path(): path		返回路径的根路径，等价于 root_name() / root_directory()，即“路径的根名 / 路径的根目录”
	+relative_path(): path		返回相对于 root-path 的路径
	+parent_path(): path		返回到父目录的路径
	+filename(): path		返回路径中包含的文件名
	+stem(): path		返回路径中包含的文件名，不包括文件的扩展名
	+extension(): path		返回路径中包含的文件名的扩展名
查询	+empty(): bool		检查路径是否为空
	+has_xxx(): bool		其中“xxx”是上面“分解”类别中的函数名。这些函数检查路径是否含有相应路径元素

2. Non-member functions (非成员函数)

部分重要的非成员函数			说明
文件类型	operator/(const path& lhs, const path& rhs)		以偏好目录分隔符连接两个路径成分 lhs 和 rhs。比如 path p["C:"]; p = p / "Users" / "batman";
	operator <<, >> (path p)		进行路径 p 上的流输入或输出
查询	s_regular_file(const path& p): bool		检查路径是否是常规文件
	is_directory(const path& p): bool		检查路径是否是目录
修改	is_empty(const path& p): bool		检查给定路径是否指代一个空文件或目录
	current_path(): path		返回当前工作目录的绝对路径 (类似linux指令 pwd)
	current_path(const path& p): void		更改当前路径为p (类似linux指令 cd)
	file_size(const path& p): uintmax_t		对于常规文件 p，返回其大小；尝试确定目录(以及其他非常规文件)的大小的结果是由编译器决定的
	space(const path& p): space_info		返回路径名 p 定位于其上的文件系统信息。space_info中有三个成员：capacity ——文件系统的总大小(字节), free ——文件系统的空闲空间(字节), available ——普通进程可用的空闲空间 (小于或等于 free)
修改	status(const path& p): file_status		返回 p 所标识的文件系统对象的类型与属性。返回的file_status是一个类，其中包含文件的类型(type)和权限(permissions)
	remove(const path& p): bool		删除路径 p 所标识的文件或空目录
	remove_all(const path& p): uintmax_t		递归删除 p 的内容 (若它是目录) 及其子目录的内容，然后删除 p 自身，返回被删文件及目录数量
	rename(const path& old_p, const path& new_p): void		移动或重命名 old_p 所标识的文件系统对象到 new_p(类似linux指令mv)
修改	copy(const path& from, const path& to): void		复制文件与目录。另外一个函数 bool copy_file(from, to) 拷贝单个文件
	create_directory(const path& p): bool		创建目录 p (父目录必须已经存在) ,若 p 已经存在，则函数无操作
	create_directories(const path& p): bool		创建目录 p (父目录不一定存在) ,若 p 已经存在，则函数无操作

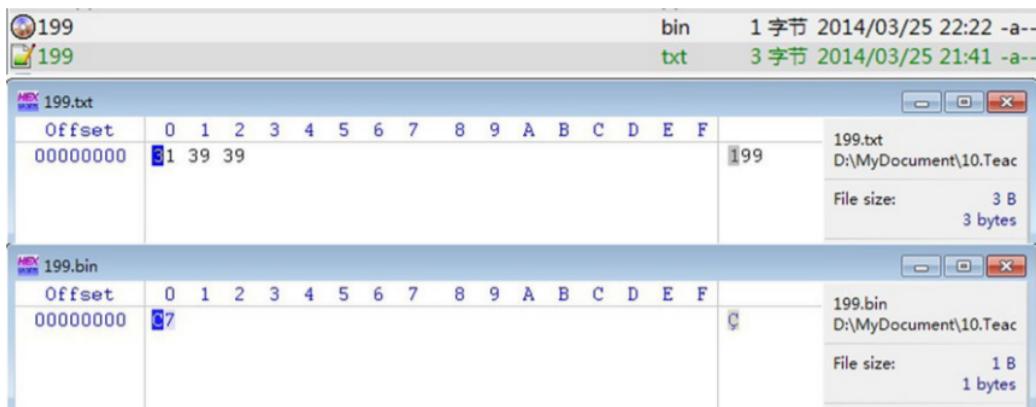
13.12 二进制输入输出

1.1. TEXT file vs BINARY file (not technically precise) (文本文件与二进制文件)

- (1) Both **stores** as a sequence of bits (in binary format) (都按二进制格式存储比特序列)
- (2) text file : **interpreted** as a sequence of **characters** (解释为一系列字符)
- (3) binary file : **interpreted** as a sequence of **bits**. (解释为一系列比特)

1.2. For example, the decimal integer 199 (对于十进制整数199)

- (1) 在文本文件中存为3个字符: '1', '9', '9'; 三个字符的ASCII码占3个字节: 0x31, 0x39, 0x39
- (2) 在二进制文件中存为字节类型的值: C7; 十进制 199 = 十六进制 C7



2.1. 文本读写: Windows文件的换行(CRLF) vs *nix文件的换行(LF)

在Windows上, '\n'输出到文件中会自动编码为'\r' '\n' 两个字符

在*nix上, '\n' 字符输出到文件中不变

2.2. Text I/O is built upon binary I/O to provide a level of abstraction for character encoding and decoding. (文本模式的读写是建立在二进制模式读写的基础上的, 只不过是将二进制信息进行了字符编解码)

3.1. Binary I/O does not require conversions. (二进制读写无需信息转换)

numeric value → write (bin I/O) → file
value in memory → copy (no conversion) → file

3.2. How to perform binary I/O ? (如何进行二进制读写)

By default, a file is opened in text mode. (文件默认以文本模式打开)

open a file using the binary mode ios::binary. (用ios::binary以二进制模式打开文件)

	Text I/O (文本模式)	Binary I/O function:(二进制模式)
读	operator >>; get(); getline();	read();
写	operator <<; put();	write();

13.2.1 如何实现二进制读写

1. The `write` Function (write函数)

1.1. prototype (函数原型)

```
1 | ostream& write( const char* s, std::streamsize count );
```

1.2. 可直接将字符串写入文件

```
1 | fstream fs("GreatWall.dat", ios::binary|ios::trunc);
2 | char s[] = "ShanHaiGuan\nJiYongGuan";
3 | fs.write(s, sizeof(s));
```

1.3. 如何将非字符数据写入文件

- (1) Convert any data into a sequence of bytes (byte stream) (先将数据转换为字节序列, 即字节流)
- (2) Write the sequence of bytes to file with `write()` (再用`write`函数将字节序列写入文件)

2. How to convert any data into byte stream? (如何将信息转换为字节流)

2.1. `reinterpret_cast`

该运算符有两种用途:

- (1) cast the address of a type to another type (将一种类型的地址转为另一种类型的地址)
- (2) cast the address to a number, i.e. integer (将地址转换为数值, 比如转换为整数)

2.2. 语法:

```
1 | reinterpret_cast<dataType>(address)
```

address is the starting address of the data (address是待转换的数据的起始地址)

dataType is the data type you are converting to. (dataType是要转至的目标类型)

对于二进制I/O来说, dataType是 `char*`

2.3. 例子

```
1 long int x {0};  
2 int a[3] {21,42,63};  
3 std::string str{"Hello"};  
4 char* p1 = reinterpret_cast<char*>(&x); // variable address  
5 char* p2 = reinterpret_cast<char*>(a); // array address  
6 char* p3 = reinterpret_cast<char*>(&str); // object address
```

3. The *read* Function (read成员函数)

3.1. prototype (函数原型)

```
1 istream& read ( char* s, std::streamsize count);
```

3.2. 例子

```
1 // 读字符串  
2  
3 fstream bio("Greatwall.dat", ios::in | ios::binary);  
4 char s[10];  
5 bio.read(s, 5);  
6 s[5] = '\0';  
7 cout << s;  
8 bio.close();
```

```
1 // 读其它类型数据 (整数)，需要使用 reinterpret_cast  
2  
3 fstream bio("temp.dat", ios::in | ios::binary);  
4 int value;  
5 bio.read(reinterpret_cast<char*>(&value), sizeof(value));  
6 cout << value;
```

14. 容器和迭代器

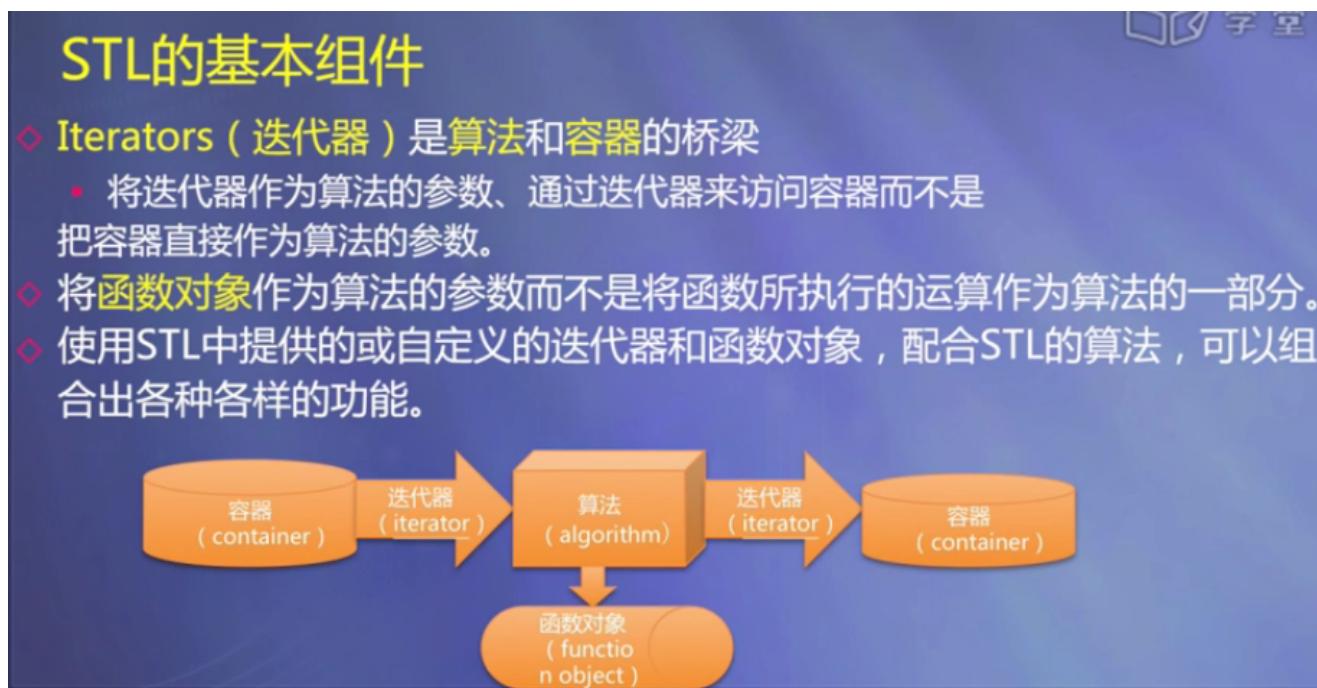
14.1 引言

标准库定义了功能强大的、基于模板的、可重用的组件，这些组件实现了许多用于处理这些数据结构的常见数据结构和算法。这一章介绍的特性经常被称作标准模板库(STL)。

标准库的三个关键组件---容器(模板化的数据结构), 迭代器以及算法。容器是能够存储几乎任何数据类型的对象的数据结构 (有一些限制) 。

迭代器具有类似于指针的属性, 用于操作容器元素。内置数组也可以由标准库算法操作, 使用指针作为迭代器。

标准库算法是执行常见数据操作 (如搜索、排序和比较元素或整个容器) 的函数模板。



避免重新发明轮子;如果可能, 请使用C++标准库的组件进行编程。

14.1.1 泛型程序设计

- 1.编写不依赖于具体数据类型的程序
- 2.将算法从特定的数据结构中抽象出来, 成为通用的
- 3.C++的STL为泛型程序设计奠定了基础

术语：概念

- ◆ 用来界定具备一定功能的数据类型。例如：
 - 将“可以比大小的所有数据类型（有比较运算符）”这一概念记为Comparable
 - 将“具有公有的复制构造函数并可以用‘=’赋值的数据类型”这一概念记为Assignable
 - 将“可以比大小、具有公有的复制构造函数并可以用‘=’赋值的所有数据类型”这个概念记作Sortable
- ◆ 对于两个不同的概念A和B，如果概念A所需求的所有功能也是概念B所需求的功能，那么就说概念B是概念A的子概念。例如：
 - Sortable既是Comparable的子概念，也是Assignable的子概念

术语：模型

- ◆ 模型（model）：符合一个概念的数据类型称为该概念的模型，例如：
 - int型是Comparable概念的模型。
 - 静态数组类型不是Assignable概念的模型（无法用“=”给整个静态数组赋值）

用概念做模板参数名

- ◆ 为概念赋予一个名称，并使用该名称作为模板参数名。
- ◆ 例如
表示insertionSort这样一个函数模板的原型：
`template <class Sortable>
void insertionSort(Sortable a[], int n);`

14.2 容器的介绍

容器被分为四个主要的类别： sequence containers(顺序容器), ordered associative containers(顺序关联容器), unordered associative containers(无序关联容器) and container adapters(容器适配器).

Container class	Description
<i>Sequence containers</i>	
<code>array</code>	Fixed size. Direct access to any element.
<code>deque</code>	Rapid insertions and deletions at front or back. Direct access to any element.
<code>forward_list</code>	Singly linked list, rapid insertion and deletion anywhere. Added in C++11.
<code>list</code>	Doubly linked list, rapid insertion and deletion anywhere.
<code>vector</code>	Rapid insertions and deletions at back. Direct access to any element.
<i>Ordered associative containers—keys are maintained in sorted order</i>	
<code>set</code>	Rapid lookup, no duplicates allowed.
<code>multiset</code>	Rapid lookup, duplicates allowed.
<code>map</code>	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
<code>multimap</code>	One-to-many mapping, duplicates allowed, rapid key-based lookup.
<i>Unordered associative containers</i>	
<code>unordered_set</code>	Rapid lookup, no duplicates allowed.
<code>unordered_multiset</code>	Rapid lookup, duplicates allowed.
<code>unordered_map</code>	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
<code>unordered_multimap</code>	One-to-many mapping, duplicates allowed, rapid key-based lookup.
<i>Container adapters</i>	
<code>stack</code>	Last-in, first-out (LIFO).
<code>queue</code>	First-in, first-out (FIFO).
<code>priority_queue</code>	Highest-priority element is always the first element out.

Fig. 15.1 | Standard Library container classes and container adapters.

顺序容器代表线性数据结构(即，它们的所有元素在概念上都是“排成一排的”)，例如 `arrays`, `vectors` 和链表。关联容器是非线性数据结构，通常可以快速定位存储在容器中的元素。

key-value pairs(键值对)：键-值对(key- value pair)是编程语言对数学概念中映射的实现。键(key)用作元素的索引，值(value)则表示所存储和读取的数据。

顺序容器和关联容器被称为**first-class containers**.堆栈和队列通常是序列容器的约束版本。因此，标准库将类模板stack,queue和priority_queue实现为容器适配器，使程序能够以受约束的方式查看序列容器。

string类支持与顺序容器相同的功能，但是仅支持存储字符数据。

◆ 容器的通用功能

- 用默认构造函数构造空容器
- 支持关系运算符：==、!=、<、<=、>、>=
- begin()、end()：获得容器首、尾迭代器
- clear()：将容器清空
- empty()：判断容器是否为空
- size()：得到容器元素个数
- s1.swap(s2)：将s1和s2两容器内容交换

◆ 相关数据类型 (S表示容器类型)

- S::iterator：指向容器元素的迭代器类型
- S::const_iterator：常迭代器类型

常用容器函数：

大多数容器都提供类似的功能。许多操作适用于所有容器，而其他操作适用于类似容器的子集。

Member function	Description
default constructor	A constructor that <i>initializes an empty container</i> . Normally, each container has several constructors that provide different ways to initialize the container.
copy constructor	A constructor that initializes the container to be a <i>copy</i> of an existing container of the same type.
move constructor	A move constructor (added in C++11 and discussed in Chapter 24) moves the contents of an existing container into a new container of the same type—the old container no longer contains the data. This avoids the overhead of copying each element of the argument container.
destructor	Destructor function for cleanup after a container is no longer needed.
<code>empty</code>	Returns <code>true</code> if there are <i>no</i> elements in the container; otherwise, returns <code>false</code> .
<code>insert</code>	Inserts an item in the container.
<code>size</code>	Returns the number of elements currently in the container.

Member function	Description
<code>copy operator=</code> <code>move operator=</code>	Copies the elements of one container into another.
	The move assignment operator (added in C++11 and discussed in Chapter 24) moves the elements of one container into another container of the same type—the old container no longer contains the data. This <i>avoids the overhead of copying</i> each element of the argument container.
<code>operator<</code>	Returns true if the contents of the first container are <i>less than</i> the second; otherwise, returns <code>false</code> .
<code>operator<=</code>	Returns true if the contents of the first container are <i>less than or equal to</i> the second; otherwise, returns <code>false</code> .
<code>operator></code>	Returns true if the contents of the first container are <i>greater than</i> the second; otherwise, returns <code>false</code> .
<code>operator>=</code>	Returns true if the contents of the first container are <i>greater than or equal to</i> the second; otherwise, returns <code>false</code> .
<code>operator==</code>	Returns true if the contents of the first container are <i>equal to</i> the contents of the second; otherwise, returns <code>false</code> .
<code>operator!=</code>	Returns true if the contents of the first container are <i>not equal to</i> the contents of the second; otherwise, returns <code>false</code> .
<code>swap</code>	Swaps the elements of two containers. As of C++11, there is a non-member-function version of <code>swap</code> that swaps the contents of its two arguments (which must be of the same container type) using <i>move</i> operations rather than <i>copy</i> operations.
<code>max_size</code>	Returns the <i>maximum number of elements</i> for a container.
<code>begin</code>	Overloaded to return either an <code>iterator</code> or a <code>const_iterator</code> that refers to the <i>first element</i> of the container.
<code>end</code>	Overloaded to return either an <code>iterator</code> or a <code>const_iterator</code> that refers to the <i>next position after the end</i> of the container.
<code>cbegin (C++11)</code>	Returns a <code>const_iterator</code> that refers to the container's <i>first element</i> .
<code>cend (C++11)</code>	Returns a <code>const_iterator</code> that refers to the <i>next position after the end</i> of the container.
<code>rbegin</code>	The two versions of this function return either a <code>reverse_iterator</code> or a <code>const_reverse_iterator</code> that refers to the <i>last element</i> of the container.
<code>rend</code>	The two versions of this function return either a <code>reverse_iterator</code> or a <code>const_reverse_iterator</code> that refers to the <i>position before the first element</i> of the container.
<code>crbegin (C++11)</code>	Returns a <code>const_reverse_iterator</code> that refers to the <i>last element</i> of the container.
<code>crend (C++11)</code>	Returns a <code>const_reverse_iterator</code> that refers to the <i>position before the first element</i> of the container.
<code>erase</code>	Removes <i>one or more</i> elements from the container.
<code>clear</code>	Removes <i>all</i> elements from the container.

Overloaded operators `<`, `<=`, `>`, `>=`, `==` and `!=` are not provided for priority_queues. Overloaded operators `<`, `<=`, `>` and `>=` are not provided for the unordered associative containers. Member functions `rbegin`, `rend`, `crbegin` and `crend` are not available in a `forward_list`.

First-Class Container Common Nested Types

下图显示了常见的first-class container嵌套类型（在每个容器类定义中定义的类型）。

typedef	Description
<code>allocator_type</code>	The type of the object used to allocate the container's memory—not included in class template <code>array</code> .
<code>value_type</code>	The type of element stored in the container.
<code>reference</code>	A reference for the container's element type.
<code>const_reference</code>	A reference for the container's element type that can be used only to <i>read</i> elements in the container and to perform <code>const</code> operations.
<code>pointer</code>	A pointer for the container's element type.
<code>const_pointer</code>	A pointer for the container's element type that can be used only to <i>read</i> elements and to perform <code>const</code> operations.
<code>iterator</code>	An iterator that points to an element of the container's element type.
<code>const_iterator</code>	An iterator that points to an element of the container's element type. Used only to <i>read</i> elements and to perform <code>const</code> operations.
<code>reverse_iterator</code>	A reverse iterator that points to an element of the container's element type. Iterates through a container back-to-front.
<code>const_reverse_iterator</code>	A reverse iterator that points to an element of the container's element type and can be used only to <i>read</i> elements and to perform <code>const</code> operations. Used to iterate through a container in reverse.
<code>difference_type</code>	The type of the result of subtracting two iterators that refer to the same container (the overloaded <code>-</code> operator is not defined for iterators of <code>lists</code> and associative containers).
<code>size_type</code>	The type used to count items in a container and index through a sequence container (cannot index through a <code>list</code>).

容器元素的要求

在使用标准库容器之前，请务必确保容器中存储的对象类型支持一组最少的功能。将对象插入容器时，将创建该对象的副本。对象类型应提供复制构造函数和复制赋值运算符（自定义版本或默认版本，具体取决于类是否使用动态内存）。此外，有序关联容器和许多算法需要比较元素，因此，对象类型应提供小于（`<`）和相等（`==`）运算符。

14.3 迭代器介绍

迭代器是专门用来遍历容器的。迭代器是专门用来遍历容器的对象(是容器类的内部类型)

假设v是一个容器，比如 `vector<int> v;`

v的迭代器的类型为：`vector<int>::iterator`

所以，定义一个迭代器变量的写法为：

```
1 | vector<int>::iterator itr;
```

迭代器与指针有许多相似之处，用于指向first-class container的元素以及用于其他目的。迭代器保存对它们所操作的特定容器敏感的状态信息;因此，为每种类型的容器实现了迭代器。某些迭代器操作在容器之间是统一的。例如，取消引用运算符 `(*)` 取消引用迭代器，以便您可以使用它指向的元素。迭代器上的 `++` 操作将其移动到容器的下一个元素（就像将指针递增到内置数组中会将指针指向下一个数组元素一样）。

STL的基本组件——迭代器 (iterator)

- 提供了顺序访问容器中每个元素的方法；
- 可以使用 `“++”` 运算符来获得指向下一个元素的迭代器；
- 可以使用 `“*”` 运算符访问迭代器所指向的元素，如果元素类型是类或结构体，还可以使用 `“->”` 运算符直接访问该元素的一个成员；
- 有些迭代器还支持通过 `“--”` 运算符获得指向下一个元素的迭代器；
- 迭代器是泛化的指针：指针也具有同样的特性，因此指针本身就是一种迭代器；
- 使用独立于STL容器的迭代器，需要包含头文件`<iterator>`。

First-class containers提供成员函数`begin`和`end`。`begin`函数返回一个指向容器第一个元素的迭代器。函数 `end` 返回一个迭代器，该迭代器指向容器末尾之后的第一个元素（一个末尾），这是一个不存在的元素，经常用于确定何时到达容器末尾。

容器`iterator`类型的对象是指可以修改的容器元素。容器`const_iterator`类型的对象是指无法修改的容器元素。

输入流迭代器和输出流迭代器

◇ 输入流迭代器

`istream_iterator<T>`

- 以输入流（如`cin`）为参数构造
- 可用`*(p++)`获得下一个输入的元素

◇ 输出流迭代器

`ostream_iterator<T>`

- 构造时需要提供输出流（如`cout`）
- 可用`(*p++) = x`将`x`输出到输出流

◇ 二者都属于适配器

- 适配器是用来为已有对象提供新的接口的对象
- 输入流适配器和输出流适配器为流对象提供了迭代器的接口

Using `istream_iterator` for Input and `ostream_iterator` for Output

我们使用带有`sequences`（也称为`ranges`）的迭代器。

`istream_iterator`/`ostream_iterator`

```

1 #include <iostream>
2 #include <iterator>
3 using namespace std;
4
5 int main() {
6     cout<<"Enter two integers: ";
7
8     // create istream_iterator for reading int values from cin
9     istream_iterator<int> inputInt{cin};
10
11    int number1{*inputInt};
12    ++inputInt;
13    int number2{*inputInt};
14
15    // create ostream_iterator for writing int values to cout
16    ostream_iterator<int> outputInt{cout};
17
18    cout<<"The sum is: ";
19    *outputInt = number1 + number2;//output result to cout
20    cout<<endl;
21 }
```

上述代码创建一个能够从标准输入对象 `cin` 中提取（输入）`int` 值的`istream_iterator`。

上述代码表示能够在标准输出对象`cout` 中插入（输出）`int` 值的`ostream_iterator`。



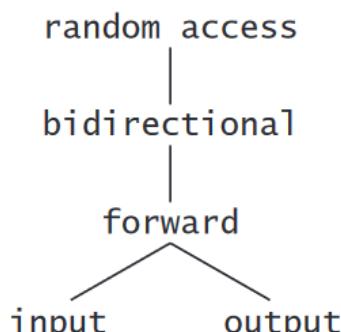
Error-Prevention Tip 15.2

The `` (dereferencing) operator when applied to a `const` iterator returns a reference to `const` for the container element, disallowing the use of non-`const` member functions.*

Iterator Categories and Iterator Category Hierarchy

Category	Description
<i>input</i>	Used to read an element from a container. An input iterator can move only in the <i>forward</i> direction (i.e., from the beginning of the container to the end) one element at a time. Input iterators support <i>only</i> one-pass algorithms—the same input iterator <i>cannot</i> be used to pass through a sequence twice.
Category	Description
<i>output</i>	Used to write an element to a container. An output iterator can move only in the <i>forward</i> direction one element at a time. Output iterators support <i>only</i> one-pass algorithms—the same output iterator <i>cannot</i> be used to pass through a sequence twice.
<i>forward</i>	Combines the capabilities of <i>input</i> and <i>output</i> iterators and retains their position in the container (as state information). Such iterators can be used to pass through a sequence more than once (for so-called multipass algorithms).
<i>bidirectional</i>	Combines the capabilities of a <i>forward iterator</i> with the ability to move in the <i>backward</i> direction (i.e., from the end of the container toward the beginning). Bidirectional iterators support multipass algorithms, such as reversing the elements of a container.
<i>random access</i>	Combines the capabilities of a <i>bidirectional iterator</i> with the ability to <i>directly</i> access <i>any</i> element of the container, i.e., to jump forward or backward by an arbitrary number of elements. These can also be compared with relational operators.

As you follow the hierarchy from bottom to top, each iterator category supports all the functionality of the categories below it in the figure. Thus the “weakest” iterator types are at the bottom and the most powerful one is at the top. Note that this is not an inheritance hierarchy.



Container Support for Iterators(容器对迭代器的支持)

每个容器支持的迭代器类别决定了该容器是否可以与特定算法一起使用。支持随机访问迭代器的容器可以与所有标准库算法一起使用，但如果算法需要更改容器的大小，则该算法不能用于内置数组或数组对象。大多数算法都可以使用指向内置数组的指针来代替迭代器。下图显示了每个容器的迭代器类别。first-class containers、字符串和内置数组都可以使用迭代器进行遍历。

Container	Iterator type	Container	Iterator type
<i>Sequence containers (first class)</i>			<i>Unordered associative containers (first class)</i>
<code>vector</code>	random access	<code>unordered_set</code>	bidirectional
<code>array</code>	random access	<code>unordered_multiset</code>	bidirectional
<code>deque</code>	random access	<code>unordered_map</code>	bidirectional
<code>list</code>	bidirectional	<code>unordered_multimap</code>	bidirectional
<code>forward_list</code>	forward		
<i>Ordered associative containers (first class)</i>			<i>Container adapters</i>
<code>set</code>	bidirectional	<code>stack</code>	none
<code>multiset</code>	bidirectional	<code>queue</code>	none
<code>map</code>	bidirectional	<code>priority_queue</code>	none
<code>multimap</code>	bidirectional		

Predefined Iterator typedefs

并非每个 `typedef` 都是为每个容器定义的。

Predefined typedefs for iterator types	Direction of ++	Capability
<code>iterator</code>	forward	read/write
<code>const_iterator</code>	forward	read
<code>reverse_iterator</code>	backward	read/write
<code>const_reverse_iterator</code>	backward	read

Iterator Operations

下图显示了每个迭代器类型可以执行的操作。

Iterator operation	Description
<i>All iterators</i>	
<code>++p</code>	Preincrement an iterator.
<code>p++</code>	Postincrement an iterator.
<code>p = p1</code>	Assign one iterator to another.
<i>Input iterators</i>	
<code>*p</code>	Dereference an iterator as an <i>rvalue</i> .
<code>p->m</code>	Use the iterator to read the element <code>m</code> .
<code>p == p1</code>	Compare iterators for equality.
<code>p != p1</code>	Compare iterators for inequality.
<i>Output iterators</i>	
<code>*p</code>	Dereference an iterator as an <i>lvalue</i> .
<code>p = p1</code>	Assign one iterator to another.
<i>Forward iterators</i>	Forward iterators provide all the functionality of both input iterators and output iterators.
<i>Bidirectional iterators</i>	
<code>--p</code>	Predecrement an iterator.
<code>p--</code>	Postdecrement an iterator.
<i>Random-access iterators</i>	
<code>p += i</code>	Increment the iterator <code>p</code> by <code>i</code> positions.
<code>p -= i</code>	Decrement the iterator <code>p</code> by <code>i</code> positions.
<code>p + i or i + p</code>	Expression value is an iterator positioned at <code>p</code> incremented by <code>i</code> positions.
<code>p - i</code>	Expression value is an iterator positioned at <code>p</code> decremented by <code>i</code> positions.
<code>p - p1</code>	Expression value is an integer representing the distance between two elements in the same container.
<code>p[i]</code>	Return a reference to the element offset from <code>p</code> by <code>i</code> positions
<code>p < p1</code>	Return <code>true</code> if iterator <code>p</code> is <i>less than</i> iterator <code>p1</code> (i.e., iterator <code>p</code> is <i>before</i> iterator <code>p1</code> in the container); otherwise, return <code>false</code> .
<code>p <= p1</code>	Return <code>true</code> if iterator <code>p</code> is <i>less than or equal to</i> iterator <code>p1</code> (i.e., iterator <code>p</code> is <i>before</i> iterator <code>p1</code> or <i>at the same location</i> as iterator <code>p1</code> in the container); otherwise, return <code>false</code> .
<code>p > p1</code>	Return <code>true</code> if iterator <code>p</code> is <i>greater than</i> iterator <code>p1</code> (i.e., iterator <code>p</code> is <i>after</i> iterator <code>p1</code> in the container); otherwise, return <code>false</code> .
<code>p >= p1</code>	Return <code>true</code> if iterator <code>p</code> is <i>greater than or equal to</i> iterator <code>p1</code> (i.e., iterator <code>p</code> is <i>after</i> iterator <code>p1</code> or <i>at the same location</i> as iterator <code>p1</code> in the container); otherwise, return <code>false</code> .

14.4 算法的介绍

标准库提供了大量算法，您将经常使用这些算法来操作各种容器。插入、删除、搜索、排序等适用于部分或全部顺序和关联容器。这些算法仅通过迭代器间接对容器元素进行操作。

14.5 顺序容器

C++提供五种顺序容器，array,vector,deque,list和forward_list。类模板array,vector和deque通常基于内置数组。类模板list和forward_list实现链表数据结构。

Member function	Description
default constructor	A constructor that <i>initializes an empty container</i> . Normally, each container has several constructors that provide different ways to initialize the container.
copy constructor	A constructor that initializes the container to be a <i>copy</i> of an existing container of the same type.
move constructor	A move constructor (added in C++11 and discussed in Chapter 24) moves the contents of an existing container into a new container of the same type—the old container no longer contains the data. This avoids the overhead of copying each element of the argument container.
destructor	Destructor function for cleanup after a container is no longer needed.
<code>empty</code>	Returns <code>true</code> if there are <i>no</i> elements in the container; otherwise, returns <code>false</code> .
<code>insert</code>	Inserts an item in the container.
<code>size</code>	Returns the number of elements currently in the container.

Member function	Description
copy operator=	Copies the elements of one container into another.
move operator=	The move assignment operator (added in C++11 and discussed in Chapter 24) moves the elements of one container into another container of the same type—the old container no longer contains the data. This <i>avoids the overhead of copying</i> each element of the argument container.
operator<	Returns true if the contents of the first container are <i>less than</i> the second; otherwise, returns false.
operator<=	Returns true if the contents of the first container are <i>less than or equal to</i> the second; otherwise, returns false.
operator>	Returns true if the contents of the first container are <i>greater than</i> the second; otherwise, returns false.
operator>=	Returns true if the contents of the first container are <i>greater than or equal to</i> the second; otherwise, returns false.
operator==	Returns true if the contents of the first container are <i>equal to</i> the contents of the second; otherwise, returns false.
operator!=	Returns true if the contents of the first container are <i>not equal to</i> the contents of the second; otherwise, returns false.
swap	Swaps the elements of two containers. As of C++11, there is a non-member-function version of swap that swaps the contents of its two arguments (which must be of the same container type) using <i>move</i> operations rather than <i>copy</i> operations.
max_size	Returns the <i>maximum number of elements</i> for a container.
begin	Overloaded to return either an <i>iterator</i> or a <i>const_iterator</i> that refers to the <i>first element</i> of the container.
end	Overloaded to return either an <i>iterator</i> or a <i>const_iterator</i> that refers to the <i>next position after the end</i> of the container.
cbegin (C++11)	Returns a <i>const_iterator</i> that refers to the container's <i>first element</i> .
cend (C++11)	Returns a <i>const_iterator</i> that refers to the <i>next position after the end</i> of the container.
rbegin	The two versions of this function return either a <i>reverse_iterator</i> or a <i>const_reverse_iterator</i> that refers to the <i>last element</i> of the container.
rend	The two versions of this function return either a <i>reverse_iterator</i> or a <i>const_reverse_iterator</i> that refers to the <i>position before the first element</i> of the container.
crbegin (C++11)	Returns a <i>const_reverse_iterator</i> that refers to the <i>last element</i> of the container.
crend (C++11)	Returns a <i>const_reverse_iterator</i> that refers to the <i>position before the first element</i> of the container.
erase	Removes <i>one or more</i> elements from the container.
clear	Removes <i>all</i> elements from the container.

通常更倾向于重用标准库容器，而不是开发自定义模板化数据结构。对于大多数应用程序来说，Vector 通常是令人满意的。

需要在容器两端频繁插入和删除的应用程序通常使用 `deque` 而不是 `vector`。虽然我们可以在 `vector` 和 `deque` 的前面和后面插入和删除元素，但类 `deque` 在前面进行插入和删除比 `vector` 更有效。

在容器的中间和/或极端处频繁插入和删除的应用程序通常使用 `list`，因为它可以有效地实现在数据结构中的任何位置插入和删除。

顺序容器的比较

- ◊ STL所提供的顺序容器各有所长也各有所短，我们在编写程序时应当根据我们对容器所需要执行的操作来决定选择哪一种容器。
- ◊ 如果需要执行大量的随机访问操作，而且当扩展容器时只需要向容器尾部加入新的元素，就应当选择向量容器 `vector`；
- ◊ 如果需要少量的随机访问操作，需要在容器两端插入或删除元素，则应当选择双端队列容器 `deque`；
- ◊ 如果不需要对容器进行随机访问，但是需要在中间位置插入或者删除元素，就应当选择列表容器 `list` 或 `forward_list`；
- ◊ 如果需要数组，`array` 相对于内置数组类型而言，是一种更安全、更容易使用的数组类型。

14.5.1 顺序容器-`vector`

我们在之前介绍的类模板 `vector` 提供了一个具有连续内存位置的动态数据结构。这样就可以通过下标运算符 `[]` 高效、直接地访问向量的任何元素，就像使用内置数组一样。当容器中的数据必须通过下标轻松访问或将要排序时，以及元素的数量可能需要增加时，最常使用类模板 `vector`。当向量的内存耗尽时，向量会分配一个更大的内置数组，复制（或移动）将原始元素放入新的内置数组中，并解除分配旧的内置数组。

向量 (`Vector`)

- ◊ 特点
 - 一个可以扩展的动态数组
 - 随机访问、在尾部插入或删除元素快
 - 在中间或头部插入或删除元素慢
- ◊ 向量的容量
 - 容量(`capacity`)：实际分配空间的大小
 - `s.capacity()`：返回当前容量
 - `s.reserve(n)`：若容量小于 `n`，则对 `s` 进行扩展，使其容量至少为 `n`

```
2 #include <vector> // vector class-template definition
3 using namespace std;
4
5 // prototype for function template printvector
6 template <typename T> void printvector(const vector<T>&
7 integers2);
8
9 int main() {
10     vector<int> integers;
11
12     cout<<"The initial size of integers is: "<<integers.size()
13         <<"\nThe initial capacity of integers is: "
14     <<integers.capacity();
15
16     //function push_back is in vector, deque and list
17     integers.push_back(2);
18     integers.push_back(3);
19     integers.push_back(4);
20
21     cout<<"\nThe size of integers is: "<<integers.size()
22         <<"\nThe capacity of integers is: "<<integers.capacity();
23     cout<<"\n\nOutput built-in array using pointer notation: ";
24     const size_t SIZE{6};
25     int values[SIZE]{1,2,3,4,5,6}; // initialize values
26
27     // display array using pointer notation
28     for(const int* ptr = cbegin(values); ptr!=
29         cend(values); ++ptr){
30         cout<<*ptr<< ' ';
31     }
32
33     cout<<"\nOutput vector using iterator notation: ";
34     printvector(integers);
35     cout << "\nReversed contents of vector integers: ";
36
37     // display vector in reverse order using
38     const_reverse_iterator
```

```

39     cout<<endl;
40 }
41
42 // function template for outputting vector elements
43 template <typename T> void printVector(const vector<T>&
44     integers2{
45     // display vector elements using const_iterator
46     for(auto constIterator=integers2.cbegin();constIterator != integers2.cend();++constIterator){
47         cout << *constIterator << ' ';
48     }

```

```

D:\clionproject\test_11_15\cmake-build-debug\test_11_15.exe
The initial size of integers is: 0
The initial capacity of integers is: 0
The size of integers is: 3
The capacity of integers is: 4

Output built-in array using pointer notation: 1 2 3 4 5 6
Output vector using iterator notation: 2 3 4
Reversed contents of vector integers: 4 3 2

```

进程已结束，退出代码为 0

顺序容器的接口（不包含单向链表（forward_list）和数组（array））

- ◇ 构造函数
- ◇ 赋值函数
 - assign
- ◇ 插入函数
 - insert , push_front (只对list和deque) , push_back , emplace , emplace_front
- ◇ 删除函数
 - erase , clear , pop_front (只对list和deque) , pop_back , emplace_back
- ◇ 首尾元素的直接访问
 - front , back
- ◇ 改变大小
 - resize

创建vector:

下列语句生成了一个类模板vector的名为integers的对象，用以存储int类型的值。vector的默认构造函数创造一个空的vector，没有存储任何元素(即，vector的size为0)，并且没有存储元素的空间(即，它的capacity为0)。所以向vector中添加元素时，需要分配内存。

```
1 | vector<int> integers;
```

vector的成员函数size和capacity:

size函数可以在处除了forward_list之外的所有容器中使用，此函数返回容器中当前存储的元素个数。capacity函数(只在vector和deque中使用)，返回在向量需要动态调整自身大小以容纳更多元素之前可以存储在向量中的元素数。

vector的成员函数push_back:

push_back函数(在array和forward_list之外的顺序容器中可以使用)将元素附加到vector中。如果vector的容量已满(即其capacity等于其size)，则vector会增加其大小 - 某些实现将vector的容量增加一倍。array和vector以外的顺序容器也提供push_front函数。



Performance Tip 15.7

It can be wasteful to double a vector's size when more space is needed. For example, a full vector of 1,000,000 elements resizes to accommodate 2,000,000 elements when one new element is added. This leaves 999,999 unused elements. You can use resize and reserve to control space usage better.

在修改vector后更新size和capacity:

当我们向vector中添加一个元素时，vector会为这一个元素分配空间，size函数会返回1，以说明当前vector中存储了一个元素。

当我们向vector中添加第二个元素时，vector的capacity会翻倍，即，变为2。size函数的返回值也会变成2。

当我们向vector中添加第三个元素时，vector的capacity会翻倍，即，变为4。size函数的返回值为3。

当vector中被元素填满以后，再试图添加元素，vector的capacity会变成8。

使用指针输出内置数组的内容：

指向内置数组的指针可用作迭代器。使用 C++14 的全局 `cbegin` 和 `cend` 函数，它们的工作方式与函数 `begin` 和 `end` 的方式相同，但返回不能用于修改数据的 `const` 迭代器。C++14 提供了 `rbegin`, `rend`, `crbegin` 和 `crend` 函数用以遍历整个内置数组或者容器。`rbegin` 和 `rend` 返回的迭代器可以用来修改数据，`crbegin` 和 `crend` 返回的 `const` 迭代器不可以修改数据。

使用迭代器输出 `vector` 的内容：

上述代码调用函数 `printVector` 使用迭代器输出 `vector` 中的内容。函数接收 `const vector` 的引用。`vector` 的成员函数 `cbegin` 返回指向 `vector` 首元素的 `const_iterator`。`vector` 的成员函数 `cend` 返回指向 `vector` 最后一个元素下一个位置的 `const_iterator`。

函数 `cbegin`、`begin`、`cend` 和 `end` 可用于所有 first-class containers。

请记住，迭代器的作用类似于指向元素的指针，并且运算符 `*` 被重载以返回对元素的引用。

```
1 // function template for outputting vector elements
2 template <typename T> void printVector(const vector<T>& integers2)
3 {
4     // display vector elements using const_iterator
5     for(auto constIterator=integers2.cbegin();constIterator != integers2.cend();++constIterator){
6         cout << *constIterator << ' ';
7     }
}
```

上述代码中的 `for` 循环可以使用基于范围的 `for` 循环替换：

```
1 for (auto const& item : integers2) {
2     cout << item << ' ';
3 }
```

试图解引用一个位于容器外的迭代器会导致错误---即由 `end` 或 `cend` 函数返回的迭代器不能解引用以及进行递增操作。

使用 `const_reverse_iterator` 反向输出 `vector` 中的元素：

C++11 添加了 `vector` 成员函数 `crbegin` 和 `crend`，它们在反向遍历容器时返回表示起点和终点的 `const_reverse_iterator`。大部分 first-class containers 支持这种类型的迭代器。`vector` 类还提供了成员函数 `rbegin` `rend` 以获得 non-const `reverse_iterator`。

C++11:shrink_to_fit:

在C++11中，可以使vector和deque调用成员函数shrink_to_fit以返回系统不需要的内存。这要求容器将其容量减少到容器中的元素数。根据 C++ 标准，实现可以忽略此请求，以便它们可以执行特定于实现的优化。

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 #include <iterator>
5 #include <stdexcept>
6 using namespace std;
7
8 int main(){
9     vector<int> values{1, 2, 3, 4, 5, 6};
10    vector<int> integers{values.cbegin(), values.cend()};
11    ostream_iterator<int> output{cout, " "};
12
13    cout << "Vector integers contains: ";
14    copy(integers.cbegin(), integers.cend(), output);
15
16    cout << "\nFirst element of integers: "<<integers.front()
17        <<"\nLast element of integers: "<<integers.back();
18
19    integers[0] = 7;//set first element to 7
20    integers.at(2) = 10;// set element at position 2 to 10
21
22    // insert 22 as 2nd element
23    integers.insert(integers.cbegin() + 1, 22);
24
25    cout << "\n\nContents of vector integers after changes: ";
26    copy(integers.cbegin(), integers.cend(), output);
27
28    // access out-of-range element
29    try{
30        integers.at(100) = 777;
31    }
32    catch(out_of_range &outOfRange){// out_of_range exception
33        cout << "\n\nException: " << outOfRange.what();
34    }
35
36    integers.erase(integers.cbegin());// erase first element
```

```

37     cout << "\n\nvector integers after erasing first element: ";
38     copy(integers.cbegin(), integers.cend(), output);
39
40     // erase remaining elements
41     integers.erase(integers.cbegin(), integers.cend());
42     cout << "\nAfter erasing all elements, vector integers "
43         <<(integers.empty() ? "is" : "is not")<< " empty";
44
45     // insert elements from the vector values
46     integers.insert(integers.cbegin(), values.cbegin(),
47                     values.cend());
47     cout << "\n\nContents of vector integers before clear: ";
48     copy(integers.cbegin(), integers.cend(), output);
49
50     // empty integers; clear calls erase to empty a collection
51     integers.clear();
52     cout << "\nAfter clear, vector integers "
53         <<(integers.empty()?"is":"is not")<< " empty" << endl;
54 }

```

```

D:\clionproject\untitled\cmake-build-debug\untitled.exe
Vector integers contains: 1 2 3 4 5 6
First element of integers: 1
Last element of integers: 6

Contents of vector integers after changes: 7 22 2 10 4 5 6

Exception: vector::_M_range_check: __n (which is 100) >= this->size() (which is 7)

Vector integers after erasing first element: 22 2 10 4 5 6
After erasing all elements, vector integers is empty

Contents of vector integers before clear: 1 2 3 4 5 6
After clear, vector integers is empty

进程已结束, 退出代码为 0

```

```
1 |     ostream_iterator<int> output{cout, " "};
```

上述代码定义了一个名为output的ostream_iterator，可以通过cout输出以单个空格分隔的整数。一个ostream_iterator只能输出int类型的或者与int类型兼容的类型的整数。构造函数的第一个参数指定了输出流，第二个参数指定了用来分隔数字的字符。

copy算法：

```
1 | copy(integers.cbegin(), integers.cend(), output);
```

上述代码使用了copy算法，来自头文件。copy算法将integers的全部内容输出到标准输出。

vector的成员函数front和back：

```
1 | cout << "\nFirst element of integers: "<<integers.front()
2 | <<"\nLast element of integers: "<<integers.back();
```

上述代码使用的函数front和back分别决定vector的第一个和最后一个元素。front返回vector中第一个元素的引用，begin函数返回指向vector第一个元素的random-access iterator。back函数返回vector最后一个元素的引用，而end函数返回指向vector最后一个元素下一个位置的random-access iterator。

The results of front and back are undefined when called on an empty vector.

访问vector的元素：

```
1 | integers[0] = 7;//set first element to 7
2 | integers.at(2) = 10;// set element at position 2 to 10
```

上述访问vector元素的方式也可以用在deque容器中。第一行代码使用重载的下标运算符返回对指定位置值的引用或对该 const 值的引用，具体取决于容器是否为 const。at 函数具有相同的功能，但是此函数会执行边界检查。如果元素不在容器内，函数会抛出异常(out_of_range)。下图为一些标准库异常类型。

Exception type	Description
out_of_range	Indicates when a subscript is out of range—e.g., when an invalid subscript is specified to vector member function at.
invalid_argument	Indicates that an invalid argument was passed to a function.
length_error	Indicates an attempt to create too long a container, string, etc.
bad_alloc	Indicates that an attempt to allocate memory with new (or with an allocator) failed because not enough memory was available.

vector的成员函数insert:

```
1 // insert 22 as 2nd element
2 integers.insert(integers.cbegin() + 1, 22);
```

每个顺序容器都提供了重载的insert函数(数组除外，它具有固定大小，forward_list除外，它具有函数insert_after)。在上述语句中，迭代器指向容器中的第二个元素，所以将22插入容器作为新的第二个元素，原先的第二个元素就变为了第三个元素。此函数返回指向插入元素的迭代器。

vector的成员函数erase:

```
1 integers.erase(integers.cbegin()); // erase first element
2 cout << "\n\nVector integers after erasing first element: ";
3 copy(integers.cbegin(), integers.cend(), output);
4
5 // erase remaining elements
6 integers.erase(integers.cbegin(), integers.cend());
7 cout << "\nAfter erasing all elements, vector integers "
8 <<(integers.empty() ? "is" : "is not")<<" empty";
```

每个first-class containers都提供了erase函数(数组除外，它具有固定大小，forward_list除外，它具有函数erase_after)。第一行代码擦除了容器中第一个元素。第二个erase函数擦除两个迭代器指明的元素位置之间的所有元素。empty函数(所有的容器和适配器都可用)用以证明容器已经为空。

通常，“擦除”会销毁从容器中擦除的对象。但是，擦除作为指向动态分配对象的指针的元素不会删除动态分配的内存，这可能会导致内存泄漏。如果该元素是unique_ptr(第17.9节)，则unique_ptr将被销毁，动态分配的内存将被删除。如果元素是shared_ptr(第24章)，则动态分配对象的引用计数将递减，并且仅当引用计数达到0时才会删除内存。

vector的成员函数insert(三个参数):

```
1 integers.insert(integers.cbegin(), values.cbegin(),
2 values.cend());
```

上述代码将values的全部元素都插入到vector中，从vector的第二个元素的位置开始插入。函数返回第一个插入的元素的迭代器，如果没有插入任何元素，则返回函数的第一个参数。

vector的成员函数clear:

clear函数(在除了array以外的所有first-class containers均可以找到)清空vector---这不一定将vector的任何内存返回给系统。

14.5.2 顺序容器-list

顺序容器list, 来自头文件<list>。允许在容器的任意位置进行插入和删除操作。如果大部分插入和删除操作都出现在容器的最后, deque容器会更加有效。类模板list是作为双向链表实现的, 即列表中的每个节点都包含指向列表中前一个节点和列表中下一个节点的指针。这使类模板列表能够支持双向迭代器, 这些迭代器允许向前和向后遍历容器。任何需要输入、输出、正向或双向迭代器的算法都可以对列表进行操作。许多列表成员函数将容器的元素作为一组有序的元素进行操作。

列表(list)

◆ 特点

- 在任意位置插入和删除元素都很快
- 不支持随机访问

◆ 接合(splice)操作

- `s1.splice(p, s2, q1, q2)` : 将s2中[q1, q2)移动到s1中p所指向元素之前

C++11: forward_list容器:

单向链表 (forward_list)

- ◆ 单向链表每个结点只有指向下一个结点的指针, 没有简单的方法来获取一个结点的前驱;
- ◆ 未定义insert、emplace和erase操作, 而定义了insert_after、emplace_after和erase_after操作, 其参数与list的insert、emplace和erase相同, 但并不是插入或删除迭代器p1所指的元素, 而是对p1所指元素之后的结点进行操作;
- ◆ 不支持size操作。

forward_list的头文件为<forward_list>实现为单向链表, 即列表中的每个节点都包含指向list中下一个节点的指针。这使类模板list能够支持正向迭代器, 这些迭代器允许容器在正向遍历。任何需要输入、输出或正向迭代器的算法都可以在forward_list

list成员函数：

```
1 #include <iostream>
2 #include <vector>
3 #include <list>
4 #include <algorithm>
5 #include <iterator>
6 using namespace std;
7
8 //prototype for function template printList
9 template <typename T> void printList(const list<T>& listRef);
10
11 int main(){
12     list<int> values;
13     list<int> othervalues;
14
15     //insert items in values
16     values.push_front(1);
17     values.push_front(2);
18     values.push_back(4);
19     values.push_back(3);
20
21     cout << "values contains: ";
22     printList(values);
23
24     values.sort(); // sort values
25     cout << "\nvalues after sorting contains: ";
26     printList(values);
27
28     // insert elements of ints into othervalues
29     vector<int> ints{2, 6, 4, 8};
30     othervalues.insert(othervalues.cbegin(), ints.cbegin(),
31     ints.cend());
31     cout << "\nAfter insert, othervalues contains: ";
32     printList(othervalues);
33
34     // remove othervalues elements and insert at end of values
35     values.splice(values.cend(), othervalues);
36     cout << "\nAfter splice, values contains: ";
37     printList(values);
38
39     values.sort(); // sort values
```

```
40     cout << "\nAfter sort, values contains: ";
41     printList(values);
42
43     // insert elements of ints into othervalues
44     othervalues.insert(othervalues.cbegin(), ints.cbegin(),
45     ints.cend());
46     othervalues.sort(); // sort the list
47     cout << "\nAfter insert and sort, othervalues contains: ";
48     printList(othervalues);
49
50     // remove othervalues elements and insert into values in
51     // sorted order
52     values.merge(othervalues);
53     cout << "\nAfter merge:\n values contains: ";
54     printList(values);
55     cout << "\n othervalues contains: ";
56     printList(othervalues);
57
58     values.pop_front(); // remove element from front
59     values.pop_back(); // remove element from back
60     cout << "\nAfter pop_front and pop_back:\n values contains:
61     ";
62     printList(values);
63
64     values.unique(); // remove duplicate elements
65     cout << "\nAfter unique, values contains: ";
66     printList(values);
67
68     values.swap(othervalues); // swap elements of values and
69     // othervalues
70     cout << "\nAfter swap:\n values contains: ";
71     printList(values);
72     cout << "\n othervalues contains: ";
73     printList(othervalues);
74
75     // replace contents of values with elements of othervalues
76     values.assign(othervalues.cbegin(), othervalues.cend());
77     cout << "\nAfter assign, values contains: ";
78     printList(values);
79
80     // remove othervalues elements and insert into values in
81     // sorted order
```

```

77     values.merge(othervalues);
78     cout << "\nAfter merge, values contains: ";
79     printList(values);
80
81     values.remove(4); // remove all 4s
82     cout << "\nAfter remove(4), values contains: ";
83     cout << "\nAfter remove(4), values contains: ";
84     printList(values);
85     cout << endl;
86 }
87
88 // printList function template definition; uses
89 // ostream_iterator and copy algorithm to output list elements
90 template <typename T> void printList(const list<T>& listRef){
91     if (listRef.empty()){
92         cout << "List is empty";
93     }
94     else{
95         ostream_iterator<T> output{cout, " "};
96         copy(listRef.cbegin(), listRef.cend(), output);
97     }
98 }
```

```

D:\clionproject\untitled\cmake-build-debug\untitled.exe
values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
After insert, otherValues contains: 2 6 4 8
After splice, values contains: 1 2 3 4 2 6 4 8
After sort, values contains: 1 2 2 3 4 4 6 8
After insert and sort, otherValues contains: 2 4 6 8
After merge:
values contains: 1 2 2 2 3 4 4 4 6 6 8 8
otherValues contains: List is empty
After pop_front and pop_back:
values contains: 2 2 2 3 4 4 4 6 6 8
After unique, values contains: 2 3 4 6 8
After swap:
values contains: List is empty
otherValues contains: 2 3 4 6 8
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8 8
After remove(4), values contains:
After remove(4), values contains: 2 2 3 3 6 6 8 8
```

创建list对象：

```
1 list<int> values;
2 list<int> othervalues;
```

上述代码创建了两个list对象，可以存储int类型的值。

```
1 values.push_front(1);
2 values.push_front(2);
```

上述代码使用函数push_front将整数插入到values的起始位置。push_front函数只能在forward_list,list和deque中使用。push_back函数将值插入到容器的末尾位置。push_back函数在除了array和forward_list之外的所有顺序容器中都可以使用。

list的成员函数sort：

成员函数sort将list中的元素以升序排列。

```
1 values.sort(); // sort values
```

list的成员函数splice：

splice函数删除otherValues中的元素并将这些元素插入到values指定位置的前面。

```
1 // remove othervalues elements and insert at end of values
2 values.splice(values.cend(), othervalues);
```

list的成员函数merge：

```
1 // remove othervalues elements and insert into values in sorted
order
2 values.merge(othervalues);
```

merge函数移除otherValues的所有元素，并将他们排序后插入到values中。进行操作的两个list必须先进行相同的排序后再使用merge函数。

list成员函数pop_front：

```
1 values.pop_front(); // remove element from front
2 values.pop_back(); // remove element from back
```

pop_front函数移除list中的第一个元素。pop_back移除list中的最后一个函数。

list的成员函数unique:

unique函数移除list中的重复元素。在进行此操作之前，需要先将list排序。

```
1 | values.unique(); // remove duplicate elements
```

list成员函数swap:

swap函数(available to all first-class containers)交换两个list中的元素。

```
1 | values.swap(othervalues); // swap elements of values and  
othervalues
```

list成员函数assign和remove:

assign函数(available to all sequence containers)将指定范围的otherValues中的元素插入到values中。

```
1 | // replace contents of values with elements of othervalues  
2 | values.assign(othervalues.cbegin(), othervalues.cend());
```

```
1 | values.remove(4); // remove all 4s
```

remove函数将values中的数值4删除。

14.5.3 顺序容器-deque

The term deque is short for “double-ended queue.”类 deque 提供对随机访问迭代器的支持，因此 deque 可以与所有标准库算法一起使用。One of the most common uses of a deque is to maintain a first-in, first-out queue of elements. 事实上，deque 是queue适配器的默认底层实现

deque 的额外存储可以在 deque 的任一端以内存块的形式分配，这些内存块通常作为指向这些块的指针的内置数组进行维护。由于 deque 的内存布局不连续，deque 迭代器必须比用于遍历向量，arrays和内置数组的指针更“智能”。

双端队列 (deque)

◆ 特点

- 在两端插入或删除元素快
- 在中间插入或删除元素慢
- 随机访问较快，但比向量容器慢

一般的，使用deque的开销比vector更大。

在deque中间的插入和删除经过优化，以最大程度地减少复制的元素数量，因此它比向量更有效，但比列表的效率逊色。

使用类deque时必须包含头文件<deque>

```
1 #include <iostream>
2 #include <deque>
3 #include <algorithm>
4 #include <iterator>
5 using namespace std;
6
7 int main() {
8     deque<double> values;
9     ostream_iterator<double> output{cout, " "};
10
11     //insert elements in values
12     values.push_front(2.2);
13     values.push_front(3.5);
14     values.push_back(1.1);
15
16     cout << "values contains: ";
17
18     // use subscript operator to obtain elements of values
19     for (size_t i{0}; i < values.size(); ++i){
20         cout << values[i] << ' ';
21     }
22
23     values.pop_front(); // remove first element
24     cout << "\nAfter pop_front, values contains: ";
25     copy(values.cbegin(), values.cend(), output);
26
27     // use subscript operator to modify element at location 1
```

```
28     values[1] = 5.4;
29     cout << "\nAfter values[1] = 5.4, values contains: ";
30     copy(values.cbegin(), values.cend(), output);
31     cout << endl;
32
33 }
```

```
D:\clionproject\test_11_17\cmake-build-debug\test_11_17.exe
values contains: 3.5 2.2 1.1
After pop_front, values contains: 2.2 1.1
After values[1] = 5.4, values contains: 2.2 5.4
```

进程已结束，退出代码为 0

14.5.4 顺序容器的插入迭代器

顺序容器的插入迭代器

- ◊ 用于向容器头部、尾部或中间指定位置插入元素的迭代器
- ◊ 包括前插迭代器 (`front_inserter`) 、后插迭代器 (`back_inserter`) 和任意位置插入迭代器 (`inserter`)
- ◊ 例：

```
list<int> s;
back_inserter iter(s);
*(iter++) = 5; //通过iter把5插入s末尾
```

14.6 关联容器

关联容器的特点和接口

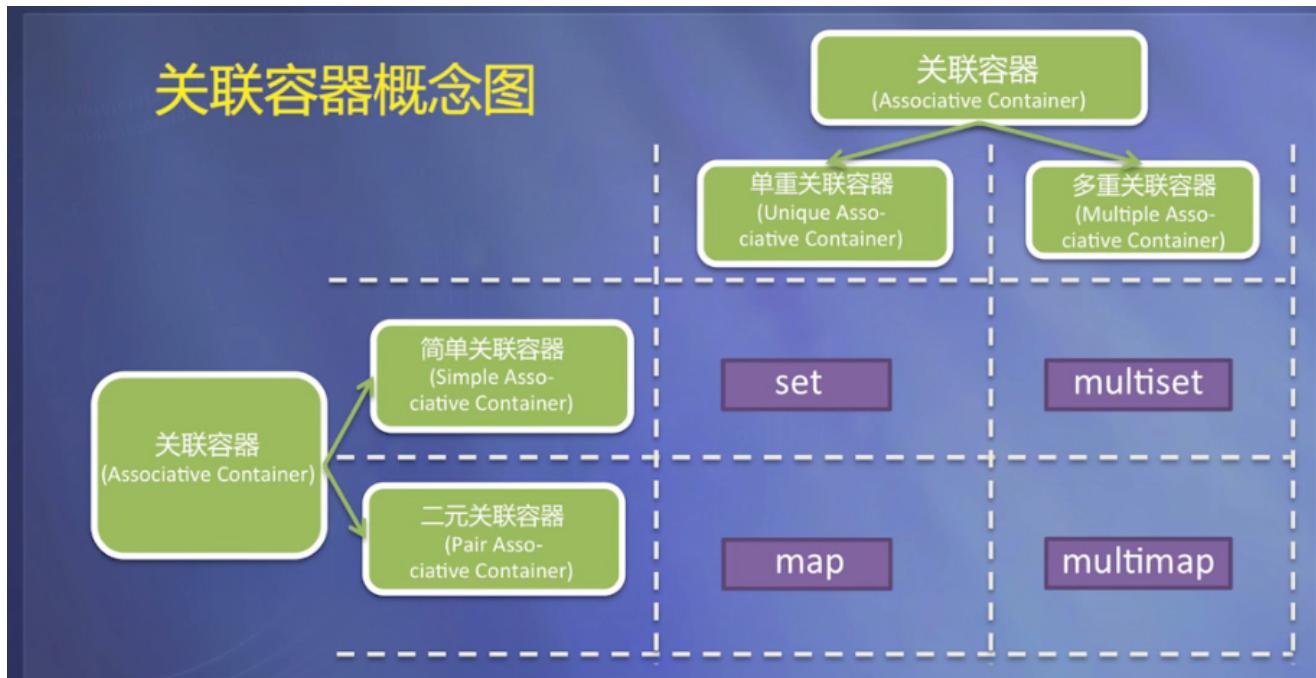
◆ 关联容器的特点

- 每个关联容器都有一个键(key)
- 可以根据键高效地查找元素

◆ 接口

- 插入 : insert
- 删除 : erase
- 查找 : find
- 定界 : lower_bound、upper_bound、equal_range
- 计数 : count

关联容器概念图



关联容器通过key (通常称为search keys) 提供对存储和检索元素的直接访问。四种有序关联容器为multiset, set, multimap 和 map。还有四种无序关联容器, unordered_multiset, unordered_set, unordered_multimap 和 unordered_map。



Performance Tip 15.10

The unordered associative containers might offer better performance for cases in which it's not necessary to maintain keys in sorted order.

Classes multiset and set provide operations for manipulating sets of values where the values themselves are the keys. The primary difference between a multiset and a set is that a multiset allows duplicate keys and a set does not. Classes multimap and map provide operations for manipulating values associated with keys (these values are sometimes referred to as mapped values). The primary difference

between a multimap and a map is that a multimap allows duplicate keys with associated values to be stored and a map allows only unique keys with associated values.

14.6.1 关联容器-multiset

multiset来自头文件<set>。

集合 (set)

◇ 集合用来存储一组无重复的元素。由于集合的元素本身是有序的，可以高效地查找指定元素，也可以方便地得到指定大小范围的元素在容器中所处的区间

```
1 #include <array>
2 #include <iostream>
3 #include <set>
4 #include <algorithm>
5 #include <iterator>
6 #include <vector>
7 using namespace std;
8
9 int main(){
10     multiset<int,less<int>> intMultiset;//multiset of ints
11
12     cout<<"There are currently "<<intMultiset.count(15)
13         <<" values of 15 in the multiset\n";
14
15     intMultiset.insert(15);// insert 15 in intMultiset
16     intMultiset.insert(15);// insert 15 in intMultiset
17     cout << "After inserts, there are "<<intMultiset.count(15)
18         <<" values of 15 in the multiset\n\n";
19
20     // find 15 in intMultiset; find returns iterator
21     auto result{intMultiset.find(15)};
22
23     if (result != intMultiset.end()){// if iterator not at end
24         cout << "Found value 15\n"; // found search value 15
25     }
26
27     // find 20 in intMultiset; find returns iterator
```

```

28     result = intMultiset.find(20);
29
30     if (result == intMultiset.end()){// will be true hence
31         cout << "Did not find value 20\n"; // did not find 20
32     }
33
34     // insert elements of array a into intMultiset
35     vector<int> a{7, 22, 9, 1, 15, 18, 30, 100, 22, 85, 13};
36     intMultiset.insert(a.cbegin(), a.cend());
37     cout << "\nAfter insert, intMultiset contains:\n";
38     ostream_iterator<int> output{cout, " "};
39     copy(intMultiset.begin(), intMultiset.end(), output);
40
41     // determine lower and upper bound of 22 in intMultiset
42     cout << "\n\nLower bound of 22: "
43         <<*(intMultiset.lower_bound(22));
44     cout << "\nUpper bound of 22: "<<*
45         (intMultiset.upper_bound(22));
46
47     // use equal_range to determine lower and upper bound
48     // of 22 in intMultiset
49     auto p{intMultiset.equal_range(22)};
50
51     cout << "\n\nEqual_range of 22:" << "\nlower bound: "
52         <<*(p.first)<<"\n Upper bound: "<<*(p.second);
53     cout << endl;
54 }
```

```

D:\clionproject\test_11_17\cmake-build-debug\test_11_17.exe
There are currently 0 values of 15 in the multiset
After inserts, there are 2 values of 15 in the multiset

Found value 15
Did not find value 20

After insert, intMultiset contains:
1 7 9 13 15 15 18 22 22 30 85 100

Lower bound of 22: 22
Upper bound of 22: 30

equal_range of 22:
lower bound: 22
Upper bound: 30

```

创建multiset:

下述语句创建了multiset，存储int类型的值，以升序存储。升序存储是此容器的默认设置，所以可以省略less。

```
1 | multiset<int, less<int>> intMultiset; // multiset of ints
```

上述语句可以写为：

```
1 | multiset<int> intMultiset; // multiset of ints
```

multiset成员函数count:

count函数对所有关联容器可用。

```
1 | cout<<"There are currently "<<intMultiset.count(15)
2 |           <<" values of 15 in the multiset\n";
```

上述count函数用以返回intMultiset中数值15的个数。

multiset成员函数insert:

```
1 | intMultiset.insert(15); // insert 15 in intMultiset
```

上述函数将15插入到intMultiset中。

```
1 | vector<int> a{7, 22, 9, 1, 18, 30, 100, 22, 85, 13};
2 | intMultiset.insert(a.cbegin(), a.cend());
```

上述语句将a中的元素从开始到末尾，插入到intMultiset中，插入后为排序完成的容器。

multiset成员函数find:

find函数(对所有关联容器可用)用以确定容器中元素的位置。

```
1 | auto result{intMultiset.find(15)};
```

返回一个iterator 或者 const_iterator(取决于容器是否为const的)。如果没有找到元素的位置，则返回一个和end函数返回的迭代器一样的迭代器。

multiset成员函数lower_bound和upper_bound

lower_bound 和 upper_bound对所有关联容器可用。

```
1 // determine lower and upper bound of 22 in intMultiset
2     cout << "\n\nLower bound of 22: "
3         <<*(intMultiset.lower_bound(22));
4     cout << "\nUpper bound of 22: "<<*
    (intMultiset.upper_bound(22));
```

使用函数 lower_bound 和 upper_bound 来查找 intMultiset 中最早出现值 22 以及值 22 最后一次出现之后的元素。

lower_bound 和 upper_bound 返回 iterators 或者 const_iterators 指向恰当的位置，或者在没有相应元素的情况下，指向和 end 函数返回结果一样的位置。

multiset成员函数equal_range:

```
1     auto p{intMultiset.equal_range(22)};
2
3     cout << "\n\nEqual range of 22:" << "\nlower bound: "
4         <<*(p.first) << "\n Upper bound: " <<*(p.second);
5     cout << endl;
```

equal_range 会返回一个 pair 类型的值，包含两个迭代器。此处 p 的值将会是两个 const_iterators。两个 const_iterators 分别是函数 lower_bound 和 upper_bound 返回的迭代器。 pair 类包含两个 public 数据成员， first 和 second。如果需要解引用 lower_bound 和 upper_bound 返回的迭代器，则需要提前保证返回的迭代器在容器的范围之内。

14.6.2 关联容器-set

set 关联容器，需要头文件 <set>。用于快速存储和检索容器中的唯一的键。set 容器的实现和 multiset 的实现一样，除了 set 容器要求容器中的键是各不相同的。set 支持 bidirectional iterators(不支持 random-access iterators)。集合可以用来存储一组无重复的元素。由于集合元素本身是有序的，可以高效的查找指定元素，也可以方便的得到指定大小范围的元素在容器中所处得空间。

```
1 #include <iostream>
2 #include <vector>
3 #include <set>
4 #include <algorithm>
```

```

5 #include <iostream>
6 using namespace std;
7
8 int main(){
9     vector<double> a{2.1, 4.2, 9.5, 2.1, 3.7};
10    set<double, less<double>> doubleSet{a.begin(), a.end()};
11
12    cout << "doubleSet contains: ";
13    ostream_iterator<double> output{cout, " "};
14    copy(doubleSet.begin(), doubleSet.end(), output);
15
16    //insert 13.8 in doubleSet; insert returns pair in which
17    // p.first represents location of 13.8 in doubleSet and
18    // p.second represents whether 13.8 was inserted
19    auto p{doubleSet.insert(13.8)}; // value not in set
20    cout<<"\n\n"<<*(p.first)
21        <<(p.second?"was":"was not")<<" inserted";
22    cout<<"\ndoubleSet contains: ";
23    copy(doubleSet.begin(), doubleSet.end(), output);
24
25    // insert 9.5 in doubleSet
26    p = doubleSet.insert(9.5); // value already in set
27    cout << "\n\n"<<*(p.first)
28        <<(p.second?"was":"was not")<<" inserted";
29    cout<<"\ndoubleSet contains: ";
30    copy(doubleSet.begin(), doubleSet.end(), output);
31    cout << endl;
32 }

```

```

D:\clionproject\test_11_17\cmake-build-debug\test_11_17.exe
doubleSet contains: 2.1 3.7 4.2 9.5

```

```

13.8was inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8

```

```

9.5was not inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8

```

进程已结束，退出代码为 0

14.6.3 关联容器-multimap

The elements of multimaps and maps are pairs of keys and values instead of individual values. multimap 来自头文件 `multimap`。 multimap 容器用于存储 `pair<const K, T>` 类型的键值对（其中 `K` 表示键的类型，`T` 表示值的类型），其中各个键值对的键的值不能做修改；并且，该容器也会自行根据键的大小对存储的所有键值对做排序操作。和 map 容器的区别在于，multimap 容器中可以同时存储多 (≥ 2) 个键相同的键值对。This is called a one-to-many relationship.

```
1 #include <iostream>
2 #include <map>
3 using namespace std;
4
5 int main(){
6     multimap<int, double, less<int> > pairs; // create multimap
7
8     cout << "There are currently "<< pairs.count(15)
9         << " pairs with key 15 in the multimap\n";
10
11    // insert two value_type objects in pairs
12    pairs.insert(make_pair(15, 99.3));
13    pairs.insert(make_pair(15, 2.7));
14    cout << "After inserts, there are "<< pairs.count(15)
15        << " pairs with key 15\n\n";
16
17    // insert five value_type objects in pairs
18    pairs.insert(make_pair(30, 111.11));
19    pairs.insert(make_pair(10, 22.22));
20    pairs.insert(make_pair(25, 33.333));
21    pairs.insert(make_pair(20, 9.345));
22    pairs.insert(make_pair(5, 77.54));
23
24    cout << "Multimap pairs contains:\nKey\tValue\n";
25
26    // walk through elements of pairs
27    for (auto mapItem : pairs){
28        cout << mapItem.first << '\t' << mapItem.second << '\n';
29    }
30
31    cout<<endl;
32
33 }
```

```
D:\clionproject\test_11_17\cmake-build-debug\test_11_17.exe
There are currently 0 pairs with key 15 in the multimap
After inserts, there are 2 pairs with key 15

Multimap pairs contains:
Key      Value
5        77.54
10       22.22
15       99.3
15       2.7
20       9.345
25       33.333
30       111.11
```

进程已结束，退出代码为 0

```
1 | multimap<int, double, less<int> > pairs; // create multimap
```

```
1 |     pairs.insert(make_pair(15, 99.3));
2 |     pairs.insert(make_pair(15, 2.7));
```

上述语句使用insert函数向pairs中插入新的键值对。标准库函数make_pair创造键值pair对象。在此种情况下，first成员代表int类型的键(15),second成员代表值(99.3)。make_pair函数会自动使用用户指定的键值对的类型。也可以使用列表初始化将上述代码简化如下：

```
1 | pairs.insert({15, 2.7});
```

14.6.4 关联容器-map

Duplicate keys are not allowed—a single value can be associated with each key. This is called a one-to-one mapping. 头文件<map>必须包含。

映射 (map)

- ◆ 映射与集合同属于单重关联容器，它们的主要区别在于，集合的元素类型是键本身，而映射的元素类型是由键和附加数据所构成的二元组。
- ◆ 在集合中按照键查找一个元素时，一般只是用来确定这个元素是否存在，而在映射中按照键查找一个元素时，除了能确定它的存在性外，还可以得到相应的附加数据。

```
1 #include <iostream>
2 #include <map>
3 using namespace std;
4
5 int main(){
6     map<int, double, less<int>> pairs;
7
8     // insert eight value_type objects in pairs
9     pairs.insert(make_pair(15, 2.7));
10    pairs.insert(make_pair(30, 111.11));
11    pairs.insert(make_pair(5, 1010.1));
12    pairs.insert(make_pair(10, 22.22));
13    pairs.insert(make_pair(25, 33.333));
14    pairs.insert(make_pair(5, 77.54)); // dup ignored
15    pairs.insert(make_pair(20, 9.345));
16    pairs.insert(make_pair(15, 99.3)); // dup ignored
17
18    cout << "pairs contains:\nKey\tvalue\n";
19
20    // walk through elements of pairs
21    for (auto mapItem : pairs){
22        cout << mapItem.first << '\t' << mapItem.second << '\n';
23    }
24
25    pairs[25] = 9999.99;// use subscripting to change value for
key 25
26    pairs[40] = 8765.43;// use subscripting to insert value for
key 40
27
28    cout << "\nAfter subscript operations, pairs
contains:\nKey\tValue\n";
29
30    // use const_iterator to walk through elements of pairs
31    for (auto mapItem : pairs){
32        cout << mapItem.first << '\t' << mapItem.second << '\n';
33    }
34
35    cout<<endl;
36 }
```

```
D:\clionproject\test_11_17\cmake-build-debug\test_11_17.exe
pairs contains:
Key      Value
5        1010.1
10       22.22
15       2.7
20       9.345
25       33.333
30       111.11

After subscript operations, pairs contains:
Key      Value
5        1010.1
10       22.22
15       2.7
20       9.345
25       9999.99
30       111.11
40       8765.43
```

进程已结束，退出代码为 0

14.7 容器适配器

三种容器适配器分别为：stack,queue和priority_queue。容器适配器不支持迭代器。适配器类的好处是可以选择适当的基础数据结构。三种适配器都提供push和pop函数。

顺序容器的适配器

- ◊ 以顺序容器为基础构建一些常用数据结构，是对顺序容器的封装
 - 栈(stack)：最先压入的元素最后被弹出
 - 队列(queue)：最先压入的元素最先被弹出
 - 优先级队列(priority_queue)：最“大”的元素最先被弹出

栈和队列的模板：

```
1 template <class T, class Sequence=deque<T> > class stack; //栈
2
3 template <class T, class FrontInsertionSequence=deque<T> > class
queue; //队列
```

栈可以用任何一种顺序容器作为基础容器，而队列只允许使用前插顺序容器(双端队列deque或列表list)

栈和队列共同支持的操作

- ◆ `s1 op s2` `op`可以是`==`、`!=`、`<`、`<=`、`>`、`>=`之一，它会对两个容器适配器之间的元素按字典序进行比较
- ◆ `s.size()` 返回`s`的元素个数
- ◆ `s.empty()` 返回`s`是否为空
- ◆ `s.push(t)` 将元素`t`压入到`s`中
- ◆ `s.pop()` 将一个元素从`s`中弹出，对于栈来说，每次弹出的是最后被压入的元素，而对于队列，每次被弹出的是最先被压入的元素
- ◆ 不支持迭代器，因为它们不允许对任意元素进行访问

14.7.1 stack

stack类来自头文件`<stack>`。允许在称为顶部的一端插入和删除，因此堆栈通常称为后进先出数据结构。

```
1 #include <iostream>
2 #include <stack>
3 #include <vector>
4 #include <list>
5 using namespace std;
6
7 // pushElements function-template prototype
8 template<typename T> void pushElements(T& stackRef);
9
10 // popElements function-template prototype
11 template<typename T> void popElements(T& stackRef);
12
13 int main(){
14     // stack with default underlying deque
15     stack<int> intDequeStack;
16
17     // stack with underlying vector
18     stack<int, vector<int>> intVectorStack;
19
20     // stack with underlying list
21     stack<int, list<int>> intListStack;
22
23     // push the values 0-9 onto each stack
24     cout << "Pushing onto intDequeStack: ";
25     pushElements(intDequeStack);
26     cout << "\nPushing onto intVectorStack: ";
```

```

27     pushElements(intVectorStack);
28     cout << "\nPushing onto intListStack: ";
29     pushElements(intListStack);
30     cout << endl << endl;
31
32     // display and remove elements from each stack
33     cout << "Popping from intDequeStack: ";
34     popElements(intDequeStack);
35     cout << "\nPopping from intVectorStack: ";
36     popElements(intVectorStack);
37     cout << "\nPopping from intListStack: ";
38     popElements(intListStack);
39     cout << endl;
40 }
41
42 // push elements onto stack object to which stackRef refers
43 template<typename T> void pushElements(T& stackRef){
44     for (int i{0}; i < 10; ++i){
45         stackRef.push(i); // push element onto stack
46         cout << stackRef.top() << ' '; // view (and display) top
47         element
48     }
49
50 // pop elements from stack object to which stackRef refers
51 template<typename T> void popElements(T& stackRef){
52     while(!stackRef.empty()){
53         cout << stackRef.top() << ' '; // view (and display) top
54         element
55         stackRef.pop(); // remove top element
56     }

```

```
D:\clionproject\test_11_17\cmake-build-debug\test_11_17.exe
Pushing onto intDequeStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intVectorStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intListStack: 0 1 2 3 4 5 6 7 8 9

Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0

进程已结束，退出代码为 0
```

top函数并没有移除栈中的元素。pop函数移除位于栈顶部的元素，并不返回任何值。push函数向栈中添加元素，top函数显示位于栈顶的元素，pop函数移除位于栈顶部的元素。

14.7.2 queue

queue来自头文件<queue>，queue中停留时间最长的元素是下一个删除的元素，因此队列称为先进先出（FIFO）数据结构。queue只允许从后面插入新元素，从前面删除元素。

data structure and deletions only from the *front*. A queue can store its elements in objects of the Standard Library's *list* or *deque* containers. By default, a *queue* is implemented with a *deque*. The common *queue* operations are *push* to insert an element at the back of the *queue* (implemented by calling function *push_back* of the underlying container), *pop* to remove the element at the front of the *queue* (implemented by calling function *pop_front* of the underlying container), *front* to get a reference to the *first* element in the *queue* (implemented by calling function *front* of the underlying container), *back* to get a reference to the *last* element in the *queue* (implemented by calling function *back* of the underlying container), *empty* to determine whether the *queue* is *empty* (this calls *empty* on underlying container) and *size* to get the number of elements in the *queue* (this calls *size* on the underlying container). In Chapter 19, we'll show you how to develop your own

```
1 #include <iostream>
2 #include <queue>
3 using namespace std;
4
5 int main(){
6     queue<double> values; // queue with doubles
7
8     // push elements onto queue values
9     values.push(3.2);
10    values.push(9.8);
```

```

11     values.push(5.4);
12
13     cout << "Popping from values: ";
14
15     // pop elements from queue
16     while(!values.empty()){
17         cout << values.front() << ' '; // view front element
18         values.pop(); // remove element
19     }
20     cout<<endl;
21 }

```

```

D:\clionproject\test_11_17\cmake-build-debug\test_11_17.exe
Popping from values: 3.2 9.8 5.4

```

进程已结束，退出代码为 0

14.7.3 priority_queue

priority_queue来自头文件<queue>。允许按排序后的顺序将元素插入到容器中，并从前面删除元素。默认情况下，priority_queue的元素存储在向量中。将元素添加到priority_queue时，它们将按优先级顺序插入，这样优先级最高的元素（即最大值）将成为从priority_queue中删除的第一个元素。这通常是通过在称为堆的数据结构中排列元素来实现的（不要与动态分配的内存的堆混淆），该堆始终在数据结构的前面保持最大值（即最高优先级的元素）。默认情况下，元素的比较是使用comparator function object less<T>，但您可以提供不同的比较器。

优先级队列

- ◊ 优先级队列也像栈和队列一样支持元素的压入和弹出，但元素弹出的顺序与元素的大小有关，每次弹出的总是容器中最“大”的一个元素。
`template <class T, class Sequence=vector<T>> class priority_queue;`
- ◊ 优先级队列的基础容器必须是支持随机访问的顺序容器。
- ◊ 支持栈和队列的size、empty、push、pop几个成员函数，用法与栈和队列相同。
- ◊ 优先级队列并不支持比较操作。
- ◊ 与栈类似，优先级队列提供一个top函数，可以获得下一个即将被弹出元素（即最“大”的元素）的引用。

有几种常见的priority_queue操作。函数push根据priority_queue的优先级顺序在适当的位置插入元素，然后按优先级顺序对元素进行重新排序。函数pop删除priority_queue中优先级最高的元素。top获取对priority_queue的top元素的引用（通过调用基础容器的函数front来实现）。empty确定priority_queue是否为空（通过调用基础容器的函数empty来实现）。size获取priority_queue中的元素数（通过调用基础容器的函数size实现）。

```
1 #include <iostream>
2 #include <queue>
3 using namespace std;
4
5 int main(){
6     priority_queue<double> priorities; // create priority_queue
7
8     // push elements onto priorities
9     priorities.push(3.2);
10    priorities.push(9.8);
11    priorities.push(5.4);
12
13    cout << "Popping from priorities: ";
14
15    // pop element from priority_queue
16    while(!priorities.empty()){
17        cout << priorities.top() << ' '; // view top element
18        priorities.pop(); // remove top element
19    }
20    cout<<endl;
21 }
```

```
:
D:\clionproject\test_11_17\cmake-build-debug\test_11_17.exe
Popping from priorities: 9.8 5.4 3.2
```

进程已结束，退出代码为 0

14.8 bitset类

bitset是C++标准库中的一个类，用于表示二进制位序列。它提供了一种方便的方式来处理二进制数据，尤其适用于位运算操作。表示一个固定长度的位序列，每个位都只能是0或1。这个固定长度在创建对象时指定，并且不能在运行时更改。类似于整数类型，`std::bitset`支持多种操作，包括位运算、位查询和位设置。

```
1 #include <bitset>
2 .
3 std::bitset<N> bitset1; // 创建一个长度为 N 的 bitset，所有位都被初始化为 0
4 std::bitset<N> bitset2(value); // 使用二进制整数 value 初始化一个长度为 N 的 bitset
5 std::bitset<N> bitset3(string); // 使用二进制字符串 string 初始化一个长度为 N 的 bitset
6 std::bitset<N> bitset4(bitset); // 使用另一个 bitset 初始化一个长度为 N 的 bitset
```

其中，`value` 是一个无符号整数，`string` 是一个只包含 '0' 和 '1' 的字符串，`bitset` 是另一个 `std::bitset` 对象。

- `size()` 返回 `std::bitset` 的长度
- `count()` 返回 `std::bitset` 中值为 1 的位的数量
- `any()` 返回 `std::bitset` 中是否存在值为 1 的位
- `none()` 返回 `std::bitset` 中是否所有位都是 0
- `all()` 返回 `std::bitset` 中是否所有位都是 1
- `test(pos)` 返回 `std::bitset` 中位于 `pos` 位置的值
- `set(pos)` 将 `std::bitset` 中位于 `pos` 位置的值设为 1
- `reset(pos)` 将 `std::bitset` 中位于 `pos` 位置的值设为 0
- `flip(pos)` 将 `std::bitset` 中位于 `pos` 位置的值取反
- `to_ulong()` 返回 `std::bitset` 转换成的无符号整数值
- `to_ullong()` 返回 `std::bitset` 转换成的无符号长整数值

`std::bitset` 重载了许多二进制运算符，如 &、|、^、~ 等，使其支持类似于整数类型的位运算操作。例如：

```
1 std::bitset<4> bitset1("1010");
2 std::bitset<4> bitset2("0110");
3
4 std::bitset<4> bitset3 = bitset1 & bitset2; // 按位与运算
5 std::bitset<4> bitset4 = bitset1 | bitset2; // 按位或运算
6 std::bitset<4> bitset5 = bitset1 ^ bitset2; // 按位异或运算
7 std::bitset<4> bitset6 = ~bitset
```

还可以使用左移、右移运算符进行位移操作：

```
1 std::bitset<4> bitset1("0101");
2
3 std::bitset<4> bitset2 = bitset1 << 2; // 左移 2 位, 结果为
4 "010100"
5 std::bitset<4> bitset3 = bitset1 >> 1; // 右移 1 位, 结果为 "0010"
```

`std::bitset` 还支持 `to_string()` 方法，将其转换成二进制字符串表示：

```
1 std::bitset<4> bitset1("1010");
2 std::string str = bitset1.to_string(); // "1010"
```

`std::bitset` 可以作为容器类型使用，可以使用下标访问、迭代器等方式访问其元素。

14.9 auto关键字

`auto` 是在变量定义的时候，用别人的类型作为自己类型的一种定义变量的方式。也叫做 **类型自动推断**。

1. `auto` 必须在定义的时候初始化。

```
1 int a = 1;
2 auto b = a; // b的类型为a的类型 int
3 auto b1; // 编译器无法推导b1的类型
4 b1=10; // 错误！
```

2. 定义在一个 `auto` 序列的变量必须始终推导成同一类型

```
1 auto a1=10, a2=20; // 正确
2 auto b1=20, b2=2.5 // 错误, 没有推导为同一类型
```

3.如果初始化表达式是引用或const,则去除引用或cons语义。auto不管&和const

```
1 int a{10};int& b=a;
2 auto c=b;//c的类型为int而非int&
3
4 const int a1{90};
5 auto b1=a1;//b1的类型为int而非const int
```

4.如果auto关键字带上&号，则不去除引用或const语义。

```
1 int a{10};int& b=a;
2 auto& c=b;//c的类型为int&
3
4 const int a1{90};
5 auto& b1=a1;//b1的类型为const int
```

5.初始化表达式为数组时，auto关键字推导类型为指针。

```
1 int a3[3]={1,3,4};
2 auto b3=a3;
3 cout<<typeid(b3).name()<<endl;//输出为int*
```

6.若表达式为数组且auto带上&，则推导类型为数组类型。

```
1 int a3[3]={2,4,5};
2 auto& b7=a3;
3 cout<<typeid(b7).name()<<endl;//输出int [3]
```

7.C++14中，auto可以作为函数的返回值类型和参数类型。

8.要避免在一行中使用直接列表初始化和拷贝列表初始化。

```
1 auto x{1},y={2};//有问题，不要同时使用直接和拷贝列表初始化
```

示例：

Classic C++ Style (经典C++风格)	Modern C++ Style (现代C++风格)
int x = 42;	auto x = 42;
float x = 42.;	auto x = 42.f;
unsigned long x = 42;	auto x = 42ul;
std::string x = "42";	auto x = "42"s; //c++14
chrono::nanoseconds x{ 42 };	auto x = 42ns; //c++14
int f(double);	auto f (double) -> int; auto f (double) { /*...*/ }; auto f = [] (double) { /*... */ }; //匿名函数

auto最常用的场景是对于复杂类型的简化。

```

1 #include <iostream>
2 #include <typeinfo>
3 using std::cout;
4 using std::cin;
5 using std::endl;
6
7
8 auto max(int x,int y){
9     return x>y?x:y;
10 }
11
12 int main() {
13     //auto变量必须在定义时初始化
14     auto x=3;
15     auto y{42};//初始化列表的方式
16     //定义在一个auto序列的变量必须始终推导成为同一类型
17     auto x1{2},x2{7},x3{4};
18     //如果初始表达式是引用或者const, 则去除引用或者const
19     int y1{4},&y2{y1};
20     auto y3{y2};
21     cout<<typeid(y3).name()<<endl;
22     //如果auto关键字带上&, 则不去除引用或const语义
23     auto& z1{y2};
24     cout<<typeid(z1).name()<<endl;
25     //初始化表达式为数组时, auto关键字推导类型为指针

```

```
26     int p[3]{1,2,3};  
27     auto p1=p;  
28     cout<<typeid(p1).name()<<endl;  
29     //若表达式为数组且auto带上&, 则推导类型为是数组类型  
30     auto& p2=p;  
31     cout<<typeid(p2).name()<<endl;  
32     //C++14中, auto可以作为函数的返回值类型和参数类型  
33     cout<<max(x1,x2)<<endl;  
34  
35     return 0;  
36 }  
37
```

尽量使用auto关键字：

使用auto是为了代码的正确性、性能、可维护性、健壮性和方便性。

对于C++的原生数组，是不能使用auto直接做类型推断的。不可以使用auto关键字来定义数组的类型。

```
1 | auto a[3]{1,4,6}; //错误！
```

14.10 decltype关键字

利用已知类型声明新变量，在编译时期推导一个表达式的类型，而不用初始化，语法格式有些像sizeof

decltype主要用于泛型编程

代码示例：

```
1 | #include <iostream>  
2 | using namespace std;  
3 |  
4 | int fun1(){  
5 |     return 10;  
6 | }  
7 |  
8 | auto fun2(){  
9 |     return 'g';  
10| } //C++14  
11|
```

```
12 int main(){
13     decltype(fun1()) x;//不会执行fun1()函数
14     decltype(fun2()) y=fun2();
15     cout<<typeid(x).name()<<endl;
16     cout<<typeid(y).name()<<endl;
17
18     return 0;
19 }
```

```
:
D:\Code\cppProject\test240407\cmake-build-debug\test240107.exe
i
c
```

进程已结束，退出代码为 0

decltype和auto都是C++11自动类型推导的关键字。它们有很多差别：

1. auto忽略最上层的const， decltype则保留最上层的const
2. auto忽略原有类型的引用， decltype则保留原有类型的引用
3. 对解引用操作， auto推断出原有类型， decltype推断出引用；
4. **auto推断时会实际执行， decltype不会执行，只做分析。**
5. 总之在使用中过程中和const、引用和指针结合时需要特别小心。

15.标准库算法

15.1 引言

无事可记

STL算法特点

- ◆ STL算法本身是一种函数模版
 - 通过迭代器获得输入数据
 - 通过函数对象对数据进行处理
 - 通过迭代器将结果输出
- ◆ STL算法是通用的，独立于具体的数据类型、容器类型

15.2 最低迭代器要求(Minimum Iterator Requirements)

容器的一个重要的部分就是它所支持的迭代器类型。这决定了容器可以使用哪种算法。例如，vector和array都支持random-access iterators。所有的标准库算法都可以用于vector，不改变容器大小的算法也可以用于array。采用迭代器参数的每个标准库算法都要求这些迭代器提供最低级别的功能。例如，如果算法需要前向迭代器，则该算法可以在支持前向迭代器、双向迭代器或随机访问迭代器的任何容器上运行。

标准库算法不依赖于它们所操作的容器的实现细节。只要容器（或内置数组）的迭代器满足算法的要求，算法就可以在容器上运行。

标准库容器的实现非常简洁。这些算法与容器分离，仅通过迭代器间接对容器的元素进行操作。这种分离使得编写适用于各种容器类的泛型算法变得更加容易。

使用产生可接受性能的“最弱迭代器”有助于生成最大可重用的组件。例如，如果算法只需要正向迭代器，则可以将其与任何支持正向迭代器、双向迭代器或随机访问迭代器的容器一起使用。但是，需要随机访问迭代器的算法只能与具有随机访问迭代器的容器一起使用。

15.2.1 迭代器失效

迭代器仅指向容器元素，因此当发生某些容器修改时，迭代器可能会失效。例如，如果在向量上调用 clear，则其所有元素都将被销毁。在调用 clear 之前指向该向量元素的任何迭代器现在都将无效。

在这里，我们总结了迭代器在insert和erase操作期间何时失效。

When *inserting* into

- a `vector`—If the `vector` is reallocated, all iterators pointing to it are invalidated. Otherwise, iterators from the insertion point to the end of the `vector` are invalidated.
- a `deque`—All iterators are invalidated.
- a `list` or `forward_list`—All iterators *remain valid*.
- an ordered associative container—All iterators *remain valid*.
- an unordered associative container—All iterators are invalidated if the container needs to be reallocated.

When erasing from a container, iterators to the erased elements are invalidated. In addition:

- for a `vector`—Iterators from the erased element to the end of the `vector` are invalidated.
- for a `deque`—If an element in the middle of the `deque` is erased, all iterators are invalidated.

15.3 Lambda表达式

lambda表达式允许自定义匿名函数，并将它们传递给函数。表达式在函数内部定义，可以使用和操作封闭函数的局部变量。

```
1 #include <iostream>
2 #include <array>
3 #include <algorithm>
4 #include <iterator>
5 using namespace std;
6
7 int main(){
8     const size_t SIZE{4};
9     array<int,SIZE> values{1,2,3,4};
10    ostream_iterator<int> output{cout, " "};
11
12    cout<<"values contains: ";
13    copy(values.cbegin(), values.cend(), output);
14    cout << "\nDisplay each element multiplied by two: ";
15
16    // output each element multiplied by two
17    for_each(values.cbegin(), values.cend(),
18              [] (auto i) {cout << i * 2 << " "});
```

```
20     // add each element to sum
21     int sum = 0; // initialize sum to zero
22     for_each(values.cbegin(), values.cend(), [&sum](auto i) {sum
23         += i;});
24
25     cout << "\nSum of value's elements is: " << sum << endl; // output sum
26 }
```

```
:
D:\clionproject\test_11_17\cmake-build-debug\test_11_17.exe
values contains: 1 2 3 4
Display each element multiplied by two: 2 4 6 8
Sum of value's elements is: 10
```

进程已结束，退出代码为 0

15.3.1 for_each算法

for_each的前两个参数代表要处理的元素的范围。

```
1 const size_t SIZE{4};
2 array<int,SIZE> values{1,2,3,4};
3 ostream_iterator<int> output{cout, " "};
4
5 cout<<"values contains: ";
6 copy(values.cbegin(), values.cend(), output);
7 cout << "\nDisplay each element multiplied by two: ";
8
9 // output each element multiplied by two
10 for_each(values.cbegin(), values.cend(),
11           [] (auto i) {cout << i * 2 << " "});
```

上述代码中，处理范围从values的第一个元素到最后一个元素。第三个参数指定范围内每个元素要调用的函数。对于上例，for_each将当前的每个元素传递给lambda表达式作为lambda表达式的参数，然后表达式使用这些元素执行它的功能。

15.3.2 Lambda with an Empty Introducer

```
1 [] (auto i) {cout << i * 2 << " "};
```

([])此方括号为lambda introducer，后面跟参数列表和函数体。lambda表达式可以使用表达式定义位置的函数的变量。lambda introducer允许指定表达式使用的变量。在C++11中必须将参数的类型明确指定，在C++14中可以使用自动类型推断(auto)。

```
1 void timesTwo(int i){  
2     cout << i * 2 << " "  
3 }
```

如果我们实现定义了如上所述的函数，for_each函数就可以将上述函数作为第三个参数，如下所示：

```
1 for_each(values.cbegin(), values.cend(), timesTwo);
```

15.3.3 Lambda with a Nonempty Introducer—Capturing Local Variables

```
1 for_each(values.cbegin(), values.cend(), [&sum](auto i) {sum += i;});
```

第二次调用for_each函数，再次使用lambda表达式：

```
1 [&sum](auto i) {sum += i;}
```

此表达式通过引用捕获局部变量sum。

15.3.4 Lambda表达式的返回类型

如果lambda的函数体包含如下类型的语句，编译器可以推断lambda表达式的返回类型：

```
1 return expression;
```

其他情况下，lambda表达式的返回类型为void，除非显式的指定表达式的返回类型：

```
1 [](parameterList) -> type {lambdaBody}
```

15.4 算法

下述为C++中的几个算法

15.4.1 fill, fill_n, generate and generate_n

```
1 #include <iostream>
2 #include <algorithm>
3 #include <array>
4 #include <iterator>
5 using namespace std;
6
7 // generator function returns next letter (starts with A)
8 char nextLetter(){
9     static char letter{'A'};
10    return letter++;
11 }
12
13 int main(){
14     array<char,10> chars;
15     fill(chars.begin(),chars.end(),'5');//fill chars with 5s
16
17     cout<<"chars after filling with 5s:\n";
18     ostream_iterator<char> output{cout," "};
19     copy(chars.cbegin(),chars.cend(),output);
20
21     //fill first five elements of chars with As
22     fill_n(chars.begin(),5,'A');
23
24     cout << "\n\nchars after filling five elements with As:\n";
25     copy(chars.cbegin(), chars.cend(), output);
26
27     // generate values for all elements of chars with nextLetter
28     generate(chars.begin(), chars.end(), nextLetter);
29
30     cout << "\n\nchars after generating letters A-J:\n";
31     copy(chars.cbegin(), chars.cend(), output);
32
33     // generate values for first five elements of chars with
34     nextLetter
35     generate_n(chars.begin(), 5, nextLetter);
36
37     cout << "\n\nchars after generating K-O for the"
38         <<" first five elements:\n";
```

```

38     copy(chars.cbegin(), chars.cend(), output);
39     cout << endl;
40
41     // generate values for first three elements of chars with a
42     // lambda
43     generate_n(chars.begin(), 3, [](){
44         static char letter{'A'};
45         return letter++;
46     });
47
48     cout << "\nchars after using a lambda to generate A-C "
49     <<"for the first three elements:\n";
50     copy(chars.cbegin(), chars.cend(), output);
51     cout << endl;
52 }
```

```

:
D:\clionproject\test_11_17\cmake-build-debug\test_11_17.exe
chars after filling with 5s:
5 5 5 5 5 5 5 5 5 5

chars after filling five elements with As:
A A A A A 5 5 5 5 5

chars after generating letters A-J:
A B C D E F G H I J

chars after generating K-O for the first five elements:
K L M N O F G H I J

chars after using a lambda to generate A-C for the first three elements:
A B C N O F G H I J

进程已结束，退出代码为 0
```

fill算法:

```
1 | fill(chars.begin(), chars.end(), '5');//fill chars with 5s
```

上述代码，将字符5放置到从chars.begin()开始到chars.end()结束的每个位置上。两个参数都至少应该是forward iterators。

fill_n算法:

```
1 //fill first five elements of chars with As
2 fill_n(chars.begin(), 5, 'A');
```

上述代码将字符A放入到chars的前五个元素的位置。第一个参数至少应该是output iterator。第二个参数指明了应该插入几个元素。第三个参数指明了插入的元素。

generate算法:

```
1 // generate values for all elements of chars with nextLetter
2 generate(chars.begin(), chars.end(), nextLetter);
```

generate 算法将对生成器函数 nextLetter 的调用结果放在从 chars.begin () 到 chars.end () 的每个 chars 元素中，但不包括 chars.end ()。作为第一个和第二个参数提供的迭代器必须至少是正向迭代器。函数 nextLetter 定义一个名为 letter 的静态局部 char 变量，并将其初始化为“A”。第 12 行中的语句返回 letter 的当前值，然后对其值进行后递增，以便在下次调用该函数时使用。

generate_n算法:

```
1 // generate values for first five elements of chars with
2 nextLetter
3 generate_n(chars.begin(), 5, nextLetter);
```

使用 generate_n 算法将对生成器函数 nextLetter 的调用结果放在 chars 的五个元素中，从 chars.begin () 开始。作为第一个参数提供的迭代器必须至少是输出迭代器。

在generate_n算法中使用lambda表达式:

```
1 // generate values for first three elements of chars with a
2 lambda
3 generate_n(chars.begin(), 3, [](){
4     static char letter{'A'};
5     return letter++;
6});
```

equal, mismatch and lexicographical_compare

15.5 函数对象(仿函数)

Function object	Type	Function object	Type
divides<T>	arithmetic	logical_or<T>	logical
equal_to<T>	relational	minus<T>	arithmetic
greater<T>	relational	modulus<T>	arithmetic
greater_equal<T>	relational	negate<T>	arithmetic
less<T>	relational	not_equal_to<T>	relational
less_equal<T>	relational	plus<T>	arithmetic
logical_and<T>	logical	multiplies<T>	arithmetic
logical_not<T>	logical		

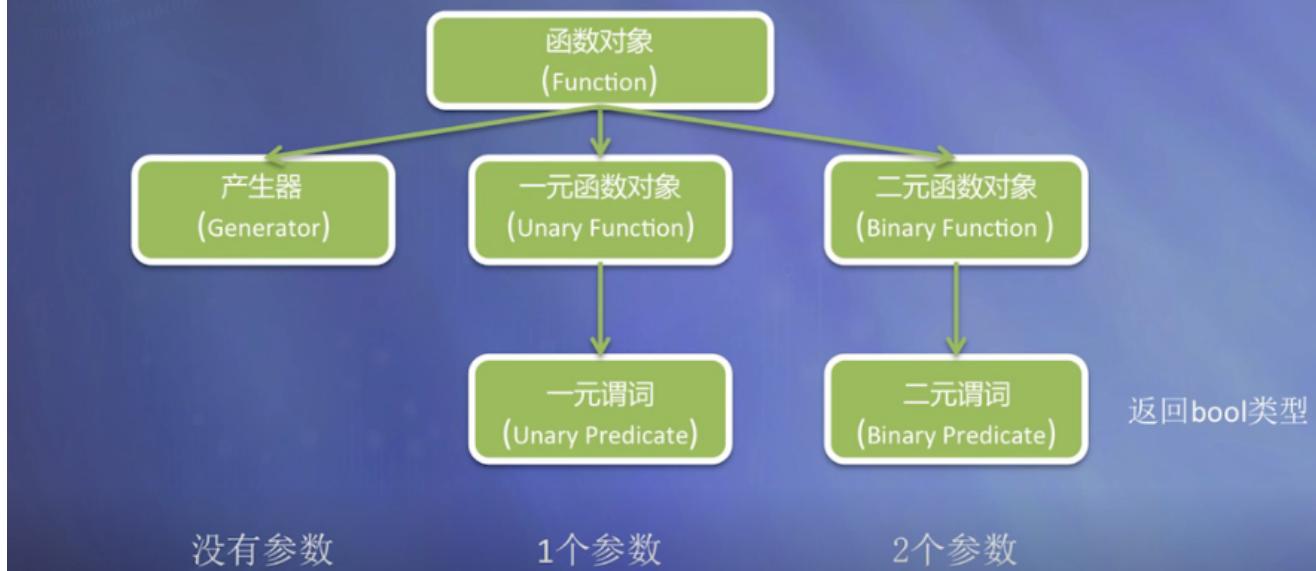
Fig. 16.14 | Some function objects in the Standard Library.

重载函数调用操作符的类，其对象常称为函数对象 (function object)，即它们是行为类似函数的对象，也叫仿函数(functor)。其实就是重载“()”操作符，使得类对象可以像函数那样调用。函数对象可以没有参数，也可以带有若干参数，其功能是获取一个值，或者改变操作的状态。

STL的基本组件——函数对象 (function object)

- ◆ 一个行为类似函数的对象，对它可以像调用函数一样调用。
- ◆ 函数对象是泛化的函数：任何普通的函数和任何重载了“()”运算符的类的对象都可以作为函数对象使用
- ◆ 使用STL的函数对象，需要包含头文件<functional>

函数对象概念图



如果一个类将 `()` 运算符重载为成员函数，这个类就称为函数对象类，这个类的对象就是函数对象。函数对象是一个对象，但是使用的形式看起来像函数调用，实际上也执行了函数调用，因而得名。**返回值为bool类型的一元函数对象，就称之为一元谓词。二元谓词同理。**

```
1 #include <iostream>
2 using namespace std;
3
4 class CAverage
5 {
6 public:
7     double operator()(int a1, int a2, int a3)
8     { //重载()运算符
9         return (double)(a1 + a2 + a3) / 3;
10    }
11 };
12
13 int main()
14 {
15     CAverage average; //能够求三个整数平均数的函数对象
16     cout << average(3, 2, 3); //等价于 cout <<
17     average.operator(3, 2, 3);
18 }
```

○是目数不限的运算符，因此重载为成员函数时，有多少个参数都可以。average是一个对象，average(3, 2, 3)实际上就是average.operator(3, 2, 3)，这使得average看上去像函数的名字，故称其为函数对象。

15.5.1 函数对象应用示例

C++STL中有以下实现“累加”功能的算法（函数模板）：

```
1 template <class Init, class T, class Pred>
2 T accumulate(Init first, Init last, T val, Pred op);
```

```
1 #include <iostream>
2 #include <array>
3 #include <algorithm>
4 #include <numeric>
5 #include <functional>
6 #include <iterator>
7 using namespace std;
8
9 // binary function adds square of its second argument and the
10 // running total in its first argument, then returns the sum
11 int sumSquares(int total,int value){
12     return total + value * value;
13 }
14
15 // class template SumSquaresClass defines overloaded operator()
16 // that adds the square of its second argument and running
17 // total in its first argument, then returns sum
18 template<typename T>
19 class SumSquaresClass{
20 public:
21     // add square of value to total and return result
22     T operator()(const T& total, const T& value){
23         return total + value * value;
24     }
25 };
26
27 int main(){
28     const size_t SIZE{10};
29     array<int, SIZE> integers{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
30     ostream_iterator<int> output{cout, " "}; //why?
31     ostream_iterator 迭代器
```

```

31
32     cout << "array integers contains: ";
33     copy(integers.cbegin(), integers.cend(), output);
34
35     // calculate sum of squares of elements of array integers
36     // using binary function sumSquares
37     int result{accumulate(integers.cbegin(), integers.cend(), 0,
38     sumSquares)};
39
39     cout << "\n\nSum of squares of integers' elements using "
40             << "binary function sumSquares: " << result;
41
42     // calculate sum of squares of elements of array integers
43     // using binary function object
44     result = accumulate(integers.cbegin(), integers.cend(), 0,
45     SumSquaresClass<int>{});
46
46     cout << "\n\nSum of squares of integers' elements using
47     binary"
47             << "\nfunction object of type SumSquaresClass<int>: "
48             << result;
49
50     // calculate sum of squares array's elements using a lambda
51     result = accumulate(integers.cbegin(), integers.cend(), 0,
52                         [](auto total, auto value){return total +
53                         value * value;});
54
54     cout << "\n\nSum of squares of integer's elements using a
55     lambda: "
55             << result << endl;
56 }

```

```

D:\clionproject\cpp_project\test_11_24\cmake-build-debug\test_11_24.exe
array integers contains: 1 2 3 4 5 6 7 8 9 10

```

```

Sum of squares of integers' elements using binary function sumSquares: 385

```

```

Sum of squares of integers' elements using binary
function object of type SumSquaresClass<int>: 385

```

```

Sum of squares of integer's elements using a lambda: 385

```

```

进程已结束, 退出代码为 0

```

16. 异常处理

16.1 引言

异常处理使您能够创建可以处理（即解决）异常的应用程序，并在发生无法或不应处理的异常时执行适当的清理。



异常处理的语法

- 抛掷异常的程序段

```
....  
throw 表达式;  
....
```

- 捕获并处理异常的程序段

```
try  
  复合语句  
  catch ( 异常声明 )  
    复合语句  
  catch ( 异常声明 )  
    复合语句  
  ...
```

- 若有异常则通过throw创建一个异常对象并抛掷
- 将可能抛出异常的程序段嵌在try块之中。通过正常的顺序执行到达try语句，然后执行try块内的保护段
- 如果在保护段执行期间没有引起异常，那么跟在try块后的catch子句就不执行。程序从try块后的最后一个catch子句后面的语句继续执行
- catch子句按其在try块后出现的顺序被检查。匹配的catch子句将捕获并处理异常（或继续抛掷异常）。
- 如果匹配的处理器未找到，则库函数terminate将被自动调用，其缺默认能是调用abort终止程序。

异常接口声明

- 可以在函数的声明中列出这个函数可能抛掷的所有异常类型。
- 例如：
`void fun() throw(A , B , C , D);`
- 若无异常接口声明，则此函数可以抛掷任何类型的异常。
- 不抛掷任何类型异常的函数声明如下：
`void fun() throw();`

自动的析构

- 找到一个匹配的catch异常处理后
 - 初始化异常参数。
 - 将从对应的try块开始到异常被抛掷处之间构造（且尚未析构）的所有自动对象进行析构。
 - 从最后一个catch处理之后开始恢复执行。

标准异常类的继承关系



异常类	头文件	异常的含义
bad_alloc	exception	用new动态分配空间失败
bad_cast	new	执行dynamic_cast失败 (dynamic_cast参见8.7.2节)
bad_typeid	typeinfo	对某个空指针p执行typeid(*p) (typeid参见8.7.2节)
bad_exception	typeinfo	当某个函数fun()因在执行过程中抛出了异常声明所不允许的异常而调用unexpected()函数时, 若unexpected()函数又一次抛出了fun()的异常声明所不允许的异常, 且fun()的异常声明列表中有bad_exception, 则会有一个bad_exception异常在fun()的调用点被抛出
ios_base::failure	ios	用来表示C++的输入输出流执行过程中发生的错误
underflow_error	stdexcept	算术运算时向下溢出
overflow_error	stdexcept	算术运算时向上溢出
range_error	stdexcept	内部计算时发生作用域的错误
out_of_range	stdexcept	表示一个参数值不在允许的范围之内
length_error	stdexcept	尝试创建一个长度超过最大允许值的对象
invalid_argument	stdexcept	表示向函数传入无效参数
domain_error	stdexcept	执行一段程序所需要的先决条件不满足

16.2 C++ throw(抛出异常用法详解)

异常处理是许多现代编程语言中不可或缺的一部分, C++ 也不例外。通过使用 throw、try、和 catch 关键字, C++ 为程序员提供了强大的异常处理机制。

16.2.1 throw的基础用法

throw 是 C++ 异常处理机制中的一个关键字, 用于在检测到异常情况时触发异常, 语法格式如下:

1 | **throw** 异常信息

异常信息可以是一个变量, 也可以是一个表达式, 表示要抛出的异常对象。

1 | **throw** 1; // 抛出一个整数的异常
 2 | **throw** "abcd"; // 抛出一个字符串的异常
 3 | **throw** int; // 错误, 异常信息不能是类型, 必须是具体的值

当在代码中检测到一个异常情况 (如非法输入、资源耗尽等) 时, 可以使用 throw 关键字来抛出一个异常。这通常会中断当前函数的执行, 并将控制权转交给最近的 catch 块。

下列是一个完整的示例:

1 | `#include <iostream>`

```
2
3 void testFunction(int choice) {
4     if (choice == 1) {
5         throw 42;
6     }
7     else if (choice == 2) {
8         throw "String type exception";
9     }
10    else {
11        throw 1.23;
12    }
13 }
14
15 int main() {
16     try {
17         testFunction(2);
18     }
19     catch (int e) {
20         std::cout << "Caught an integer exception: " << e <<
21         std::endl;
22     }
23     catch (const char* e) {
24         std::cout << "Caught a string exception: " << e <<
25         std::endl;
26     }
27     catch (const double& e) {
28         std::cout << "Caught a standard exception: " << e <<
29         std::endl;
30     }
31 }
```

```
D:\clionproject\test_10_22\cmake-build-debug\test_10_22.exe
Caught a string exception: String type exception
```

```
进程已结束，退出代码为 0
```

16.2.2 C++异常类

throw 抛出的异常值，可以是基本数据类型，也可以是类类型。C++ 标准库中提供了一些常见的异常类，它们定义在 `<stdexcept>` 头文件中。

如下是从 `<stdexcept>` 头文件中摘录的一部分异常类：

```
1  namespace std
2  {
3      class logic_error; // logic_error: 表示程序逻辑错误的基类。
4      class domain_error; // 当数学函数的输入参数不在函数的定义域内时抛
5          出。
6      class invalid_argument; // 当函数的一个参数无效时抛出。
7      class length_error; // 当操作导致容器超过其最大允许大小时抛出。
8      class out_of_range; // 当数组或其他数据结构访问越界时抛出。
9
10     class runtime_error; // 表示运行时错误的基类。
11     class range_error; // 当数学计算的结果不可表示时抛出。
12     class overflow_error; // 当算术运算导致溢出时抛出。
13     class underflow_error; // 当算术运算导致下溢时抛出。
14 }
```

```
1 #include <iostream>
2 #include <stdexcept>
3
4 void divideNumbers(double num1, double num2) {
5     if (num2 == 0) {
6         throw std::invalid_argument("Denominator cannot be
7             zero");
8     }
9     std::cout << "Result: " << num1 / num2 << std::endl;
10
11 int main() {
12     try {
13         divideNumbers(10, 2);
14         divideNumbers(10, 0);
15     } catch (const std::invalid_argument& e) {
16         std::cerr << "Caught exception: " << e.what() <<
17             std::endl;
18     }
19 }
```

```
19     return 0;
20 }
```

```
D:\clionproject\test_10_22\cmake-build-debug\test_10_22.exe
Result: 5
Caught exception: Denominator cannot be zero
```

在上述代码中，divideNumbers() 函数接受两个浮点数，并尝试进行除法。如果除数 (denominator) 是 0，则通过 throw 关键字抛出一个 std::invalid_argument 异常。这个异常会被 main() 函数中的 catch 块捕获，并输出相应的错误消息。

16.3 C++异常处理(try和catch)

程序运行时常会碰到一些错误，例如除数为 0、年龄为负数、数组下标越界等，这些错误如果不能发现并加以处理，很可能会导致程序崩溃。C++ 异常处理机制就可以让我们捕获并处理这些错误，然后我们可以让程序沿着一条不会出错的路径继续执行，或者不得不结束程序，但在结束前可以做一些必要的工作，例如将内存中的数据写入文件、关闭打开的文件、释放分配的内存等。

运行时错误如果放任不管，系统就会执行默认的操作，终止程序运行，也就是我们常说的程序崩溃（Crash）。

一个发生运行时错误的程序：

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main(){
6     string str = "http://c.biancheng.net";
7     char ch1 = str[100]; //下标越界, ch1为垃圾值
8     cout<<ch1<<endl;
9     char ch2 = str.at(100); //下标越界, 抛出异常
10    cout<<ch2<<endl;
11    return 0;
12 }
```

运行代码，在控制台输出 ch1 的值后程序崩溃。下面我们将分析一下原因。

at() 是 string 类的一个成员函数，它会根据下标来返回字符串的一个字符。与 [] 不同，at() 会检查下标是否越界，如果越界就抛出一个异常；而 [] 不做检查，不管下标是多少都会照常访问。

所谓抛出异常，就是报告一个运行时错误，程序员可以根据错误信息来进一步处理。

上面的代码中，下标 100 显然超出了字符串 str 的长度。由于第 6 行代码不会检查下标越界，虽然有逻辑错误，但是程序能够正常运行。而第 8 行代码则不同，at() 函数检测到下标越界会抛出一个异常，这个异常可以由程序员处理，但是我们在代码中并没有处理，所以系统只能执行默认的操作，也即终止程序执行。

16.3.1 捕获异常

我们可以借助 C++ 异常机制来捕获上面的异常，避免程序崩溃。捕获异常的语法为：

```
1 try{  
2     //可能抛出异常的语句  
3 }  
4 catch(exceptionType variable){  
5     //处理异常的语句  
6 }
```

try 和 catch 都是 C++ 中的关键字，后跟语句块，不能省略 {}。try 中包含可能会抛出异常的语句，一旦有异常抛出就会被后面的 catch 捕获。从 try 的意思可以看出，它只是“检测”语句块有没有异常，如果没有发生异常，它就“检测”不到。catch 是“抓住”的意思，用来捕获并处理 try 检测到的异常；如果 try 语句块没有检测到异常（没有异常抛出），那么就不会执行 catch 中的语句。

catch 关键字后面的 exceptionType variable 指明了当前 catch 可以处理的异常类型，以及具体的出错信息。稍后再对异常类型展开讲解，当务之急是演示一下 try-catch 的用法，先有一个整体上的认识。

```
1 #include <iostream>  
2 #include <string>  
3 #include <exception>  
4 using namespace std;  
5  
6 int main(){  
7     string str = "http://c.biancheng.net";  
8  
9     try{
```

```
10     char ch1 = str[100];
11     cout<<ch1<<endl;
12 }catch(exception e){
13     cout<<"[1]out of bound!"<<endl;
14 }
15
16 try{
17     char ch2 = str.at(100);
18     cout<<ch2<<endl;
19 }catch(exception &e){ //exception类位于<exception>头文件中
20     cout<<"[2]out of bound!"<<endl;
21 }
22
23 return 0;
24 }
```

```
D:\clionproject\test_10_22\cmake-build-debug\test_10_22.exe
C
[2]out of bound!
```

可以看出，第一个 try 没有捕获到异常，输出了一个没有意义的字符（垃圾值）。因为 `[]` 不会检查下标越界，不会抛出异常，所以即使有错误，try 也检测不到。换句话说，发生异常时必须将异常明确地抛出，try 才能检测到；如果不抛出来，即使有异常 try 也检测不到。所谓抛出异常，就是明确地告诉程序发生了什么错误。

第二个 try 检测到了异常，并交给 catch 处理，执行 catch 中的语句。需要说明的是，异常一旦抛出，会立刻被 try 检测到，并且不会再执行异常点（异常发生位置）后面的语句。本例中抛出异常的位置是第 17 行的 `at()` 函数，它后面的 `cout` 语句就不会再被执行，所以看不到它的输出。

说得直接一点，检测到异常后程序的执行流会发生跳转，从异常点跳转到 catch 所在的位置，位于异常点之后的、并且在当前 try 块内的语句就都不会再执行了；即使 catch 语句成功地处理了错误，程序的执行流也不会再回退到异常点，所以这些语句永远都没有执行的机会了。本例中，第 18 行代码就是被跳过的代码。

执行完 catch 块所包含的代码后，程序会继续执行 catch 块后面的代码，就恢复了正常的执行流。

为了演示「不明确地抛出异常就检测不到异常」，大家不妨将第 10 行代码改为 `char ch1 = str[1000000000];`，访问第 100 个字符可能不会发生异常，但是访问第 1 亿个字符肯定会发生异常了，这个异常就是内存访问错误。运行更改后的程序，会发现第 10 行代码产生了异常，导致程序崩溃了，这说明 try-catch 并没有捕获到这个异常。

16.3.2 发生异常的位置

异常可以发生在当前的 try 块中，也可以发生在 try 块所调用的某个函数中，或者是所调用的函数又调用了另外的一个函数，这个另外的函数中发生了异常。这些异常，都可以被 try 检测到。

下述代码为 try 块中直接发生的异常：

```
1 #include <iostream>
2 #include <string>
3 #include <exception>
4 using namespace std;
5
6 int main(){
7     try{
8         throw "Unknown Exception"; //抛出异常
9         cout<<"This statement will not be executed."<<endl;
10    }catch(const char* &e){
11        cout<<e<<endl;
12    }
13
14    return 0;
15 }
```

下述代码为 try 块中调用的某个函数发生了异常：

```
1 #include <iostream>
2 #include <string>
3 #include <exception>
4 using namespace std;
5
6 void func(){
7     throw "Unknown Exception"; //抛出异常
8     cout<<"[1]This statement will not be executed."<<endl;
9 }
10
11 int main(){
```

```
12  try{
13      func();
14      cout<<"[2]This statement will not be executed."<<endl;
15  }catch(const char* &e){
16      cout<<e<<endl;
17  }
18
19  return 0;
20 }
```

func() 在 try 块中被调用，它抛出的异常会被 try 检测到，进而被 catch 捕获。从运行结果可以看出，func() 中的 cout 和 try 中的 cout 都没有被执行。

下述代码为 try 块中调用了某个函数，该函数又调用了另外的一个函数，这个另外的函数抛出了异常：

```
1 #include <iostream>
2 #include <string>
3 #include <exception>
4 using namespace std;
5
6 void func_inner(){
7     throw "Unknown Exception"; //抛出异常
8     cout<<"[1]This statement will not be executed."<<endl;
9 }
10
11 void func_outer(){
12     func_inner();
13     cout<<"[2]This statement will not be executed."<<endl;
14 }
15
16 int main(){
17     try{
18         func_outer();
19         cout<<"[3]This statement will not be executed."<<endl;
20     }catch(const char* &e){
21         cout<<e<<endl;
22     }
23
24     return 0;
25 }
```

发生异常后，程序的执行流会沿着函数的调用链往前回退，直到遇见 try 才停止。在这个回退过程中，调用链中剩下的代码（所有函数中未被执行的代码）都会被跳过，没有执行的机会了。

`exceptionType` 是异常类型，它指明了当前的 catch 可以处理什么类型的异常；

`variable` 是一个变量，用来接收异常信息。当程序抛出异常时，会创建一份数据，这份数据包含了错误信息，程序员可以根据这些信息来判断到底出了什么问题，接下来怎么处理。

异常既然是一份数据，那么就应该有数据类型。C++ 规定，异常类型可以是 `int`、`char`、`float`、`bool` 等基本类型，也可以是指针、数组、字符串、结构体、类等聚合类型。C++ 语言本身以及标准库中的函数抛出的异常，都是 `exception` 类或其子类的异常。也就是说，抛出异常时，会创建一个 `exception` 类或其子类的对象。

`exceptionType variable` 和函数的形参非常类似，当异常发生后，会将异常数据传递给 `variable` 这个变量，这和函数传参的过程类似。当然，只有跟 `exceptionType` 类型匹配的异常数据才会被传递给 `variable`，否则 `catch` 不会接收这份异常数据，也不会执行 `catch` 块中的语句。换句话说，`catch` 不会处理当前的异常。

我们可以将 `catch` 看做一个没有返回值的函数，当异常发生后 `catch` 会被调用，并且会接收实参（异常数据）。

但是 `catch` 和真正的函数调用又有区别：

- 真正的函数调用，形参和实参的类型必须要匹配，或者可以自动转换，否则在编译阶段就报错了。
- 而对于 `catch`，异常是在运行阶段产生的，它可以是任何类型，没法提前预测，所以不能在编译阶段判断类型是否正确，只能等到程序运行后，真的抛出异常了，再将异常类型和 `catch` 能处理的类型进行匹配，匹配成功的话就“调用”当前的 `catch`，否则就忽略当前的 `catch`。

总起来说，`catch` 和真正的函数调用相比，多了一个「在运行阶段将实参和形参匹配」的过程。另外需要注意的是，如果不希望 `catch` 处理异常数据，也可以将 `variable` 省略掉，也即写作：

```
1 try{  
2     // 可能抛出异常的语句  
3 }catch(exceptionType){  
4     // 处理异常的语句  
5 }
```

这样只会将异常类型和 `catch` 所能处理的类型进行匹配，不会传递异常数据了。

16.3.3 多级catch

```
1 try{  
2     //可能抛出异常的语句  
3 }catch (exception_type_1 e){  
4     //处理异常的语句  
5 }catch (exception_type_2 e){  
6     //处理异常的语句  
7 }  
8 //其他的catch  
9 catch (exception_type_n e){  
10    //处理异常的语句  
11 }
```

当异常发生时，程序会按照从上到下的顺序，将异常类型和 catch 所能接收的类型逐个匹配。一旦找到类型匹配的 catch 就停止检索，并将异常交给当前的 catch 处理（其他的 catch 不会被执行）。如果最终也没有找到匹配的 catch，就只能交给系统处理，终止程序的运行。

```
1 #include <iostream>  
2 #include <string>  
3 using namespace std;  
4  
5 class Base{ };  
6 class Derived: public Base{ };  
7  
8 int main(){  
9     try{  
10         throw Derived(); //抛出自己的异常类型，实际上是创建一个  
Derived类型的匿名对象  
11         cout<<"This statement will not be executed."<<endl;  
12     }catch(int){  
13         cout<<"Exception type: int"<<endl;  
14     }catch(char *){  
15         cout<<"Exception type: cahr *"<<endl;  
16     }catch(Base){ //匹配成功（向上转型）  
17         cout<<"Exception type: Base"<<endl;  
18     }catch(Derived){  
19         cout<<"Exception type: Derived"<<endl;  
20     }  
21  
22     return 0;
```

在 catch 中，我们只给出了异常类型，没有给出接收异常信息的变量。

本例中，我们定义了一个基类 Base，又从 Base 派生类出了 Derived。抛出异常时，我们创建了一个 Derived 类的匿名对象，也就是说，异常的类型是 Derived。

我们期望的是，异常被 `catch(Derived)` 捕获，但是从输出结果可以看出，异常提前被 `catch(Base)` 捕获了，这说明 catch 在匹配异常类型时发生了向上转型 (Upcasting)。

16.3.4 catch 在匹配过程中的类型转换

C/C++ 中存在多种多样的类型转换，以普通函数（非模板函数）为例，发生函数调用时，如果实参和形参的类型不是严格匹配，那么会将实参的类型进行适当的转换，以适应形参的类型，这些转换包括：

- 算数转换：例如 int 转换为 float，char 转换为 int，double 转换为 int 等。
- 向上转型：也就是派生类向基类的转换，请猛击《[C++向上转型（将派生类赋值给基类）](#)》了解详情。
- const 转换：也即将非 const 类型转换为 const 类型，例如将 char * 转换为 const char *。
- 数组或函数指针转换：如果函数形参不是引用类型，那么数组名会转换为数组指针，函数名也会转换为函数指针。
- 用户自定的类型转换。

catch 在匹配异常类型的过程中，也会进行类型转换，但是这种转换受到了更多的限制，仅能进行「向上转型」、「const 转换」和「数组或函数指针转换」，其他的都不能应用于 catch。

向上转型在上面的例子中已经发生了，下面的例子演示了 const 转换以及数组和指针的转换：

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int nums[] = {1, 2, 3};
6     try{
7         throw nums;
8         cout<<"This statement will not be executed."<<endl;

```

```
9     }catch(const int *){
10         cout<<"Exception type: const int *"  
<<endl;
11     }
12
13     return 0;
14 }
```

运行结果：

```
Exception type: const int *
```

nums 本来的类型是 `int [3]`，但是 `catch` 中没有严格匹配的类型，所以先转换为 `int *`，再转换为 `const int *`。

数组也是一种类型，数组并不等价于指针，这点已在《[数组和指针绝不等价，数组是另外一种类型](#)》和《[数组到底在什么时候会转换为指针](#)》中进行了详细讲解。

16.4 C++ exception类：C++标准异常的基类

C++语言本身或者标准库抛出的异常都是 `exception` 的子类，称为标准异常（Standard Exception）。你可以通过下面的语句来捕获所有的标准异常：

```
1 try{
2     //可能抛出异常的语句
3 }catch(exception &e){
4     //处理异常的语句
5 }
```

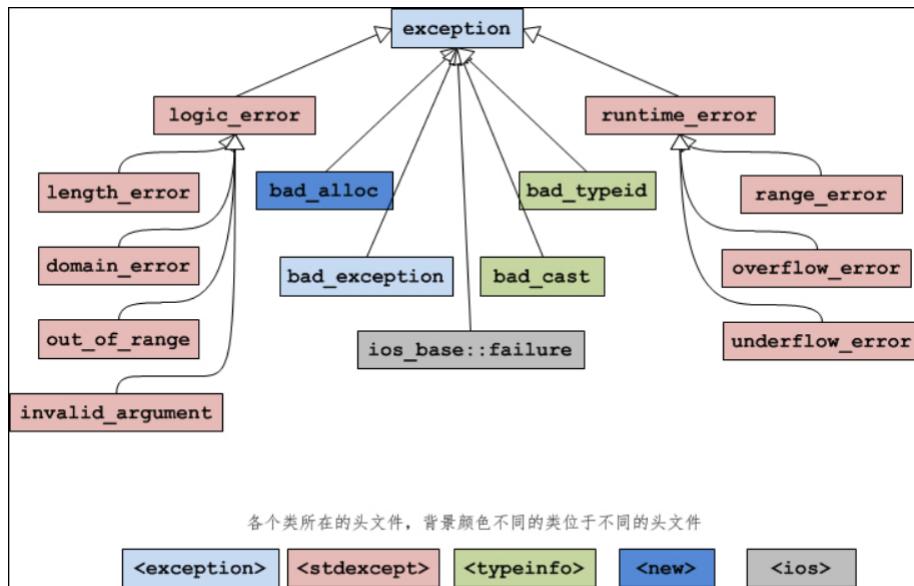
之所以使用引用，是为了提高效率。如果不使用引用，就要经历一次对象拷贝（要调用拷贝构造函数）的过程。

`exception` 类位于`<exception>`头文件中，它被声明为：

```
1 class exception{
2 public:
3     exception () throw(); //构造函数
4     exception (const exception&) throw(); //拷贝构造函数
5     exception& operator= (const exception&) throw(); //运算符重载
6     virtual ~exception() throw(); //虚析构函数
7     virtual const char* what() const throw(); //虚函数
8 }
```

这里需要说明的是 `what()` 函数。`what()` 函数返回一个能识别异常的字符串，正如它的名字“`what`”一样，可以粗略地告诉你这是什么异常。不过C++标准并没有规定这个字符串的格式，各个编译器的实现也不同，所以 `what()` 的返回值仅供参考。

下图展示了 `exception` 类的继承层次：



图：exception 类的继承层次以及它们所对应的头文件

先来看一下 `exception` 类的直接派生类：

异常名称	说明
<code>logic_error</code>	逻辑错误。
<code>runtime_error</code>	运行时错误。
<code>bad_alloc</code>	使用 <code>new</code> 或 <code>new[]</code> 分配内存失败时抛出的异常。
<code>bad_typeid</code>	使用 <code>typeid</code> 操作一个 <code>NULL</code> 指针，而且该指针是带有虚函数的类，这时抛出 <code>bad_typeid</code> 异常。
<code>bad_cast</code>	使用 <code>dynamic_cast</code> 转换失败时抛出的异常。
<code>ios_base::failure</code>	io 过程中出现的异常。
<code>bad_exception</code>	这是个特殊的异常，如果函数的异常列表里声明了 <code>bad_exception</code> 异常，当函数内部抛出了异常列表中没有的异常时，如果调用的 <code>unexpected()</code> 函数中抛出了异常，不论什么类型，都会被替换为 <code>bad_exception</code> 类型。

`logic_error` 的派生类：

异常名称	说明
<code>length_error</code>	试图生成一个超出该类型最大长度的对象时抛出该异常，例如 <code>vector</code> 的 <code>resize</code> 操作。
<code>domain_error</code>	参数的值域错误，主要用在数学函数中，例如使用一个负值调用只能操作非负数的函数。
<code>out_of_range</code>	超出有效范围。
<code>invalid_argument</code>	参数不合适。在标准库中，当利用 <code>string</code> 对象构造 <code>bitset</code> 时，而 <code>string</code> 中的字符不是 0 或 1 的时候，抛出该异常。

`runtime_error` 的派生类：

异常名称	说 明
range_error	计算结果超出了有意义的值域范围。
overflow_error	算术计算上溢。
underflow_error	算术计算下溢。

标准异常类的基础

- ◆ exception : 标准程序库异常类的公共基类。
- ◆ logic_error 表示可以在程序中被预先检测到的异常。
 - 如果小心地编写程序，这类异常能够避免。
- ◆ runtime_error 表示难以被预先检测的异常。

16.4 自定义异常类

17. 模板

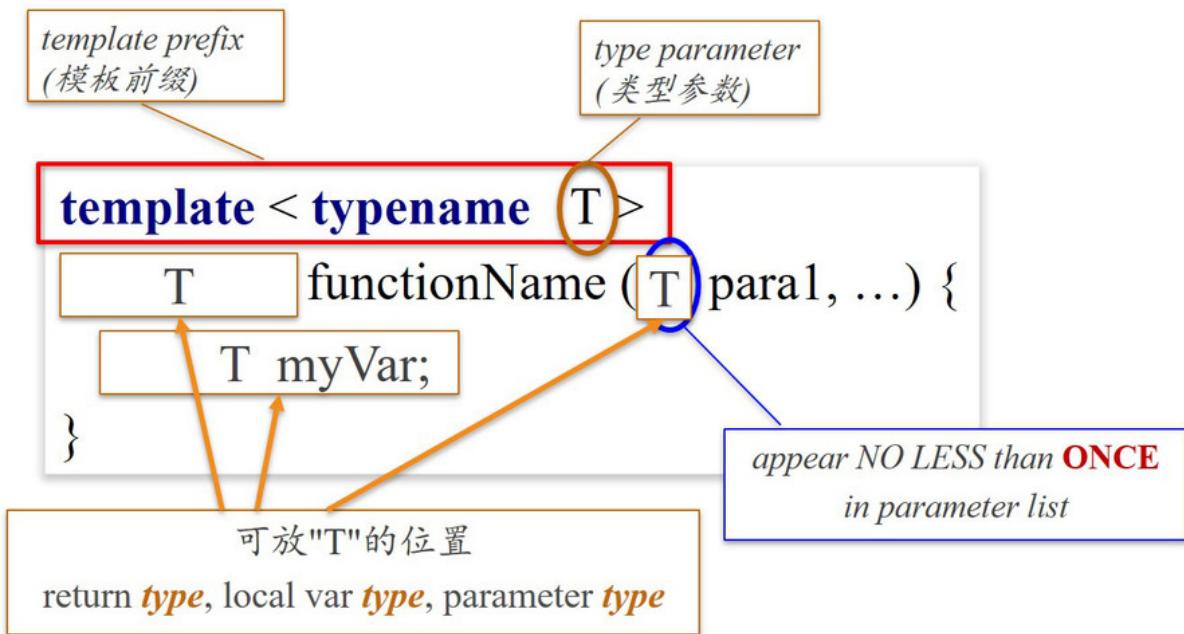
17.1 函数模板(Function Template)

1.Function template (函数模板)

C++引入了带有泛型的函数模板

1.1. How to specify a type parameter? (如何声明类型参数)?

- 1 <typename T> : 描述性强，较好
- 2 <class T> : 易与类声明混淆，不推荐
- 3 <T> : 不推荐



2. Multiple type parameters (多个类型参数)

若函数模板有多于一个类型参数该怎么处理?

应该类型参数间用逗号分隔

```

1 template < typename T, typename S >
2
3 auto add (T x1, S x2) { //C++14
4
5     return (x1 + x2);
6
7 }
```

2.2. 编码规范

Names representing template types should be a single uppercase letter.

用于表示模板类型的名字应该使用一个大写字母

例如:

```

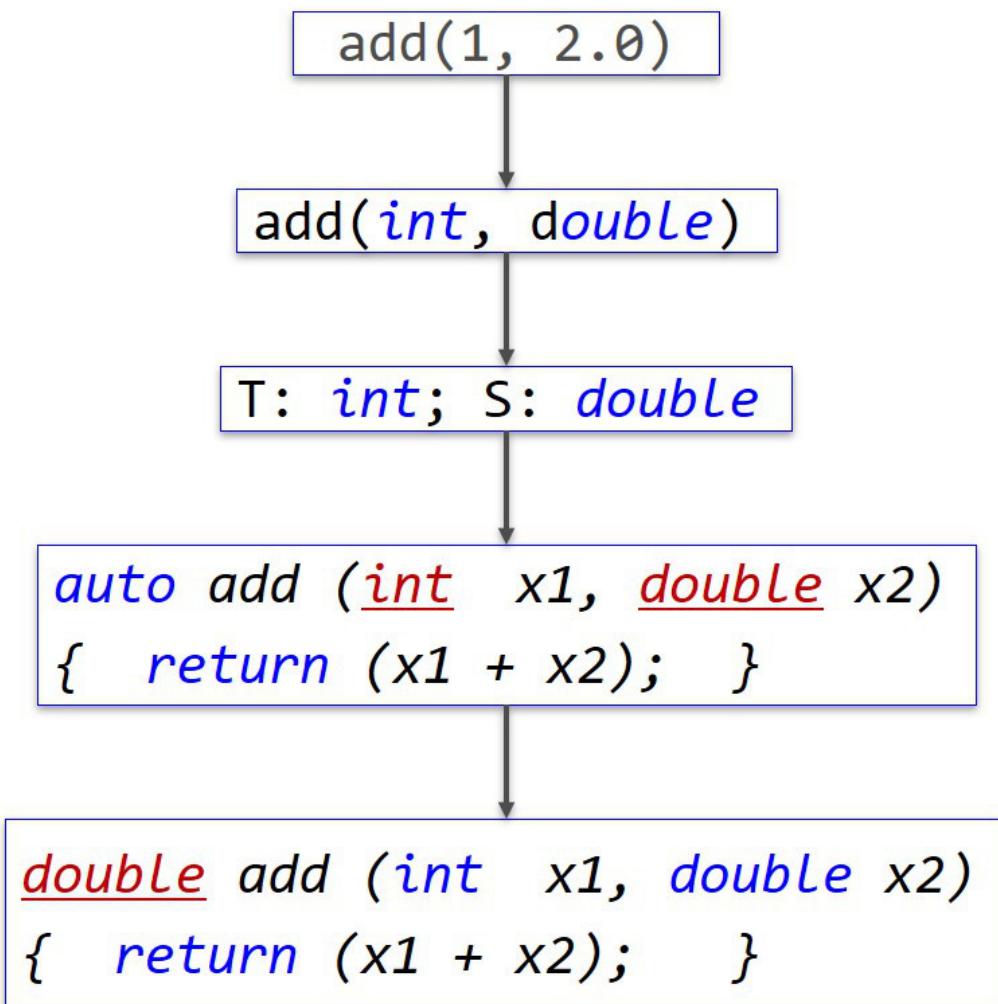
1 template<class T> ...
2
3 template<class C, class D> ...
```

3. Using Function Template (使用函数模板)

```

1 template <typename T, typename S>
2
3 auto add (T x1, S x2) { //C++14
4
5     return (x1 + x2);
6
7 }
8
9 int main () {
10     auto y = add ( 1, 2.0 );
11     return 0;
12 }
```

编译器根据函数调用的实参，生成函数模板的实例：



17.2 函数模板实例化

1. 什么是函数模板实例化

1.1. 实例化：

函数模板只是蓝图，本身不是类型、函数。编译器扫描代码，遇到模板定义时，并不立即产生代码。确定模板实参后，编译器生成实际函数代码

1.2. 两种实例化方法 (确定模板实参的方法)

Explicit instantiation (显式实例化)

Implicit instantiation (隐式实例化)

2. Explicit instantiation (显式实例化)

强制某些函数实例化，可出现于程序中模板定义后的任何位置。

```
1 template < typename T >
2 void f(T s){
3     std::cout << s << '\n';
4 }
5
6 template void f<double>(double); // 实例化，编译器生成代码
7 // void f(*double* s) { // *T*: *double*
8
9 //     std::cout << s << '\n';
10
11 // }
12
13 template void f<>(char); // 实例化 f<char>(char)，推导出模板实参
14 template void f(int); // 实例化 f<int>(int)，推导出模板实参
```

3. Implicit instantiation (隐式实例化)

编译器查看函数调用，推断模板实参，实现隐式实例化。

```
1 #include <iostream>
2
3 template<typename T>
4
5 void f(T s) {
6
7     std::cout << s << '**\n**';
8
9 }
10
11 int main(){
12 }
```

```
13  f<double>(1); // 实例化并调用 f<double>(double)
14
15  f<>('a'); // 实例化并调用 f<char>(char)
16
17  f(7); // 实例化并调用 f<int>(int)
18
19  void (*ptr)(std::string) = f; // 实例化 f<string>(string)
20
21 }
```

4. Instantiated function/class (实例函数/实例类)

4.1. C++11 (Section:14.7)

A function instantiated from a function template is called an ***instantiated function***. A class instantiated from a class template is called an ***instantiated class***.(由函数模板实例化得到的函数叫做“实例函数”，由类模板实例化得到的类叫做“实例类”)

非正规称呼:template function / template class

17.3 类模板

1. Class Template (类模板)

1.1 类模板是将类中某些类型变为泛型，从而定义一个模板

1.2. Generic types of class members (类成员的泛型)

Data field member(数据域成员) : can be of a generic data ***type*** (可以成为泛型数据)

Function member(函数成员): return ***type***, parameter ***type***, local var ***type*** (返回值类型、参数类型、局部变量都可以成为泛型)

2. Class Templates

StackOfIntegers	Stack<T>
-elements: int[100]	-elements: T[100]
-size: int	-size: int
+StackOfIntegers()	+Stack()
+empty(): bool	+empty(): bool
+peek(): int	+peek(): T
+push(value: int): int	+push(value: T): T
+pop(): int	+pop(): T
+getSize(): int	+getSize(): int

3. Syntax for class template (类模板的语法)

```

1  template<typename T>
2  class Stack{
3  public:
4      Stack();
5      bool empty();
6      T peek();
7      T push(T value);
8      T pop();
9      int getSize();
10
11 private:
12     T elements[100];
13     int size;
14 };
15 //模板前面放
16
17 //类型名后跟
18 template<typename T>
19 bool Stack<T>::empty()
20 {

```

```
21     return (size ==0);
22 }
23 //对比non-template class
24 // int Stack::peek()
25 template<typename T>
26 T Stack<T>::peek()
27 {
28     return elements[size - 1];
29 }
```

创建使用类模板时，类的声明和实现要放在同一个文件中，不能分离。

17.4 类模板实例化

1.显式实例化

```
1 | template class Stack<int>; // 将类模板实例化为一个处理int类型的Stack类
```

2.隐式实例化

```
1 | Stack<char> charStack; // 先实例化一个CharStack类(名字由编译器按规则
2 | // 生成)
3 | // class CharStack { ... char
4 | elements[100]; ... };
5 | // 然后用 CharStack charStack; 创建一个对象
6 |
7 | Stack<int> intStack; // 实例化一个IntStack类
8 | // class IntStack{ ... int elements[100]; ...
9 | };
10 | // 然后用 IntStack intStack; 创建一个对象
11 |
12 |
13 |
14 |
15 | vector intVector{1, 2, 3}; // C++17, 模板类型参数根据初始化语句自动推
16 | // 导
17 | // 实例化为 vector<int>
```

17.5 默认类型与非类型参数

1. Default type (默认类型)

1.1. Review: default parameter value (复习：参数的默认值)

```
1 | int circle (int x, int y, int r=100);
```

1.2. Default type parameter (默认类型参数)

可以为类模板的类型参数指定一个默认类型

例如，指定泛型Stack的默认类型参数为 int

```
1 | template<typename T = int>
2 | class Stack{
3 | ...
4 | };
```

1.3. 用默认类型定义一个对象

```
1 | //a stack for int values
2 |
3 | stack<> stack;
```

函数模板不能使用默认类型参数。

2. Non-type Parameters (非类型参数)

在模板前缀中使用非类型参数，实例化模板时，非类型实参应该是对象

```
1 | template<typename T, int capacity>
2 | class Stack{
3 | ...
4 | private:
5 |     T elements[capacity];
6 |     int size;
7 | };
8 |
9 | Stack<char, 100> charStack;
10
11
```

```
12
13 //对象作为非类型参数
14 template<typename T, color c>
15 class Label{
16     ....
17 };
18
19 color color(0, 0, 255);
20 Label<char, color> label;
```

17.6 模板与继承

普通类可以从类模板实例继承

模板可以从普通类继承

类模板可以继承类模板

