# Chapter 1. 开始

## 1.1. 编写简单的 C++程序

## 1.2. 初识输入输出

1、C++并未定义任何输入输出(IO)语句，而是用一个全面的标准库来提供 IO 机制(iostream 库)
2、cin 是 istream 类型的对象，也被称为标准输入
3、cout 是 ostream 类型的对象，也被称为标准输出
4、cerr 是 ostream 类型的对象，用于输出警告和错误信息，也被称为标准错误
5、clog 是 ostream 类型的对象，用来输出程序运行时的一般信息
6、endl：被称为操纵符的特殊值，**作用是结束当前行，并将于设备关联的缓冲区中的内容刷到设备中。缓冲刷新操作可以保证到目前为止程序所产生的所有输出都真正写入输出流中，而不是停留在内存中等待写入流**
7、运算符

- <<：输出运算符
- >>：输入运算符

8、让 while(cin>>a)循环终止，按 ctrl+z+enter。(让输入失败就行了，也就是让流返回参数 fail 置位)

## 1.3. 注释简介

1、注释种类：

- 单行注释：//
- 界定符对注释：/*　　something　　*/

2、界定符注释：

- 风格：注释内每行都以一个星号开头，从而指出整个范围都是多行注释的一部分
- 注释界定符不能嵌套

## 1.4. 控制流

- while 语句
- for 语句
- if 语句

## 1.5. 类简介

1、类机制是 C++最终的特性之一，C++最初的设计焦点就是能定义使用像内置类型一样自然的类类型
2、成员函数是定义为类的一部分的函数，有时也被称为方法

# Chapter 2. 变量和基本类型

## 2.1. 基本内置类型

### 2.1.1. 算数类型

| 类型 | 含义 | 最小尺寸 |
|---|---|---|
| bool | 布尔类型 | 未定义 |
| char | 字符 | 8 位 |
| wchar_t | 宽字符 | 16 位 |
| char16_t | Unicode 字符 | 16 位 |
| char32_t | Unicode 字符 | 32 位 |
| short | 短整型 | 16 位 |
| int | 整型 | 16 位 |
| long | 长整型 | 32 位 |
| long long | 长整型 | 64 位 |
| float | 单精度浮点数 | 6 位有效数字 |
| double | 双精度浮点数 | 10 位有效数字 |
| long double | 扩展精度浮点数 | 10 位有效数字 |

1、带符号类型和无符号类型

- int、short、long、long long 都是带符号的，在其前面添加 unsigned 就可以得到无符号类型
- 字符类型被分为三种：char、signed char、unsigned char，但表现形式只有两种，类型 char 实际上回表现为 signed char 和 unsigned char 的一种，具体有编译器决定

### 2.1.2. 类型转换

- 非布尔的算数值 赋值给 布尔类型：0 代表 false，其余代表 true
- 布尔值 赋值给 非布尔类型：false 赋值为 0，true 赋值为 1
- 浮点数 赋值给 整数类型：仅保留浮点数中小数点之前部分
- 整数 赋值给 浮点类型：小数部分记为 0，若整数超过浮点数的容量，精度会损失
- **赋值给 无符号类型一个超出它表示范围的值：初始值对无符号类型表示数值总数取模后的数**
- **赋值给 带符号类型一个超出它表示范围内的值：结果未定义**

### 2.1.3. 字面值常量

1、整数和浮点型字面值

- 0 开头代表八进制
- 0x\0X 开头达标 16 进制
- 整型字面值具体的数据类型由它的值和符号决定，默认情况下，十进制字面值是带符号数，八进制和十六进制字面值既可能是带符号的也可能是无符号的

2、字符和字符串字面值

- char 型字面值：由单引号括起来的一个字符
- 字符串型字面值：双引号括起来的零个或多个字符

- 字符串字面值的类型实际上是由常量字符构成的数组(至少包含'\0')
- **字符串字面值的类型是：const char[] 大多数情况下退化为 const char***

3、转义序列

| | |
|---|---|
| 换行符 | \n |
| 横向制表符 | \t |
| 纵向制表符 | \v |
| 退格符 | \b |
| 报警符(响铃符) | \a |
| 双引号 | \" |
| 反斜线 | \\(写路径时) |
| 问号 | \? |
| 单引号 | \' |
| 回车符 | \r |
| 进纸符 | \f |

4、指定字面值类型

5、布尔值字面值：

- true
- false

6、指针字面值

- nullptr

## 2.2. 变量

1、变量提供一个具名的，可供程序操作的存储空间，C++中每个变量都有数据类型，数据类型决定变量所占的空间大小和布局方式，该空间能存储的值得范围，以及变量能参与的运算符

### 2.2.1. 变量的定义

1、基本形式：

- 类型说明符
- 一个或多个变量名组成的列表
- 变量名以逗号分割
- 最后以分号结束

2、初始值：

- 对象在创建时获得了一个特定的值
- **初始化不是赋值，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除，而以一个新值来替代。**

3、列表初始化：

- 用花括号来初始化变量
- 如果使用列表初始化切初始值存在丢失信息的风险(类型转换)，编译器将报错

4、默认初始化：

- 如果定义变量时没有指定初始值，则变量被默认初始化
- 对于内置类型：

- 定义于任何函数体之外的内置类型默认初始化为 0
- 定义于函数体内的内置类型将不执行默认初始化，值任意

### 2.2.2. 变量声明和定义的关系

1、为了允许把程序拆分成多个逻辑部分来编写，C++语言支持分离式编译机制，该机制允许程序分割为若干个文件，每个文件可被独立编译

2、为了支持分离式编译，C++语言将声明和定义区分开。

- 声明：使得名字为程序所知，一个文件如果想使用别处定义的名字则必须包含对那个名字的声明。
- 定义：负责创建与名字关联的实体，申请存储空间，可能还赋初值

3、想声明而非定义一个变量，在变量前面添加关键字 extern

4、任何包含显示初始化的声明即成为定义(即便有 extern 也不行)

5、变量只能被定义一次，但可以多次声明

6、如果想在多个文件中使用同一变量，就必须将声明和定义分离。变量的定义必须出现在且只能出现在一个文件中，而其他用到该变量的文件必须对其进行声明，却决不能重复定义

### 2.2.3. 标识符

1、C++的标识符由字母、数字和下划线组成，必须以字母和下划线开头

2、对大小写敏感

3、注意规范

### 2.2.4. 名字的作用域

1、作用域是程序的一部分，其中的名字有其特定的含义

2、C++大多数作用域以花括号分割

3、名字的有效区域始于名字的声明语句，一声明语句所在的作用域末端作为结束

4、建议在第一次使用变量时再定义它

5、嵌套的作用域

## 2.3. 符合类型

### 2.3.1. 引用

1、引用为对象起了另一个名字，但引用不是对象(引用的地址与原变量的地址是一样的，因为引用就是这个对象的别名，原名就是原变量名)，因此引用本身不能用 const 限定符(即引用的 const 属性不可能是顶层的)

- const int&i
- int const&i//与上面的语句相同，都是底层的 const 属性

2、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，无法令引用重新绑定到另一个对象，因此引用必须初始化(与 Java 引用不同！)

3、int & i=j;

4、各类引用绑定的限制

- 非常量引用只能绑定该类型的非常量 <1>

- 常量引用可以绑定该类型的所有类型：常量左值，非常量左值，常量右值(包括字面值)，非常量右值 <4>
- 右值引用可以绑定该类型的右值非常量以及右值常量(字面值类型) <2>
- 引用必须赋初值，引用不是对象(类内引用不必含有初始值，初始化在构造函数中实现)
- 初始化常量引用：可以用任意表达式为其赋值，只要该表达式的结果可

5、C++中任何数据类型都可以有引用(包括内置类型)，但是 Java 只有类类型才有引用

### 2.3.2. 指针

1、指针本身就是一个对象，允许对指针的拷贝和赋值
2、指针可以在其生命周期内先后指向不同的对象
3、无需再定义指针时赋初值
4、利用*p 访问对象
5、int *p=&i;
6、void*指针

- 用于存放任意对象的地址(有点类似于 Java 的 Object 引用哦)
- 但在 C++中，这种存放任意对象的地址没有多大的用处，因为不能直接操作 void* 指针所指的对象
- 以 void*的视角来看内存空间也仅仅就是内存空间，没办法访问内存空间中所存的对象
- 可以通过类型转换将其转换为正确的类型后，加以使用(Java 中的 Object 引用也可以向下转型)

## 2.4. const 限定符

1、const 对象一旦创建后就不能再改变
2、默认状态下，const 对象仅在文件内有效

- 编译器将在编译过程中把用到该变量的地方都替代成对应的值，也就是说，编译器会找到代码中所有用到该 const 变量的地方，然后将其替换成定义的值
- 为了执行上述替换，编译器必须知道变量的初始值，如果程序包含多个文件，则每个用了 const 对象的文件都必须能访问到它的初始值才行。要做到这一点，就必须在每一个用到该变量的文件中都对它有定义(将定义该 const 变量的语句放在头文件中，然后用到该变量的源文件包含头文件即可)，为了支持这一用法，同时避免对同一变量的重复定义，默认情况下 const 被设定为尽在文件内有效(const 的全局变量，其实只是在每个文件中都定义了一边而已)
- 有时候出现这样的情况：const 变量的初始值不是一个常量表达式，但又确实有必要在文件间共享。这种情况下，我们不希望编译器为每个文件生成独立的变量，相反，我们想让这类 const 对象像其他对象一样工作。即：在一个文件中定义 const，在多个文件中声明并使用它，无论声明还是定义都添加 extern 关键字
  - .h 文件中：extern const int a;
  - .cpp 文件中：extern const int a=f();

3、*如果想在多个文件之间共享 const 对象，必须在变量定义之前添加 extern*

4、**关于类内 const 成员的初始化：**

- 构造函数中显式初始化：在初始化部分进行初始化，而不能在函数体内初始化
- 如果没有显式初始化，就调用定义时的初始值进行初始化

### 2.4.1. const 的引用

1、例子：

- const int ci=111;
- const int &ri=ci;

2、术语：常量引用

- 常量引用是对 const 的引用
- **严格来说，并不存在常量引用，因为引用不是一个对象，所以我们没法让引用本身恒定不变**
- **事实上，C++不允许随意改变引用绑定的对象，这样理解引用又都是常量**
- **引用的对象是常量还是非常量可以决定其能参与的操作，却无论如何都不会影响到引用和对象的绑定关系本身**

3、常量引用可以绑定非常量和常量(两者均包括左右值)

4、非常量引用只能绑定非常量

### 2.4.2. 指针和 const

1、指向常量的指针：不能用于改变其所指对象的值

2、常量指针：将指针本身设定为常量，一旦初始化，不能改变(必须初始化)

### 2.4.3. 顶层底层 const

1、顶层 const 例子

- int const *p=&i;
- const int i=5;

2、顶层的 const 可以表示任意的对象是常量(包括指针，不包括引用，因为引用本身不是对象，没法指定顶层的 const 属性)

3、**只有指针的 const 属性既可以是顶层又可以是底层**

4、**引用只有底层 const 属性**

### 2.4.4. constexpr 和常量表达式

1、常量表达式：指值不会改变且在编译过程就能得到计结果的表达式，**注意常量表达式与 const 变量的区别，只有用编译时可计算的对为 const 变量进行初始化，该 const 对象才能成为常量表达式**

- const int max_files=20;
- const int limit=max_files+1;
- const int a=f();**//WRONG**

2、constexpr 变量

- 允许将变量声明为 constexpr 类型以便由编译器来验证变量的值是否是一个常量表达式
- 声明为 constexpr 的变量一定是一个常量，而且必须用常量表达式初始化
- 不能用普通函数作为 constexpr 变量的初始化，**但是可以用特殊的 contexpr**

函数(足够简单，以使得编译时就可以计算结果)来初始化 constexpr 变量。

#### 2.4.5. 字面值类型

1、初始化声明为 constexpr 时用到的类型就是字面值类型

2、指针和引用都能定义成 constexpr，但是他们的初始值受到严格的限制

3、一个 constexpr 指针的初始值必须是 nullptr 或 0，或者是存储于某个固定地址中的对象

4、函数体内定义的变量一般来说并非存放在固定地址，因此 constexpr 指针不能指向这类对象；相反，定义于所有函数之外的对象其地址固定不变，能用来初始化 constexpr 指针

5、**constexpr 如果定义了一个指针，那么该限定符只与指针有效，与指针所指的对象无关(与 const 不同，const 的位置决定了 const 是顶层还是底层)**

- const int *p=nullptr;  //指向整型常量的指针
- constexpr int *q=nullptr;  //指向整数的常量指针

### 2.5. 处理类型

#### 2.5.1. 类型别名

1、方法 1：typedef

- typedef double wages; //wages 是 double 的同义词
- typedef double base,*p;//base 是 double 的同义词，p 是 double*的同义词

2、方法 2：using

- using SI=Sales_Item;//SI 是 Sales_Item 的同义词

#### 2.5.2. auto 类型说明符

1、auto 能让编译器替我们分析表达式所属的类型

2、**auto 会忽略顶层 const，保留底层的 const，但是当设置一个类型为 auto 的引用时，初始值中的顶层常量属性仍然保留**

#### 2.5.3. decltype 类型指示符

1、de'c'ltype 会保留变量的所有类型信息(包括顶层 const 和引用在内)

2、如果表达式的内容是解引用操作，得到的将是引用类型

- int i=42;
- int *p=&i;
- decltype(*p)：得到 int&

3、decltype((c))会得到 c 的引用类型(无论 c 本身是不是引用)

### 2.6. 自定义数据结构

#### 2.6.1. 预处理器概述

1、结构

- #ifndef ABCDEF //当 ABCDEF 未定义时为真
- #define ABCDEF //定义 ABCDEF
- //something
- #endif

2、**整个程序中的预处理变量包括头文件保护符必须唯一，通常的做法是基于头文件的类的名字来构建保护符的名字，以确保其唯一性。为了避免与程序中其他**

实体名字发生冲突，一般把预处理变量的名字全部大写

# Chapter 3. 字符串、向量和数组

## 3.1. 命名空间的 using 声明

1、形式：using namespace::name;//注意 using namespace std;是 using 指示
- using std::cin;

2、头文件中不要包含 using 声明

## 3.2. 标准库类型 string

1、string 的定义在命名空间 std 中。
2、初始化方式：
- string s1;
- string s2(s1);
- string s2=s1;
- string s3("value");
- string s3="value";
- string s4(n,'c')

3、操作
- os<<s;//返回 os
- in>>s;//返回 in
- getline(is,s);//返回 is
- s.empty();
- s.size();//Java 中的 String 是 String.length()
- s[n];
- s1==s2//java 中只能用 String.equals 来比较

4、在执行读取操作时，string 对象会自动忽略开头的空白(空格、换行符、制表符等，但不包括'\0')，真正的第一个字符开始读起，直到遇见下一个空白
5、可以用范围 for 语句处理 string 中的每个字符

## 3.3. 标准库类型 vector

1、定义和初始化 vector
- vector<T> v1
- vector<T> v2(v1)
- vector<T> v2=v2
- vector<T> v3(n,val)
- vector<T> v4(n)
- vector<T> v5{a,b,c...}
- vector<T> v5={a,b,c...}

2、添加元素
- v.push_back(<something>);

3、在使用范围 for 语句的时候不能改变容器的大小
4、支持下标访问

## 3.4. 迭代器介绍

1、遍历容器的通用机制
2、严格来说 string 不算容器，但是 string 支持与容器类型类似的操作，string 也支持迭代器
3、迭代器类型

- 对于 vector<T>来说
- vector<T>::iterator
- vector<T>::const_iterator

4、在用迭代器遍历容器的时候，不能改变容器的大小，否则迭代器会失效

## 3.5. 数组

1、数组的声明形式如 a[d],a 是名字 d 是维度，维度说明了数组中元素的个数，必须大于 0
2、数组维度必须是一个常量表达式(常量表达式，常量表达式初始化的 const，constexpr)
3、***数组不允许拷贝和赋值***
4、C++中，指针和数组有非常紧密的联系，使用数组的时候，编译器一般会把它转换成指针。
4、数组还有一个特性，在很多用到数组名字的地方，编译器都会自动地将其替换成为一个指向数组首元素的指针。一个典型的例子就是使用数组下标(可以是负数，天啊，为毛搞这么弯弯绕)的时候
5、标准库函数 begin 和 end

- begin(a);//获取数组 a 的首元素指针
- end(a);//获取数组 a 的尾后元素指针

6、数组参数传递的几个例子

```
void f1(int a[]){
    cout<<sizeof(a)/sizeof(a[0])<<endl;
}
void f2(int a[10]){
    cout<<sizeof(a)/sizeof(a[0])<<endl;
}
void f3(int (&a)[10]){
    cout<<sizeof(a)/sizeof(a[0])<<endl;
}
int main(){
    int a[10]={0,1,2,3,4,5,6,7,8,9};
    f1(a);    //输出 1
    f2(a);    //输出 1
    f3(a);    //输出 10
}
```

- 当形参类型为数组类型时，**传递进来的是数组的首地址，并且会丢失数组长度信息，即使用 sizeof 也无法获取**，因此要在传入一个参数表明数组的维度。另外指定数组的引用类型时，必须指定数组的纬度

# Chapter 4. 表达式

1、表达式由一个或多个运算对象组成，对表达式求值将得到一个结果，字面值和变量是最简单的表达式，其结果就是字面值和变量的值。

## 4.1. 基础

### 4. 1. 1. 基本概念

1、C++定义了一元运算符和二元运算符

2、C++可以重载运算符，可以自定义运算对象的类型和返回值的类型，但是运算对象的个数以及运算符的优先级和结合律是无法改变的

3、C++表达式不是右值就是左值

- 右值：当一个对象被当做右值时，用的是对象的值
- 左值：当一个对象被当做左值时，用的是对象的身份

4、使用关键字 decltype 求表达式的类型(注意，该表达式不包括是单个的变量)

- **如果表达式的结果是左值，decltype 后会得到该类型的引用(也解释了，decltype((i))将得到 i 的引用类型，因为显式把 i 当做非变量表达式了)**
- 如果表达式的结果是右值，decltype 后会得到该类型

## 4.2. 算数运算符

1、类型：

- + expr
- - expr
- expr * expr
- expr / expr
- expr % expr
- expr + expr
- expr - expr

## 4.3. 逻辑和关系运算符

1、类型：

- ! expr
- expr < expr
- expr <= expr
- expr > expr
- expr >= expr
- expr == expr
- expr != expr
- expr && expr
- expr || expr

## 4.4. 赋值运算符

1、赋值运算符的左侧对象必须是一个可修改的左值

2、如果赋值运算符左右两个运算对象类型不同，则右侧运算对象将转换成左侧运算对象的类型
3、各类符合赋值运算符

## 4.5. 递增和递减运算符

1、可用于迭代器，有些迭代器不支持算数运算，但是支持递增递减运算
2、前置版本(++i)返回递增后的左值
3、后置版本(i++)返回递增前初始值的右值副本

## 4.6. 成员访问运算符

1、写法：
- ptr->men
- (*ptr).men

## 4.7. 条件运算符

1、写法
- cond?expr1:expr2

## 4.8. 位运算符

1、类型：
- ~ expr
- expr << expr
- expr >> expr
- expr & expr
- expr ^ expr
- expr | expr

## 4.9. sizeof 运算符

1、sizeof 运算符返回一条表达式或者一个类型名字所占的字节数，满足右结合律
2、返回类型是 size_t
3、计算数组的大小：sizeof(ary)/sizeof(*ary) **or** sizeof(ary)/sizeof(ary[0])

## 4.10. 逗号运算符

1、首先求左侧表达式的值，然后将求值结果丢掉，都好运算符真正的结果是右侧表达式的值

## 4.11. 类型转换

1、隐式转换：
- 在大多数表达式中，比 int 类型小的整型首先提升为较大的整数类型
- 在条件中，非布尔值转换成布尔类型
- 初始化中，初始值转换成变量的类型
- 在赋值语句中，右侧运算对象转换成左侧运算对象的类型
- 如果算数运算或关系运算的运算对象有多重类型，需要转换成同一种类型

- 函数调用时也会发生类型转换(Chapter6)

2、算数转换：

- 运算符的运算对象将转换成(所有运算对象中)最宽的类型

3、其他隐式转换：

- 数组转为指针：在大多数表达式中，数组自动转换成指向数组首元素的指针(当被用于 deltype 关键字的参数，或者作为取址符&、sizeof、typeid 以及用引用来初始化数组时，该转换不会发生)

4、显式转换：

- static_cast：任何具有明确意义的类型转换，只要不包含底层 const，都可以使用 static_cast
  - 另外可以将 void*指针转换成原来的类型，但必须指定正确的类型，否则将会产生未定义的后果
- const_cast：只能改变运算对象的底层 const
  - 一旦去掉某个对象的 const 属性，编译器就不再阻止我们对该对象进行写操作了，这样是合法的
- C 风格的类型转换：int i= (int) j;　(Java 属于这种)

## 4.12. 运算符优先级表

1、口诀：

- 作用域
- 下标，成员访问
- 后置递增递减
- 单目运算符
- .*　->*
- 乘除模
- 加减
- 移位
- 大小于关系
- 相等不等关系
- 位于
- 位异或
- 位或
- 逻辑与
- 逻辑或
- 条件
- 赋值
- 符合赋值
- 异常
- 逗号

# Chapter 5. 语句

## 5.1. 简单语句

1、C++大多数语句都以分号结束，一个表达式末尾加上分号就变成了表达式语句，表达式语句的作用是执行表达式并丢弃结果。
2、空语句
- ;

3、复合语句：用花括号括起来的语句和声明的序列，复合语句也被称为块

## 5.2. 语句作用域

1、可以在 if、switch、while、for 语句的控制结构内定义变量，定义在控制结构内的变量只在相应语句的内部可见

## 5.3. 条件语句

1、if 语句
2、switch 语句

## 5.4. 迭代语句

1、while 语句
2、for 语句
- for 循环定义部分，只能有一个声明语句，只能出现一个类型

3、范围 for 语句
4、do while 语句

## 5.5. 跳转语句

1、break 语句：用于终止离它最近的 while、do while、for 或 switch
2、continue 语句：终止最近循环中当前迭代并立即开始下一次迭代
3、goto 语句：别用！

## 5.6. try 语句块和异常处理

1、格式

```
try{
        program-statements
}catch (exception-declaration){
}
...
```

2、标准异常:<stdexcept>
- exception：最常见的问题
- runtime_error：运行时才能检测出的问题
- range_error：运行时错误，生成的结果超出了有意义的值或范围
- overflow_error：运行时错误：计算上溢
- underflow_error：运行时错误：计算下溢
- logic_error：程序逻辑错误

- domain_error：逻辑错误：参数对应的结果值不存在
- invalid_argument：逻辑错误：无效参数
- length_error：逻辑错误：试图创建一个超出该类型最大长度的对象
- out_of_range：逻辑错误：使用一个超出范围的值

3、异常类型只定义了一个名为 what 的成员函数，用于返回一个指向 C 风格字符串的 const char*，用于提供关于异常的一些文本信息。

# Chapter 6. 函数

1、函数是一个命名了的代码块，通过调用函数执行相应的代码，函数可以有 0 个或多个参数，而且(通常)会产生一个结果，可以重载。

## 6.1. 函数基础

1、组成部分
- 返回类型
- 函数名字
- 0 个或多个形参组成的列表
- 函数体

2、我们通过调用运算符来执行函数，调用运算符的形式是一对圆括号，它作用于一个表达式，该表达式是函数或者指向函数的指针。圆括号内是一个用逗号隔开的实参列表。

3、形参列表可以为空，但不能省略

4、**局部对象**
- 形参和函数体内部定义的变量统称为局部变量
- 仅在函数内部可见，或隐藏外层作用域的同名的所有声明

5、**局部静态对象**
- 在函数体内用 static 关键字定义
- 局部静态对象在程序的执行路径第一次经过对象定义语句时初始化，到程序终止才被销毁。

6、函数的声明
- 函数的声明必须在函数的使用之前
- 函数可以多次声明，但最多定义一次
- 建议在头文件中声明，在源文件中定义

7、**分离式编译：**
- 大多数编译器提供了分离式编译每个文件的机制，这一过程通常会产生一个后缀名是.obj(Windows)或.o(UNIX)的文件，或追命的含义是该文件包含对象代码。
- 接下来编译器负责把对象文件**链接**在一起形成可执行文件。

## 6.2. 参数传递

1、形参的类型决定了形参和实参的交互方式
- 引用类型：将形参绑定到对应的实参上
- 非引用类型：将实参的值拷贝后赋值给形参

2、传值参数(非引用类型的形参)：对形参的改动不会改变实参，**但是如果该形参是一个指针(传递时拷贝指针的值)，如果通过指针改变其指向的数据是会影响实参的(C++要达到实参形参为同一个对象(包括内置类型)，即不发生拷贝，最好传递引用)**

3、传引用参数：
- 避免拷贝
  - 拷贝大的类类型对象或者容器对象比较低效
  - 某些类类型不支持拷贝(IO 类型)

- **使用引用形参返回额外信息**
- 如果函数无需改变引用形参的值，最好将其声明为常量引用

4、定义或声明函数形参部分，两个类型一样的形参，必须写出两个类型！

5、**const 实参和形参：**

- 实参初始化形参时会自动忽略掉顶层 const 属性。(对于函数重载来说，是否有 const 都是一样的)
- 尽量使用常量引用

6、**数组的两个特殊性质(不允许拷贝，以及数组使用时通常会将其转换成指针)**对我们定义和使用作用在数组上的函数有影响：

- 我们无法以值传递的方式使用数组形参；因为数组会被转换成指针，所以当我们为函数传递一个数组时，实际上传递的是指向数组首元素的指针。
- **C++允许将变量定义成数组的引用，T (&t)[size]**

7、含有可变参数的函数(针对的是所有实参具有相同的类型的情况)：

- 可以传递一个名为 initializer_list 的标准库类型

8、关于命令参数

```
int main(int argc,char *argv[]){}
```

- 第二个参数 argv 是一个数组，它的元素是指向 C 风格字符串的指针
- 第一个形参 argc 表示数组中字符串的数量。因为第二个形参是数组，main 函数也可以定义成：　　**char*是字符串类型**

```
int main(int argc,char **argv){}
```

- 注意：可选的实参从 argv[1]开始　　argv[0]存放的是程序的名字

## 6.3. 返回类型和 return 语句

1、return 语句和终止当前正在执行的函数并将控制权返回到调用该函数的地方。

2、**返回一个值的方式和初始化一个变量或形参的方式完全一样：返回的值用于初始化调用点的一个临时量，该临时量就是函数调用的结果。**

3、返回引用或指针必须是传入的参数(该参数也以引用或者指针的方式传入)，因为不能返回局部对象的引用和指针

4、返回引用得到左值，其他返回类型都是右值。

5、可以返回胡括号包围的值得列表，用列表对返回的临时量进行初始化

6、返回数组指针

- 使用类型别名：
  - typedef int arrT[10];//首先将 int[10]用一个别名 arrT 来表示
  - arrT* func(int i);
- 直接声明一个返回数组指针的函数
  - Type (*function(parameter_list))[dimension];
- 使用尾置返回类型
  - auto function(parameter_list)->Type(*)[dimension];
- 使用 decltype
  - decltype(ary) *function(parameter_list);//注意 decltype 并不负责把数组类型转换成对应指针，因此需要添加上*

## 6.4. 函数重载

1、定义：如果同一个作用域内的几个函数名字相同，但是形参列表不同，我们称之为重载函数

2、对于重载函数，它们应该在形参数量或形参类型上有所不同。返回类型不作为判断是否为重载函数的类型，也就是说，函数的唯一性有三个方面：函数名字，形参数量，形参类型

3、顶层 const 不影响形参的类型

4、调用重载函数的可能结果：

- 编译器找到一个与实参最佳匹配的函数，并生成调用该函数的代码
- 找不到任何一个函数与调用的实参匹配，此时编译器发出无匹配的错误
- 有多与一个函数可以匹配，但是每一个都不是明显的最佳选择，将发生二义性调用的错误

5、重载与作用域：如果我们在内层作用域中声明名字，它将隐藏外层作用域中声明的同名实体，在不同的作用域中无法重载函数名

## 6.5. 特殊用途语言特性

1、默认实参：

- 我们可以为一个或多个形参定义默认值
- 一旦某个形参被赋予了默认值，它后面的所有形参都必须有默认值
- 在给定作用域中一个形参只能被赋予一次默认实参，换句话说，函数的后续声明只能为之前那些没有默认值的形参添加默认实参，而且该形参右侧的所有形参都必须有默认值
- 通常应该在函数声明中指定默认实参，并将该声明放在合适的头文件中。

2、内联函数：

- 内联函数可以避免函数调用的开销
- 在函数的返回类型前面加上关键字 **inline** 就可以声明称内联函数了
- 一般来说，内联机制用于优化规模较小、流程直接、频繁调用的函数，很多编译器都不支持内联递归

3、constexpr 函数：

- 是指能用于常量表达式的函数
- 函数的返回类型及所有形参的类型都得是字面值类型，而且函数体重必须有且只有一条 return 语句
- constexpr 被隐式指定为内联函数
- 我们允许 constexpr 返回值并非一个常量(如果函数的调用出现在需要常量表达式的上下文中，返回非常量会发出错误信息)

4、**把内联函数和 constexpr 函数放在头文件内**：和其他函数不一样，内联函数和 constexpr 函数可以在程序中多次定义，毕竟，编译器想展开函数仅有声明是不够的，还需要函数定义，对于某个给定的内联函数或者 constexpr 函数来说，它的多个定义必须完全一致，基于这个原因，内联函数和 constexpr 函数通常在头文件中。

5、assert 预处理宏：<cassert>

- 预处理宏其实是一个预处理变量，它的行为有点类似于内联函数。

- assert 宏使用一个表达式作为它的条件：assert(expr);
- 首先对 expr 求值，若表达式为假，assert 输出信息并终止执行的程序，若表达式为真，assert 什么也不做。
- 预处理名字由预处理器而非编译器管理，因此可以直接使用预处理名字而无需提供 using 声明
- 和预处理变量一样，宏名字在程序中必须唯一。含有 cassert 头文件的程序，不能再定义名为 assert 的量，函数或者其他实体。即使没有包含 cassert 最好也不要使用 assert 作为名字，因为很多头文件都包含了 cassert。

## 6.6. 函数匹配

1、步骤：

- 第一步：确定本次调用对应的重载函数集，集合中的函数称为候选函数
- 第二步：考察本次调用提供的实参，然后从候选函数中选出能被这组实参调用的函数，这些新选出的函数称为可行函数。
- 第三步：从可行函数中选择与本次调用最匹配的函数。(实参类型和行参类型越接近，它们匹配得越好)

2、**调用重载函数时疆良避免强制类型转换**

3、类型转换等级：

- 精确匹配：
  - 是参类型和行参类型相同
  - 实参从数组类型或函数类型转换成对应的指针类型
  - 像实参添加顶层 const 或者从实参中删除顶层 const
- 通过 const 转换实现的匹配
- 通过类型提升实现的匹配
- 通过算数类型转换或指针转换实现的匹配
- 通过类类型转换实现的匹配

## 6.7. 函数指针

1、函数指针指向的是函数而非对象，和其他指针一样，函数指针指向某种特定类型

2、声明一个函数指针

- ReturnType (*name) (parameter_list);

3、**当我们把函数名作为一个值使用时，该函数自动转换成指针**

4、

# Chapter 7. 类

1、类的基本思想是数据抽象和封装。
2、数据抽象是一种依赖于接口和实现分离来的编程(以及设计)技术
3、类的接口包括用户所能执行的操作
4、类的实现包括数据成员，负责接口实现的函数体以及定义各类所需的各种私有函数。
5、封装体现了类的接口和实现的分离，封装后的类隐藏了它的实现细节，也就是说，类的用户只能使用接口而无法访问实现部分

## 7.1. 定义抽象数据类型

1、this：
- 成员函数通过一个名为 this 的额外隐式参数来访问调用它的那个对象
- 任何自定义名为 this 的参数或变量都是非法的
- this 是一个常量指针

2、const 关键字：
- 默认情况下，this 的类型是指向类类型**非常量版本**的**常量指针，**意味着默认情况下，我们不能把 this 绑定到一个常量对象上，这一情况也使得我们不能再一个常量对象上调用普通的成员函数
- C++允许把 const 关键字放在成员函数的参数列表之后，此时表示 this 是一个指向常量的指针
- **常量对象以及常量对象的引用或指针都只能调用常量成员函数。**

3、类作用域和成员函数：
- 类本身就是一个作用域，类的成员函数的定义嵌套在类的作用域之内。
- 编译器分两步处理类：
  - 首先编译成员的声明
  - 然后才轮到成员函数体(如果有的话)
  - 因此成员函数体可以随意使用类中的其他成员而无需在意这些成员出现的次序
- 如果成员被声明为常量成员函数，且在类的外部定义，那么定义也必须在行参列表后加上 const 属性。

4、定义一个返回 this 对象的函数
- 返回该类型对象的引用
- 返回语句：return *this;

5、定义与类相关的非成员函数
- 通常把函数的声明和定义分开
- **一般与类的声明的在同一个头文件中，在这种方式下，用户使用接口的任何部分都只需要引入一个文件**

### 7.1.1. 构造函数

1、简介
- 每个类都定义了它的对象被初始化的方式，类通过一个或几个特殊的成员函数来控制其对象的初始化过程，这些函数叫做构造函数。
- 无论何时只要类的对象被创建，就会执行构造函数

2、属性：

- 构造函数没有返回类型，除此之外类似于其他函数
- 构造函数有参数列表和函数体
- 不同的构造函数必须在参数列表上有所区别(数量以及类型)
- **构造函数不能被声明成 const 的，因为直到构造函数完成初始化过程，对象才能真正取得其"常量"属性**
- **构造函数可以在 const 对象的构造过程中向其写值**

3、合成的默认构造函数

- 类通过一个特殊的构造函数来控制默认初始化的过程，这个函数叫做默认构造函数，默认构造函数无需任何实参
- **如果没有显式得定义构造函数，那么编译器将隐式地为我们定义一个默认构造函数**
- 编译器创建的默认构造函数又称为合成的默认构造函数，按照下列规则初始化数据成员：
  - 如果存在类内的初始值，用它来初始化成员
  - 否则默认初始化该成员

4、构造函数初始值列表：它负责为新创建的对象的一个或几个数据成员赋值。

## 7.2. 访问控制与封装

1、访问说明符：

- public：定义在 public 说明符之后的成员在整个程序内可被访问，public 成员定义类的接口(与外界交互)
- private：定义在 private 说明符之后的成员可以被类的成员函数访问，但是不能被使用该类的代码访问，private 封装了(隐藏了)类的实现细节。

2、class 与 struct 关键字

- **唯一区别：struct 和 class 的默认访问权限不同**

### 7.2.1. 友元

- 类可以允许其他类或者函数访问它的非公有成员，方法是令其他类或者函数成为它的友元。
- 把函数作为友元：只需要增加一条以 **friend** 关键字开头的函数声明语句即可。
- 一般来说，最好在类的定义开始或结束的位置，集中声明友元
- **友元的声明仅仅指定了访问权限，而非一个通常意义上的函数声明**，如果希望类的用户能够调用某个友元函数，那么我们就必须在友元声明之外再对函数进行一次声明
- 为了使友元对类的用户可见，我们通常把友元的声明和类本身放置在同一个头文件中

## 7.3. 类的其他特性

### 7.3.1. 类成员再探

1、定义类型成员

- 除了定义数据和函数成员之外，类还可以自定义某种类型在类型中的别名，用于隐藏其具体实现(该别名是 public 的，用户可以使用，但是用户无法知道，别名的真正类型)
- 可以使用 typedef：typedef typeName typeNewName;
- 也可以使用 using 声明：using typeNewName=typeName;
- 用来定义类型的成员必须先定义后使用，这与普通成员有区别

2、令成员作为内联函数
- 常有一些规模较小的函数适合于被声明成内联函数
- 定义在类内部的成员函数都是自动 inline 的
- 在声明和定义地方同时说明 inline 是合法的，但是最好只在类外部定义的地方说明 inline
- 与我们在头文件中定义 inline 函数的原因一样，inline 成员函数也应该与相应的类定义在同一个同文件中

3、可变数据成员
- 我们希望修改类的某个数据成员，即使是在一个 const 成员函数内，可以通过在变量的声明中加入 mutable 关键字来做到这一点
- 一个可变数据成员永远不会是 const 的，即使它是 const 对象的成员。因此，一个 const 成员函数可以改变一个可变成员的值

4、类数据成员的初始值：
- 类内初始值必须使用=的初始化形式或者用花括号括起来的直接初始化形式。

### 7.3.2. 返回*this 的成员函数

1、如果返回类型不是引用类型，那么 return *this;将返回*this 的副本
2、从 const 成员函数返回*this
- 一个 const 成员函数如果以引用的形式返回*this，那么它的返回类型将是常量引用。

3、基于 const 的重载
- 由于非常量版本的函数对于常量对象是不可用的，所以我们只能在一个常量对象上调用 const 成员函数
- 非常量对象可以调用 const 成员函数以及非 const 的成员函数，但是就重载匹配来说，非 const 的成员函数无疑是一个更好的匹配

### 7.3.3. 类类型

1、两个类只要名字不同，那他们就是完全不同的两个类型
2、类的声明：
- 就像可以把函数的声明和定义分开一样，我们也能仅声明类而暂时不定义它，这种声明被称为前向声明

- 在类的声明之后，定义之前，该类型是一个不完全的类型，此时该类的应用场景仅限以下情形：
  - **可以定义指向这种类型的指针或引用**
  - **可以声明(但是不能定义)以不完全类型作为参数或返回类型的函数**
- 对于一个类来说，我们创建它的对象之前，该类必须被定义过，而不能仅仅被声明。否则编译器就无法了解这样的对象需要多少存储空间。
- 类似，类也必须首先定义，才能用引用或指针访问其成员，因为在定义之前，编译器无法知道该类有哪些成员
- 直到类被定义之后，数据成员才能被声明称这种类型，换句话说，我们必须首先完成类的定义，然后编译器才能知道存储该数据成员需要多少空间，因为只有当类全部完成后，类才算被定义，**所以一个类的成员类型不能是该类自己**
- 一旦一个类的名字出现以后，它就被认为是声明过了，因此允许包含指向它自身类型的引用或指针
- **允许定义静态成员，其类型是该类型自己，当然也可以是引用或指针**

### 7.3.4. 友元再探

1、类之前的友元关系
- 如果一个类指定了友元类，则友元类的成员函数可以访问此类包括非公有成员在内的所有成员
- 友元关系不存在传递性
- 每个类负责控制自己的友元类或友元函数

2、重载函数和友元
- 如果一个类想把一组重载函数声明为它的友元，它需要对这组函数中的每一个分别声明

3、友元声明和作用域，见 P252 例子
- 类和非成员函数的声明不是必须放在他们的友元声明之前
- 当一个名字第一次出现在一个友元声明中，我们隐式地假定该名字在当前的作用域是可见的，然而，友元本身并不一定真的声明在当前作用域中

## 7.4. 类的作用域

1、每个类都会有自己的作用域，在类的作用域之外，普通的数据和函数成员只能由对象、引用、或者指针使用成员访问运算符来访问，对于类类型成员则使用作用域运算符访问

2、作用域和定义在类外部的成员：
- 一个类就是一个作用域的事实能够很好地解释为什么我们在类的外部定义成员函数时必须同时提供类名和函数名，因为在类的外部，成员的名字被隐藏了
- 一旦遇到了类名，定义的剩余部分就在类的作用域之外了，这里的剩余部分包括参数和函数体，我们可以直接使用类的其他成员而无需再次授权了

### 7.4.1. 名字查找与类的作用域

1、名字查找的过程

- 首先，在名字所在的块中寻找声明语句，只考虑在名字的使用之前出现的声明
- 如果没有找到，继续查找外层作用域
- 如果最终没有找到匹配的声明，则程序报错

2、对于定义在类内部的成员来说，名字查找的规则有所区别：

- 首先，编译成员的声明
- 直到类全部可见后才编译函数体
- 编译器处理完类中的全部声明后才会处理成员函数的定义

## 7.5. 再探构造函数

### 7.5.1. 构造函数初始值列表

1、构造函数的初始值有时必不可少

- 如果成员是 const 或者引用，必须将其初始化，**而我们初始化 const 的唯一机会就是通过构造函数初始化部分(：之后)**
- 当成员属于类类型切米有定义默认构造函数，也必须将这个成员初始化

2、最好令构造函数初始值的顺序与成员声明的顺序保持一致，而且如果可能的话，尽量避免使用某些成员初始化其他成员。

### 7.5.2. 委托构造函数

1、一个委托构造函数使用它所属类的其他构造函数执行它自己的初始化过程，或者说它把自己的一些(或者全部)职责委托给了其他构造函数。

2、成员初始值列表只有一个唯一的入口：就是类名本身**(与 Java 不同，Java 的构造器没有初始化部分，因此在函数体中调用其他构造器即可)**

```
class A{
        int value;
        A(int i):value(i){}
        A():A(5){}//委托构造函数，注意对其他构造函数的调用出现在初始化部分
};
```

### 7.5.3. 默认构造函数的作用

1、默认初始化的情况：

- 我们在块作用域内不适用任何初始值定义一个非静态变量
- 当一个类本身含有类类型的成员，且使用合成的默认构造函数时
- 当类类型的成员没有在构造函数初始值列表显式得初始化时

2、值初始化的情况：

- 在数组初始化的过程中，如果我们提供的初始值数量少于数组的大小时
- 当我们不用初始值定义一个局部静态变量时
- **当我们通过书写形如 T()的表达式显示的请求初始化时**

### 7.5.4. 隐式的类类型转换

1、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转换的规则

2、只允许一步类类型转换

3、通过将函数声明为 explicit 来抑制隐式转换

4、explicit 构造函数只能用于直接初始化

### 7.5.5. 聚合类

1、聚合类使得用户可以直接访问成员，并且具有特殊的初始化语法

- 我们可以提供一个花括号括起来的成员初始值列表，并用它来初始化聚合类的数据成员，初始值的顺序必须与声明的顺序一致
- 如果初始值列表中的元素个数少于类的成员数量，则靠后的成员被值初始化

2、特点：

- 所有成员都是 public 的
- 没有定义任何构造函数
- 没有类内初始值
- 没有基类

3、缺点：

- 要求类的所有成员都是 public 的
- 将正确初始化每个对象的每个成员的重任交给了类的用户
- 添加或删除一个成员后，所有的初始化语句都需要更新

### 7.5.6. 字面值常量类

1、字面值类型：

- 算数类型
- 引用和指针
- 满足一定条件的某些类类型

2、字面值常量类：

- 数据成员都是字面值类型的聚合类是字面值常量类
- 满足以下条件的是字面值常量类：
  - 数据成员必须是字面值类型
  - 类必须至少含有一个 constexpr 构造函数
  - 如果一个数据成员含有类内初始值，则内置类型成员的初始值必须是一条常量表达式，如果成员为某种类类型，则初始值必须使用成员自己的 constexpr 构造函数
  - 类必须使用西沟函数的默认定义

## 7.6. 类的静态成员

1、有的时候，类需要它的一些成员与类本身直接相关，而不是与类的各个对象保持关联

2、声明静态成员：我们通过在成员声明之前加上关键字 static 使得其与类关联在一起

3、类的静态成员存在于任何对象之外，对象中不包含任何与静态数据成员有关的数据

4、使用作用域运算符直接访问静态成员**(Java 中使用'类名'+'点运算符'访问)**

5、定义静态成员

- 在类外定义时，不能重复 static 关键字
- 一般来说，不能在类的内部初始化静态成员
- 必须在类的外部定义和初始化每个静态成员
- **要想保证对象只定义一次，最好的办法是把静态数据成员的定义与其他非内联函数的定义放在同一个 cpp 文件中**
- 我们可以为类的静态成员提供 const 整数类型的类内初始值，不过要求静态成员必须是字面值常量类型的 constexpr

# Chapter 8. IO 库

## 8.1. IO 类

1、头文件

- <iostream>：
  - istream
  - ostream
  - iostream
  - wistream
  - wostream
  - wiostream
- <fstream>：
  - ifstream
  - ofstream
  - fstream
  - wifstream
  - wofstream
  - wfstream
- <sstream>：
  - istringstream
  - ostringstream
  - stringstream
  - wistringstream
  - wostringstream
  - wstringstream

2、标准库使我们能忽略这些不同类型的流之间的差异，这是通过继承机制实现的。因此标准库流特性可以无差别的应用于普通流，文件流和 string 流

### 8.1.1. IO 对象无拷贝或赋值

1、由于不能拷贝 IO 对象，因此我们不能讲形参或返回类型设置为流类型

2、进行 IO 操作的函数通常以引用的方式传递和返回流，读写一个 IO 对象会改变其状态，因此传递和返回的引用不能是 const 的

### 8.1.2. 条件状态

1、各类条件状态：

- strm::iostate：strm 是一种 IO 类型，iostate 是一种与机器无关的类型，提供了表达条件状态的完整功能
- strm::badbit：指出流已崩溃
- strm::failbit：指出一个 IO 操作失败了
- strm::eofbit：指出流到达了文件结束
- strm::goodbit：指出流未处于错误状态，此值保证为 0

2、成员函数

- s.eof()：若流的 eofbit 置位，则返回 true
- s.fail()：若流的 failbit 货 badbit 置位，则返回 true
- s.bad()：若流的 badbit 置位，则返回 true
- s.good()：若流处于有效状态，则返回 true
- s.clear()：将 s 中所有条件状态复位，将流的状态设置为有效
- s.clear(flags)：根据给定的 flags 标志位，将流 s 中对应条件状态复位，若、flags 的类型为 strm::iostate。返回 void。
- s.setstate(flags)：根据给定的 flags 标志位，将流 s 中对应条件状态置位，flags 的类型为 strm::iostate。返回 void。
- s.rdstate()：返回流 s 的当前条件状态，返回值类型为 strm::iostate

3、查询流的状态

- 将流作为条件使用，只能告诉我们流是否有效(即返回 s.good())
- badbit 表示系统级错误
- 到达文件结束 eofbit 和 failbit 都会被置位
- goodbit 的值为 0 表示流未发生错误
- 如果 badbit、failbit 和 eofbit 任何一个被置位，则监测流状态的条件会失败
- 使用 good()或 fail()总能正确地表示流是否为有效状态，因为所有错误位未置位时，goot()才会返回 true；只要有一个错误位置位，那么 failbit 就会被置位，因此 fail()会返回 false

### 8.1.3. 管理输出缓冲

1、每个输出流都管理一个缓冲区，用来保存程序读写的数据

2、有了缓冲机制，操作系统就可以将程序的多个输出操作组合成单一的系统级操作，由于设备的写可能很耗时，允许操作系统将多个输出操作组合成单一的设备写操作可以大大提升性能。

3、缓冲区的刷新：

- 程序正常结束，作文 main 函数的 return 操作一部分，缓冲刷新被执行
- 缓冲区满时，需要刷新华冲，然后新的数据才能继续写入缓冲区
- 我们可以使用操纵符如 endl 来显式刷新缓冲区
- 在每个操作之后，我们可以用操纵符 unitbuf 设置流的内部状态，来清空缓冲区，默认情况下，cerr 是设置 unitbuf 的，因此写到 cerr 的内容都是立即刷新的
- **一个输出流可以被关联到另一个流，在这种情况下，当读写被关联的流时，关联到的流的缓冲区会被刷新，例如 cin 和 cerr 都关联到 cout，因此读 cin 或写 cerr 都会导致 cout 的缓冲区被刷新**
- **我们可以使用 unitbuf 操纵符，它告诉流，接下来的每次写操作都进行一次 flush 操作，而 nounitbuf 操纵符则重置流，使其恢复使用正常的系统管理的缓冲区刷新机制**
- 如果程序崩溃，缓冲区不会被刷新

4、关联输入和输出流

- 当一个输入流已被关联到一个输出流时，任何试图从输入流读取数据的操作都会先刷新关联的输出流

- 交互系统通常应该关联输入与输出流，这意味着所有输出，包括用户提示信息，都会在读操作之前被打印出来。
- s.tie()：返回其所关联的流的指针，若未关联到流，则返回空指针
- s.tie(ostream *p)：将该流 s 关联到 p 所指向的流
- 每个流最多关联一个流
- 一个流可以被多个流关联

## 8. 2. 文件的输入输出

1、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：向一个给定文件写入数据
- fstream：可以读写给定文件

2、fstream 特有的操作
- *fstream* fstream; 创建一个未绑定的文件流，*fstream* 指 ifstream、ofstream 以及 fstream 的任意一种
- *fstream* fstream(s); 创建一个 *fstream*，并打开名为 s 的文件，使用默认 mode 打开文件，具体的 mode 类型依据 *fstream* 的具体类型
- *fstream* fstream(s); 按指定 mode 打开文件
- fstrm.open(s)：打开名为 s 的文件，并将文件与 fstrm 绑定
- fstrm.close()：关闭与 fstrm 绑定的文件
- fstrem.is_open()：返回 bool，指出 fstrm 关联的文件是否**成功打开且尚未关闭**

3、**当一个 fstream 对象被销毁时，close 会自动被调用**

### 8. 2. 1. 使用文件流对象

### 8. 2. 2. 文件模式

1、方式：
- in：以读方式打开
- out：以写方式打开
- app：每次写操作均定位到文件末尾
- ate：打开后立即定位到文件末尾
- trunc：截断文件(覆盖写入)
- binary：以二进制的方式进行 IO

2、注意点：
- 只能对 ofstream 或 fstream 对象设定为 out 模式
- 只能对 ifstream 或 fstream 对象设定为 in 模式
- 只有 out 被设定时，才能设定 trunc 模式，即不能只设定为 trunc 模式
- 只要 trunc 没被设定，就可以设定 app 模式，在 app 模式下，即使没有显式指定 out 模式，文件也总是以输出方式被打开
- 默认情况下，即使没有指定 trunc，out 模式打开的文件也会被截断

3、**保留被 ofstream 打开的文件中已有数据的唯一方法是显式指定 app 或 in 模式**

## 8.3. string 流

1、sstream 定义了三个类型(istringstream、ostringstream 和 stringstream)来支持内存 IO，这些类型可以向 string 写入数据，从 string 读取数据，就像 string 是一个 IO 一样

2、stringstream 特有的操作

- *sstream* strm;　strm 是一个未绑定的 stringstream 对象，*sstream* 是三种类型中的一种
- sstream strm(s);　strm 是一个 *sstream* 对象，保存 string s 的一个拷贝，此构造函数是 explicit 的
- strm.str()：返回 strm 所保存的 string 的拷贝
- strm.str(s)：将 string s 拷贝到 strm 中，返回 void
- **strm.str("")：重置 stringstream 流，不要用 clear()，这是清空标志位**

3、**用途：**

- 按行读取文件，将读取的 string 对象 s 放入 istringstream 中，就可以利用>>来读取单词了(默认以空白为分隔)
- 作为将其他类型的数据转为 string 的方法，用 ostringstream 的对象，利用运算符<<吸收各个类型的数据，然后调用 s.str()转为 string

### 8.3.1. 标准 IO 库的格式化输出

# Chapter 9. 顺序容器

一个容器就是一些特定类型对象的集合，顺序容器为程序员提供了控制元素存储和访问顺序的能力，这种顺序不依赖元素的值，而是与元素加入容器时的位置相对应。

## 9.1. 顺序容器概述

1、不同元素在不同方面中有着不同的性能折中
- 向容器添加或从容器中删除元素的代价
- 非顺序访问容器中元素的代价

2、顺序容器类型：
- vector：可变大小数组，支持快速随机访问，在尾部之外插入元素很慢
- deque：双向队列，支持快速随机访问，在头尾位置插入删除很快
- list：双向列表，只支持双向顺序访问，在任何位置插入/删除操作速度很快
- forward_list：单向链表，只支持单向顺序访问，在链表任何位置插入删除都很快
- array：固定大小数组，支持快速随机访问，不能添加或删除元素
- string：与 vector 类似的容器，但专门用于保存字符，随机访问块，在尾部插入删除元素快

3、现代 C++程序应该使用标准库容器，而不是使用原始的数据结构--数组

4、通常 vector 是更好的选择，除非你有很好的理由选择其他容器

## 9.2. 容器库概览

1、类型别名
- iterator
- const_iterator
- size_type
- difference_type
- value_type
- reference
- const_reference

2、构造函数
- C c;
- C c1(c2);
- C c(b,e)
- C c{a1,b1,c1,...,}

3、赋值与 swap
- c1=c2
- c2={a,b,c}
- a.swap(b)
- swap(a,b)

4、大小
- c.size()
- c.max_size()
- c.empty()

5、添加删除元素
- c.insert(args)
- c.emplace(inits)
- e.erase(args)
- c.clear()

6、获取迭代器
- c.begin(),c.end()
- c.cbegin(),c.cend()

7、反向容器的额外成员
- reverse_iterator
- const_reverse_iterator
- c.rbegin(),c.rend()
- c.crbegin(),c.crend()

### 9.2.1. 迭代器

1、与容器一样，迭代器有着公共的接口：如果一个迭代器提供某个操作，那么所有提供相同操作的迭代器对这个操作的实现方式都是相同的
- 所有迭代器都允许我们访问容器中的元素，利用解引运算符
- 所有迭代器都定义了递增运算符，从当前元素移动到下一个元素

2、迭代器范围
- 由一对迭代器表示，两个迭代器分别指向同一个容器的元素或者是尾元素之后的元素，[begin,end)，左闭右开的区间
- 左闭合范围蕴含的编程假定：
  - 如果 begin 与 end 相等，则范围为空
  - 如果 begin 与 end 不等，则至少包含一个元素
  - 我们可以对 begin 递增多次，使得 begin==end

### 9.2.2. 容器类型成员

### 9.2.3. begin 和 end 成员

1、当不需要写访问时，应该使用 cbegin 和 cend

### 9.2.4. 容器的定义和初始化

1、形式
- C c; 如果 C 是一个 array，则 c 中元素按默认方式初始化，否则 c 为空
- C c1(c2); c1 初始化 c2，c1 与 c2 的类型必须相同(与 Java 不同)
- C c1=c2;
- C c{a,b,c,...}
- C c={a,b,c...}

- C c(b,e);  c 初始化为迭代器 b 和 e 指定范围中的元素的拷贝，范围中的元素类型必须与 C 的元素类型相容，array 不可用
- C c(n);  只有顺序容器(不包括 array)的构造函数才能接受大小参数
- C c(n,t)

2、array 具有固定大小
- 定义是需要指定容器大小
- 数组不能用于拷贝或用于初始化另一数组，但是 array 可以，类型大小必须相同

#### 9.2.5. 赋值和 swap

1、形式：
- c1=c2
- c2={a,b,c...}
- swap(c1,c2)
- c1.swap(c2)
- c.assign(b,e)  关联容器和 array 不支持 assign
- c.assign(il)
- c.assign(n,t)

2、swap
- swap 交换两个容器的操作保证会很快---元素本身并未交换，swap 只是交换了两个容器的内部数据结构，而不涉及拷贝，删除或插入操作。
- **思考一下 vector 的实现，一个 vector 对象内部会持有一个数组的指针，那么交换两个 vector 对象的话，只需要将这两个 vector 对象内部的数组指针交换一下，然后处理一下其他状态参数即可**

3、统一使用非成员版本的 swap 是一个好习惯

#### 9.2.6. 容器大小操作

1、c.size()
2、c.empty()
3、c.max_size()

#### 9.2.7. 关系运算符

1、所有容器都支持==和！=
2、除了无需关联容器外的所有容器都支持关系运算符(>,>=,<,<=)
3、关系运算符两边的运算对象必须是相同类型的容器
4、**容器的关系运算符使用元素的关系运算符完成比较**(与 Java 不同，对于引用而言，Java 中的==比较的是两个引用是否引用同一个对象，要比较值是否相同需要调用.equals()方法)

## 9.3. 顺序容器操作

#### 9.3.1. 向顺序容器添加元素

1、特例：

- array 不支持这些操作，因为会改变大小
- fordward_list 有专有版本的 insert 和 emplace
- fordward_liast 不支持 push_back 和 emplace_back
- vector 和 string 不支持 push_front 和 emplace_front

2、操作：

- c.push_back(t)
- c.emplace_back(args)
- c.push_front(t)
- c.emplace_front(args)
- c.insert(p,t)
- c.emplace(p,args)
- c.insert(p,n,t)
- c.insert(p,b,e)
- c.insert(p,il)

3、当我们用一个对象来初始化容器时，或将一个对象插入到容器中时，实际上放入到容器中的是对象值得一个拷贝，而不是对象本身，就像我们将一个对象传递给非引用参数一样，容器中的元素与提供值得对象之间没有任何关联(与 Java 不同，Java 中容器若保存的是类类型，那么放入的是对象的引用！)

4、**将元素插入到 vector、deque、string 中的任何位置都是合法的，但是可能很耗时，由于其中的元素是按地址依次存储的，若在中间插入，为了维护这个性质，需要将左边或者右边的元素依次向左/右移动一格，因此耗时**

5、使用 insert 的返回值

- iter=c.insert(iter,value);//循环调用这个，与循环调用 push_back()等价
- c.push_back(value);

6、emplace：

- 利用传递的参数构造一个元素，会调用该类对应的构造函数

### 9.3.2. 访问元素

1、包括 array 在内的所有顺序容器都有 front 成员函数，返回首元素的引用

2、除了 forward_list 之外的所有顺序容器都有一个 back 成员函数，返回尾元素的引用

3、注意：访问成员函数返回的是引用

- 访问越界会抛出 out_of_range 的异常
- 而直接访问，如果出现访问越界则会出现运行时的错误

### 9.3.3. 删除元素

1、支持的操作：

- c.pop_back()：删除 c 的尾元素
- c.pop_front()：删除 c 的首元素
- c.erase(p)：删除迭代器 p 指向的元素，返回被删除元素之后的迭代器
- c.erase(b,e)：删除[b,e)范围内的元素，返回 e
- c.clear()：删除 c 中所有元素
- 删除元素的成员函数并不检查其参数，因此必须保证被删除的元素是存在的

### 9.3.4. forward_list 的操作

P312

### 9.3.5. 改变容器大小

1、操作：

- c.resize(n)：调整 c 的大小为 n，若 n<c.size()，则多余的元素被丢弃
- c.resize(n,t)：调整 c 的大小为 n，任何新添加的元素赋值为 t

### 9.3.6. 容器操作可能使迭代器失效

1、建议：不要保存 end 返回的迭代器，而是要实时获取最新的 c.end()进行判断

## 9.4. vector 对象是如何增长的

1、当不得不获取新的内存空间时，vector 和 string 的实现通常会分配比新的空间需求更大的内存空间

2、每次重新分配内存空间需要移动所有元素，但是摊还代价通常比 list 和 deque 还要快

3、因此如果只需要在尾后递增容器的话，推荐使用 vector 而不是 list

4、capacity：不分配新的内存空间的前体下最多能保存多少个元素

5、size：当前容器已经保存的元素个数

## 9.5. 额外的 string 操作

1、详细请参考 P320-P328，下面仅列出比较重要的几种

### 9.5.1. 构造 string 的其他方法

1、构造函数

- string s(cp,n)：s 是 cp 指向数组中前 n 个字符的拷贝
- string s(s2,pos2)：s 是 string s2 从下标 pos2 开始的字符的拷贝
- string s(s2,pos2,len2)：s 是 string s2 从下标 pos 开始 len2 个字符的拷贝

2、substr

- s.substr(pos1,n)：返回 s 中[pos1,pos1+n)的子串
  - 若 pos1+n>s.size()，则只拷贝到末尾
- s.substr(pos1)：返回 s 中[pos1,end)的子串

### 9.5.2. 改变 string 的其他方法

1、append 和 replace 函数

2、**感觉 append 并不如 Java 中 StringBuilder.append()好用**

3、如果想要追加不同类型转为 string 的形式，最好用 string 流来操作

## 9.6. 容器适配器

1、除了顺序容器外，标准库还定义了三个顺序容器：

1) stack：定义在<stack>头文件中
2) queue：定义在<queue>头文件中
3) priority_queue：定义在<queue>头文件中

2、每个适配器都定义两个构造函数

- 默认构造函数创建一个空对象
- 接受一个容器的构造函数拷贝该容器来初始化适配器

3、更确切的，我认为叫装饰器更合适，因为我认为适配器是连接两个具有不同接口的一种功能类

# Chapter 10. 泛型算法

## 10.1. 概述

1、大多数算法定义在 algorithm 中
2、一般情况下，算法不直接操纵容器，而是遍历由两个迭代器指定的一个元素范围来进行操作，**因此算法永远不会改变底层容器的大小**

## 10.2. 初识泛型算法

1、规则：除了少数例外，标准库算法都对一个范围内(迭代器范围)的元素进行操作

### 10.2.1. 只读算法

1、find(begin,end,value)：
- 查找[begin,end)范围内的元素，返回第一个等于 value 的迭代器，若找不到，返回 end

2、accumulate(begin,end,iniVal)：
- 对[begin,end)范围内的元素求和，初值为 iniVal

3、count：
4、equal(begin1,end1,begin2)：
- 对范围[begin1,end)与 begin2 开始的元素进行对比，全部相同则返回 true

### 10.2.2. 写容器元素的算法

1、fill(begin,end,val)：
2、fill_n(begin,size,val)：

### 10.2.3. 重排容器元素的算法

1、unique

## 10.3. 定制操作

### 10.3.1. 向算法传递函数

1、谓词：
- 可调用对象(函数，函数指针，lambda，重载了函数调用运算符的类)
- 返回一个能用作条件的值
- 一元谓词：只接受单一参数
- 二元谓词：接受两个参数

### 10.3.2. lambda 表达式

1、lambda 表示一个可调用的代码单元，**可将其理解为一个未命名的内联函数**
2、lambda 具有一个返回类型，一个参数列表，和一个函数体

```
[capture list] (parameter list) -> returnType {function body}
```

- capture list：捕获列表，是 lambda 所在的函数中定义的局部变量的列表

- lambda 必须使用尾置返回来指定返回类型
- **可以忽略参数列表和返回类型，但是必须永远包含捕获列表和函数体**
- **如果 lambda 的函数体包含任何单一 return 语句之外的内容，且未指定返回类型，则返回 void**

3、与一个普通函数调用类似，调用一个 lambda 时给定的实参用来被初始化 lambda 的形参。lambda 不能有默认实参

4、一个 lambda 通过将局部变量包含在其捕获列表中来指出将会使用这些变量，捕获列表指引 lambda 在其内部包含访问局部变量所需的信息

5、一个 lambda 只有在其捕获列表中捕获一个他所在函数中的局部变量，才能在函数体中使用该变量

### 10.3.3. lambda 捕获和返回

1、概念：

- 当定义一个 lambda 时，编译器生成一个与 lambda 对应的未命名类类型
- 可以这样理解，当向一个函数传递一个 lambda 时，同时定义了一个新类型和该类型的一个对象
- 默认情况下，从 lambda 生成的类都包含一个对应该 lambda 所捕获的变量的数据成员，类似任何普通数据成员，lambda 的数据成员也在 lambda 对象创建时被初始化

2、值捕获：

- 采用值捕获的前提是变量可以拷贝
- 与参数不同，**被捕获的变量是在 lambda 创建时拷贝**，而不是调用时拷贝
- 由于被捕获变量的值是在 lambda 创建时拷贝，因此随后对其修改不会影响到 lambda 内对应的值

3、引用捕获：

- 如果以引用的方式捕获一个变量，就必须确保引用的对象在 lambda 执行的时候是存在的

4、建议：尽量保持 lambda 的变量捕获简单化

- 如果是普通变量，如 int、string 或其他非指针类型，可以采用简单的值捕获方式
- 如果捕获一个指针或迭代器，或采用引用方式捕获，必须确保在 lambda 执行时，绑定到迭代器、指针或引用的对象仍然存在

5、隐式捕获：

- 让编译器根据 lambda 体中的代码来推断我们需要使用哪些变量
- 在捕获列表中写一个&或=
- 可以与指定捕获混合使用，例如[x,&]以值捕获 x，其他以引用方式捕获

### 10.3.4. 绑定参数

1、对于捕获列表为空的 lambda，可以用函数来代替，但是捕获列表不为空，则无法用函数代替，因为谓词是严格匹配的，无法传递更多的参数。

2、bind (定义在 functional 中)：**接受一个可调用对象，生成一个新的可调用的对象来“适应”原对象的参数列表。auto newCallable=bind(callable,arg_list);**

3、newCallable 本身是一个可调用对象，arg_list 是一个逗号分隔的参数列表，对应给定的 callable 的参数。

- 例：

  auto check6=bind(check_size,_1,6)

  check_size(a,b) a 为 string ,b 为长度值 string::size_type

  **_1 代表由 check6 传给 check_size 的 check6 的第一个参数，以形参列表中的位置传递给 check_size**

  **本例中，_1 占位符在形参列表中占第一个位置，因此 check6 的第一个参数作为 check_size 的第一个参数**

4、占位符：新函数的第 n 个参数，传递给已有函数时绑定到占位符所在的形参位置_1 _2 …… _n 都定义在一个名为 placeholders 的命名空间中，而它本身定义在 std 命名空间中，因此使用_1 需要提供 using 声明：using std::placeholders::_1; 或者声明所有：using namespace std::placeholders;

- 例：

  auto g=bind(f,a,b,_2,c,_1);

  g 向 f 传递 2 个参数

  第一个参数占据着函数 f 的第五个形参

  第二个参数占据着函数的第三个形参

5、绑定引用参数

- 默认情况下，bind 那些不是占位符的参数被拷贝到 bind 返回的可调用对象中，但是与 lambda 相似，又是对有些绑定的参数希望以引用的方式传递，或是要绑定的参数的类型无法拷贝。
- 这种情况下，需要使用标准库 ref 函数，ref 返回一个对象，包含给定的引用，此对象是可以拷贝的。cref 函数，保存 const 引用的类。

## 10. 4. 再探迭代器

1、种类：

- 插入迭代器：这些迭代器被绑定到一个容器上，可用来向容器插入元素
- 流迭代器：这些迭代器被绑定到输入或输出流上，可用来遍历所有关联的 IO 流
- 反向迭代器：这些迭代器向后而不是向前移动，除了 forward_list 之外的标准库容器都有反向迭代器
- 移动迭代器：这些专用的迭代器不是拷贝其中的元素，而是移动它们

### 10. 4. 1. 插入迭代器

1、定义：

- 插入器是一种迭代器适配器，接受一个容器，生成一个迭代器，能实现向给定元素添加元素

2、操作：

- it=t：在 it 指定的当前位置插入 t，假定 c 是 it 绑定的容器，依赖于插入迭代器的不同种类，此赋值会分别调用 c.push_back(t)、c.push_front(t)、c.insert(t,p)，返回什么？

- *it,++it,it++：这些操作虽然存在，但不会 it 做任何事，每个操作返回 it

### 10. 4. 2. 流迭代器

1、类型：

- istream_iterator
- ostream_iterator

2、istream_iterator

- 当创建一个 istream_iterator 时，可以将它绑定到一个流
  - istream_iterator<int> int_it(cin);
- 可以默认初始化 istream_iterator，这样就创建了一个可以当做尾后值使用的迭代器
  - istream_iterator<int> int_eof;
- 可以使用 istream_iterator 初始化一个容器
  - vector<int> vec(int_it,int_eof);
- 操作：
  - in1==in2：in1 和 in2 必须读取相同类型，如果它们都是尾后迭代器或者绑定到相同的输入，则两者相等
  - in1!=in2：类似上
  - *int：返回从流中读取的数据
  - in->men：与(*in).men 含义相同
  - ++in,in++：递增迭代器

3、ostream_iterator

- 操作：
  - ostream_iterator<T> out(os);
  - ostream_iterator<T> out(os,d);
  - out=val：用<<运算符将 val 写入到 out 绑定的 ostream 中**(有点像插入迭代器)**
  - *out,out++,++out：运算符存在，但是不做任何事，都返回 out**(有点像插入迭代器)**

4、我们可以为任何定义了输入运算符>>的类型创建 istream_iterator 对象；同理可以为任何定义了输出运算符<<的类型定义 ostream_iterator 对象

5、istreambuf_iterator、ostreambuf_iterator

- 不会忽略空格，或者自动添加空格
- 可以用 istreambuf_iterator 来完整读取整个文件内容

6、读取整个文件内容的方法

- 读取至 char*

```
std::ifstream t;
int length;
t.open("file.txt");//open input file
t.seekg(0, std::ios::end); //go to the end
length = t.tellg();//report location (this is the length)
```

```
t.seekg(0, std::ios::beg);//go back to the beginning
buffer = new char[length];//allocate memory
dimension
t.read(buffer, length); // read the whole file into the buffer
t.close();//close file handle
```

- 利用流迭代器 istreambuf_iterator，读取到 string

```
ifstream t("file.txt");
string str((istreambuf_iterator<char>(t)), istreambuf_iterator<char>());
```

- 利用 rdbuf()，读取到 string

```
std::ifstream t("file.txt");
std::stringstream buffer;
buffer << t.rdbuf();
std::string contents(buffer.str());
```

#### 10.4.3. 反向迭代器

1、反向迭代器就是在容器中从尾元素(cbegin())向首元素反向移动的迭代器，其中 cend()为首前元素

## 10.5. 泛型算法结构

### 10.5.1. 5 类迭代器

1、**输入迭代器：**

- 用于比较两个迭代器的相等和不相等运算符你(==,!=)
- 用于推进迭代器的前置和后置递增运算符(++)
- 用于读取元素的解引用运算符(*)，**解引只会出现在赋值运算符的右侧**
- 箭头运算符(->),等价于(*it).member，即，解引用迭代器，并提取对象的成员
- 适用范围：只适用于顺序访问
- istream_iterator 是一种输入迭代器，find acculumate 要求输入迭代器
- 只能对序列进行单次扫描

2、**输出迭代器：**

- 用于推进迭代器的前置和后置递增运算(++)
- 解引用运算符(*)，**只能出现在赋值运算符的左侧**(向一个已经解引用的输出迭代器赋值，就是将值写入它所指向的元素
- 只能向一个输出迭代器赋值一次
- 只能对序列进行一次扫描
- 作为目的位置的迭代器通常都是输出迭代器，如 copy 的第三个参数
- ostream_iterator 是一种输出迭代器

3、**前向迭代器：**

- 可以读写元素，只能沿着一个方向移动
- 前向迭代器支持所有输入和输出迭代器的操作。
- 可以多次读写同一个元素

- 可以对序列进行多次扫描
- replace　　forward_list 上的迭代器时前向迭代器

4、**双向迭代器：**
- 可以正向/反向读写序列中的元素。
- 只支持所有前向迭代器的操作
- 支持前置和后置递减运算符
- list 的迭代器是双向迭代器
- reverse　　**除了 forward_list，其他标准库都提供符合双向迭代器要求的迭代器**

5、**随机访问迭代器：**
- 用于比较两个迭代器的相对位置的关系运算符(< <= > >=)
- 迭代器和一个整数值的加减运算(+、+=、-、-=)计算结果是迭代器在序列中前进(或后退)给定整数个元素后的位置
- 用于两个迭代器上的减法运算符(-)，得到两个迭代器的距离
- 下标运算符(iter[n]),和*(iter[n])
- sort
- **array deque string vector 的迭代器都是随机访问迭代器**
- 支持双向迭代器的所有功能
- 提供在常量时间内访问序列中任意元素的能力

### 10.5.2.　　算法形参模式

1、接受单个目标迭代器的算法假定：按其需要写入数据，不管写入多少个元素都是安全的
- alg(beg,end,other args);
- alg(beg,end,dest,other args);
- alg(beg,end,beg2,other args);

2、接受第二个输入序列的算法
- alg(beg,end,beg2,end2,other args);

3、算法命名规范
- 重载：不接受额外参数的算法，通常都是重载函数(如 sort)
- _if 版本：接受一个元素值的算法通常有另一个不同名的版本
- _copy 版本：写到额外目的空间的算法通常有一个在名字后面附加_copy 的版本

## 10.6. 特定容器算法

1、链表 list 和单项链表 forward_list 第一了几个成员函数形式的算法，定义了独有的 sort，merge，remove，reverse 和 unique。因为 sort 要求随机访问迭代器，因此不能用于 list 和 forward_list

2、链表特有的操作会改变容器的大小，而并非象征意义上的添加和删除

# Chapter 11. 关联容器

1、关联容器支持高效的关键字查找和访问
2、两个主要的关联容器类型是 map 和 set
3、map 中的元素是一些关键字-值(key-value)对，关键字起到索引的作用，值则表示与索引相关联的数据
4、set 中的元素只包含一个关键字，set 支持高效的关键字查询操作--检查一个给定关键字是否在 set 中
5、map 和 multimap 定义在 map 中，set 和 multiset 定义在 set 中，无序容器则定义在 unordered_map 和 unordered_set 中
6、类型：

- map
- set
- multimap
- multiset
- unordered_map
- unordered_set
- unordered_multimap
- unordered_multiset

## 11.1. 使用关联容器

1、map

- 定义一个 map 必须制定关键字和值得类型
- map<string,int> word_count

2、set

- 定义一个 set 必须指定元素类型
- set<string> s

## 11.2. 关联容器概述

1、关联容器不支持顺序容器的位置相关操作，因为关联容器中元素是根据关键字存储的
2、关联容器也不支持构造函数或插入操作这些接受一个元素值和一个数量值得操作
3、关联容器的迭代器是双向的

### 11.2.1. 定义关联容器

1、定义

- 一个 map 必须指明关键字类型与值类型
- 定义一个 set 需要指明关键字类型。

2、构造：

- 关联容器都定义了一个默认构造函数，它创建一个指定类型的空容器
- 也可以将关联容器初始化为另一个同类型容器的拷贝

- 支持从一个值范围来初始化关联容器，只要这些值可以转化为容器所需类型即可
- 新标准下，也可以用值初始化

3、map 或 set 中关键字必须是唯一的，但是 multimap 和 multiset 没有此限制

### 11.2.2. 关键字类型的要求

1、对于有序容器--map、multimap、set、multiset，关键字类型必须定义元素比较的方法，默认情况下使用关键字的<运算符来比较两个关键字

2、有序容器的关键字类型

- 可以向算法提供自己定义的比较操作
- **与之类似，我们也可以提供自己定义的操作来代替关键字上的<运算符。**
- **所提供的操作必须在关键字类型上定义一个严格弱序**
- **在实际编程中，如果一个类型定义了"行为正常"的<运算符，则它可以用作关键字类型**

3、使用关键字类型的比较函数

- 自定义的操作必须在尖括号中紧跟元素类型给出(比较操作类型，一种函数指针类型)
- multiset<Sales_data,**decltype(compareIsbn)***> bookstore(**compareIsbn**);
- 在模板参数中需要添加函数指针类型，并在初始化时传入该函数指针
- Java 中会更加规范，实现了 Comparable 接口或者 Comparator 接口即可

### 11.2.3. pair 类型

1、定义在头文件 utility 中

2、操作：

- pair<T1,T2> p;
- pair<T1,T2> P(v1,v2);
- pair<T1,T2> P={V1,V2};
- make_pair(v1,v2);
- p.first;
- p.second;

## 11.3. 关联容器操作

1、额外的类型别名：

- key_type：关键字类型
- mapped_type：每个关键字关联的值的类型，只适用于 map
- value_type：对于 set 与 key_type 相同，对于 map 为 pair<**const** key_type,mapped_type>
- 对于 set，key_type 和 value_type 是一样的，set 中保存的就是关键字
- 对于 map，元素是关键字-值对，即每个元素是一个 pair 对象，包含一个关键字和一个关联的值，由于我们不能改变一个元素的关键字，因此 pair 关键字部分是 const 的

### 11.3.1. 关联容器迭代器

1、解引用迭代器：

- 当解引用一个关联容器迭代器时，会得到一个类型为容器的 value_type 的值得引用
- 对于 map 而言，value_type 是一个 pair 类型，first 成员保存 const 的关键字，second 成员保存值
- 对于 set 而言，value_type 就是关键字类型

2、set 的迭代器是 const 的

- 虽然 set 同是定义了 iterator 和 const_iterator 类型，但是两种类型都只允许只读访问 set 中的元素
- 与不能改变 map 元素的关键字一样，set 的关键字也是 const 的

3、关联容器和算法

- **通常不对关联容器使用泛型算法**
- 关键字是 const 这一特性通常意味着不能讲关联容器传递给修改或重排容器元素的算法
- **关联容器定义了一个名为 find 的成员，它通过一个给定的关键字直接获取元素，效率非常高(Java 中是 Set.contains(key),Map.containsKey，或者 Set.add(...)也会返回一个 bool)**

### 11.3.2. 添加元素

1、insert 成员

- 关联容器的 insert 成员向容器中添加一个元素或一个元素范围，由于 map 和 set 中包含不重复的关键字，因此，插入一个已存在的元素对容器没有任何影响
- insert 有两个版本，分别接受一对迭代器或者是一个初始化列表
- set<int> set2;
- vector<int> ivec={2,3,4,5};
- set2.insert(ivec.cbegin(),ivec.cend())
- set2.insert({1,2,3,4,5})

2、向 map 添加元素

- 必须记住元素类型是 pair
- **通常想要插入的数据并没有一个现成的 pair 对象，可以在 insert 的参数列表中创建一个 pair(Java 采用的是 Map.put(key,value))**
- map<string,size_t> word_count;
- word_count.insert({word,1});
- word_count.insert(make_pair(word,1))
- word_count.insert(pair<string,size_t>(word,1));
- word_count.insert(map<string,size_t>::value_type(word,1));

3、关联容器的插入操作

- c.insert(v)
- c.emplace(args)
- c.insert(b,e)
- c.insert(il)

4、检测 insert 的返回值

- 对于不包含重复关键字的容器，添加单一元素的 insert 和 emplace 版本返回一个 pair，first 成员是迭代器，指向具有给定关键字的元素，second 成员是 bool 指，指出元素是插入成功(true)还是已经存在于容器中(false)
- 对于包含重复关键字的容器，接受单个元素的 insert 操作返回一个指向新元素的迭代器，无需返回 bool，因为总会成功插入

### 11.3.3. 删除元素

1、c.erase(k)：删除每个关键字为 k 的元素，返回一个 size_type 值，指出删除元素的数量，对于非 multi 的关联容器，只能是 1
2、c.erase(p)：返回被删除元素之后的迭代器
3、c.erase(b,e)：删除迭代器[b,e)范围内的元素

### 11.3.4. 下标操作

1、map 和 unordered_map 提供了下标运算符和一个对应的 at 函数
2、set 不支持下标，因为 set 中没有雨关键字相关联的值
3、不能对 multimap 或 unordered_multimap 使用下标，因为这些容器有多个值与一个关键字关联
4、**map 下标运算符接受一个索引，获取与此关键字相关联的值，但是与其他下标运算符不同的是，如果关键字不再 map 中，会为他创建一个元素并插入到 map 中，关联值将进行只初始化**
5、下标运算符返回类型：
- 通常，解引用一个迭代器所返回的类型与下标运算符返回的类型是一样的
- 对于 map，当对一个 map 进行下表操作返回一个 mapped_type 对象，解引用一个 map 迭代器会返回一个 value_value 对象

### 11.3.5. 访问元素

1、所关心的弱势一个特定元素是否在容器中，find 是最佳选择
2、对于不允许重复关键字的容器，find 和 count 没有区别，如果不需要计数，最好使用 find
3、查找操作：
- c.find(k)：返回迭代器，指向第一个关键字为 k 的元素，若 k 不在容器中，返回尾后迭代器
- c.count(k)：返回关键字等于 k 的元素的数量，对于不允许重复关键字的容器，返回值不是 0 就是 1
- c.lower_bound(k)：返回迭代器，指向第一个关键字**不小于** k 的元素，**如果容器存在元素 k 的话，指向第一个具有给定关键字的元素，如果容器中不存在 k 元素，那么返回的迭代器含义是插入该关键字的安全插入点**
- c.upper_bound(k)：返回迭代器，指向第一个关键字**大于** k 的元素，**(如果容器存在元素 k 的话，指向最后一个匹配给定关键字元素之后的元素)**
- **以上两个函数，若容器存在元素 k 的话，可以看做是返回一个包含 k 元素的左闭右开的范围[c.lower_bound(k),c.upper_bound(k))**
- **如果 c.lower_bound(k),c.upper_bound(k)返回相同的迭代器，则说明给定关键字不在容器中，但是返回不同的迭代器并不意味着 k 一定在容器中！**
- c.equal_range(k)：返回一个迭代器 pair，表示关键字等于 k 的元素的范围。

若 k 不存在，pair 两个成员均为 c.end()

4、在 multimap 或 multiset 中查找元素

- 如果一个 multimap 或 multiset 中有多个元素具有给定关键字，则这些元素在容器中会相邻存储

## 11.4. 无序容器

1、这些容器不是使用比较运算符来组织元素，而是使用哈希函数和关键字类型的==运算符。

2、在关键字类型的元素没有明显的序关系的情况下，无序容器是很有用的

3、理论上哈希计数能获得更好的平均性能

4、如果关键字类型固有是无序的，或者性能测试发现问题可以用哈希技术解决，就可以使用无序容器

5、管理桶(类似于算法导论上的槽位)

- 无序容器在存储上组织为一组桶，每个桶保存零个或多个元素
- 无序容器使用一个哈希函数将元素映射到桶
- 访问一个元素，容器首先计算元素的哈希值，它指出应该搜索哪个桶
- 容器将具有一个特定哈希值的所有元素都保存在相同的桶中
- 如果容器允许重复，那么所有具有相同关键字的元素也都会在同一个桶中
- 无需容器的性能依赖哈希函数的质量和桶的数量和大小
- 桶接口：
  - c.bucket_count()：正在使用的桶的数目
  - c.max_bucket_count()：容器能容纳的最多的桶的数量
  - c.bucket_size(n)：第 n 个桶中有多少元素
  - c.bucket(k)：关键字为 k 的元素在哪个桶中
- 桶迭代：
  - local_iterator：用来访问桶中元素的迭代器类型
  - const_local_iterator：桶迭代器的 const 版本
  - c.begin(n),c.end(n)：桶 n 的首元素迭代器和尾后迭代器
  - c.cbegin(n),c.cend(n)：const 版本
- 哈希策略
  - c.load_factor()：每个桶平均元素数量，返回 float 值

6、**map 以及 set 的底层实现是红黑树**

# Chapter 12. 动态内存

1、静态内存用来保存 static 对象、类的 static 数据成员以及定义在任何函数之外的变量
2、栈内存用来保存定义在函数内的非 static 对象
3、分配在静态内存或栈内存中的对象由编译器自动创建和销毁
- 对于栈对象，仅在其定义的程序块运行时才存在
- static 对象在使用之前分配，在程序结束时销毁

4、除了静态内存和栈内存，每个程序还拥有一个内存池，这部分内存被称作自由空间，或堆(heap)。程序用堆来存储动态内存-->那些在程序运行时分配的对象，动态对象的生存期由程序来控制，也就是说，当动态对象不再使用时，我们的代码必须显式得销毁它们

## 12.1. 动态内存与智能指针

1、在 C++中，动态内存是通过一对运算符完成的
- new：在动态内存中为对象分配空间，并返回一个指向该对象的指针
- delete：接受一个动态对象的指针，销毁该对象，并释放与之关联的内存

2、容易出现的问题：
- 忘记释放内存，会造成内存泄露
- 在尚有指针引用内存的情况下就释放了内存，造成产生引用非法内存的指针

3、为了更容易地使用动态内存，新标准库提供了两种智能指针类型来管理动态对象
- 智能指针的行为类似常规指针，重要的区别是他负责自动释放所指向的对象
- shared_ptr：允许多个指针指向同一个对象
- unique_ptr：独占所指向的对象
- weak_ptr：伴随类，弱引用，指向 shared_ptr 所管理的对象
- 以上三种类型都定义在<memory>头文件中

### 12.1.1. shared_ptr 类

1、支持的操作：
- shared_ptr<T> sp //空指针，指向 T 类型的对象
- unique_ptr<T> up
- p //将 p 用作条件判断，若 p 指向一个对象则为 true，**对于一般的指针则不能，除非将它赋值为 0，否则任何地址，无论是否有效都是 true**
- *p
- p->men
- p.get()返回 p 中保存的指针，返回的类型是内置指针类型，若智能指针释放了其对象，返回的指针所指向的对象也就消失了
- swap(p,q)
- p.swap(q)

2、shared_ptr 特有的操作：

- make_shared<T>(args) //返回一个 shared_ptr 指针，指向一个动态分配的类型为 T 的对象，使用 args 初始化此对象
- shared_ptr<T> p(q) //p 是 shared_ptr q 的拷贝，此操作会增加 q 中的计数器
- p=q //p 和 q 都是 shared_ptr，，所保存的指针必须能相互转换，此操作会递减 p 的引用计数，递增 q 的引用计数
- p.unique() //若 p.use_count()==1 则返回 true，否则 false
- p.use_count() //返回与 p 共享对象的智能指针的数量，很慢，用于调试

3、make_shared 函数
- **最安全的分配和使用动态内存的方法是调用一个名为 make_shared 的标准库函数**
- 此函数在动态内存中分配一个对象并初始化它，返回指向此对象的 shared_ptr
- make_shared 用其参数类构造给定的对象，即调用该类的构造函数
- 如果不传递任何参数 make_shared<T>()，对象就会值初始化

4、shared_ptr 的拷贝和赋值
- 当进行拷贝或赋值操作时，每个 shared_ptr 都会记录有多少个其他 shared_ptr 指向相同对象
- 我们可以认为每个 shared_ptr 都有一个关联计数器，称为引用计数
- 引用递增的情况：
  - 拷贝
  - 将其作为参数传递给函数，(非引用传递)
  - 作为函数的返回值，(非引用的返回类型)
- 引用递减的情况：
  - 对其赋值
  - 局部 shared_ptr 离开其作用域
- 当指向 一个对象的最后一个 shared_ptr 被销毁时，shared_ptr 类会自动销毁此对象

5、使用动态内存的原因：
- 程序不知道自己需要多少个对象
- 程序不知道所需对象的准确类型
- 程序需要在多个对象间共享数据

### 12. 1. 2. 直接管理内存

**1、自己直接管理内存的类与使用智能指针的类不同，它们不能依赖对象拷贝、赋值、和销毁操作的任何默认定义???**

2、使用 new 动态分配和初始化对象
- 在自由空间分配的内存是无名的，因此 new 无法为其分配的对象命名，而是返回一个指向该对象的指针
- 默认情况下，动态分配的对象是默认初始化的，这意味着内置类型或组合类型的对象的值将是未定义的，而类类型对象将用默认构造函数进行初始化
- 可以使用直接初始化方式来初始化一个动态分配的对象

- 可以使用传统的构造方式(圆括号)来初始化一个动态分配的对象
- 在新标准下，可以使用列表初始化(花括号)

➢ 可以采用值初始化(在类型名后跟上空括号)

- 对于定义了自己的构造函数的类类型，要求值初始化是没有意义的，不管采用什么形式，对象都会用默认构造函数来初始化
- 对于内置类型，值初始化的内置类型对象有着良好定义的值，而默认初始化的对象的值是未定义的

```
int *p=new int();
```

3、动态分配的 const 对象

➢ 一个动态分配的 const 对象必须初始化

➢ 对于定义了默认构造函数的类类型，其 const 动态对象可以隐式初始化

- const string *p=new const string; //合法

➢ 其他类型的对象就必须显式初始化

- const int *p=new const int; //不合法
- const int *p=new const int(5); //合法

4、内存耗尽

➢ 一个程序用光所有的内存，new 表达式就会失败

➢ 默认情况下，如果 new 不能分配所要求的内存空间，将会抛出 bad_alloc 的异常

➢ 可以改变 new 的方式来阻止抛出异常

- int *p=new (nothrow) int; 若无空间可分配，返回空指针
- 这种形式的 new 为"定位 new"

5、释放动态内存

➢ 通过 delete 表达式来讲动态内存归还给系统

➢ delete 接受一个指针，指向想要释放的对象

➢ 传给 delete 的指针必须指向动态分配的内存，或是空指针

6、delete 之后重置指针值：

➢ 当我们 delete 一个指针后，指针值就变为无效了，虽然指针已经无效，但是很多机器上指针仍然保存着(已经释放了的)动态内存的地址

➢ 在 delete 之后，指针就变成了空悬指针，即指向一块曾经保存过数据但现在已经无效的内存的指针

➢ **如果需要保留指针，可以在 delete 之后将 nullptr 赋予指针，这样就清除地指出指针不指向任何对象，而且可以用 p==nullptr 来进行判断了**

➢ **接上面，如果有多个指针指向相同的内存，在 delete 内存之后重置指针的方法只对该指针有效，对其他任何指向(已释放)内存的指针是没有作用的，因此只能提供有限的保护**

### 12.1.3. shared_ptr 和 new 结合使用(不建议!!!)

1、**如果我们不初始化一个智能指针，它就会被初始化为一个空指针**

2、我们可以用 new 返回的指针来初始化智能指针

3、接受指针参数的智能指针构造器是 explicit 的，必须使用直接初始化形式而不

能用拷贝初始化

4、**默认情况下，一个用来初始化智能指针的普通指针必须指向动态内存，因为智能指针默认使用 delete 释放它所关联的对象**

5、***不要使用 get 初始化另一个智能指针或者为智能指针赋值，因为使用 get 返回的指针的代码不能 delete 此指针***

6、支持的操作：

- shared_ptr<T> p(q) //p 管理内置指针 q 所指向的对象，q 必须指向 new 分配的内存，且能够转为还 T*类型
- shared_ptr<T> p(u) //p 从 unique_ptr 接管对象所有权，将 u 置空
- shared_ptr<T> p(q,d) //类似…，p 将使用可调用对象 d 来代替 delete
- shared_ptr<T> p(p2,d) //类似…
- p.reset() //若 p 是唯一指向其对象的 shared_ptr 会释放该对象
- p.reset(q) //令 p 指向 q
- p.reset(q,d) //令 p 指向 q，且用可调用对象 d 来代替 delete

### 12.1.4. 智能指针和异常

1、那些分配了资源，但是没有定义析构函数来释放这些资源的类，可能会遇到与使用动态内存相同的错误---程序员非常容易忘记释放资源

2、使用自己的操作释放资源

- 默认情况下 shared_ptr 假定它们指向的是动态内存，因此当一个 shared_ptr 被销毁时，它默认地对它管理的指针进行 delete 操作
- 我们可以为 shared_ptr 定义自己的删除器

### 12.1.5. unique_ptr

1、某时刻只能有一个 unique_ptr 指向一个给定对象

2、与 shared_ptr 不同，没有类似 make_shared 的标准库函数返回一个 unique_ptr，因此当我们定义 unique_ptr 时，必须将其绑定到一个 new 返回的指针上，初始化 unique_ptr 必须采用直接初始化形式

3、由于一个 unique_ptr 拥有它指向的对象，因此 unique_ptr 不支持普通拷贝或赋值操作

4、支持的操作：

- unique_ptr<T> u1 //空 unique_ptr
- unique_ptr<T,D> u2 //空 unique_ptr，**会使用类型为 D 的可调用对象释放指针**
- unique_ptr<T,D> u(d) //空 unique_ptr，使用类型为 D 的对象 d 代替 delete
- u=nullptr //释放 u 指向的对象，将 u 置空
- u.release() //u 放弃对指针的控制权，返回指针，将 u 置空
- u.reset() //释放 u 指向的对象
- u.reset(q) //令 u 指向这个对象
- u.reset(nullptr)

### 12.1.6. weak_ptr

1、weak_ptr 是一种不控制所指向对象生存周期的智能指针，它指向由一个

shared_ptr 管理的对象

2、将一个 weak_ptr 绑定到 shared_ptr 不会改变 shared_ptr 的引用计数，一旦最后一个指向对象的 shared_ptr 被销毁，对象就会被释放，即使有 weak_ptr 指向该对象，对象还是会被释放

3、P420-421

## 12.2. 动态数组

1、new 和 delete 运算符一次分配/释放一个对象，某些应用需要一次为很多对象分配内存的功能

2、C++语言和标准库提供两种一次分配一个对象数组的方法

- 另一种 new 表达式语法
- allocator 类，允许我们将分配和初始化分离

3、**大多数应用应该使用标准库容器而不是动态分配数组，因为容器更简单，而且不易出现内存管理错误并且有更好的性能**

### 12.2.1. new 和数组

1、语法：

- 为了让 new 分配一个对象数组，我们要在类型名后跟一对方括号，其中指明要分配的对象数目
- 方括号中的大小必须是整型，但不必是常量
- 分配一个数组会得到一个元素类型的指针
- new T[]

2、**我们用 new 分配一个数组时，并未得到一个数组类型的对象，而是得到一个数组元素类型的指针**

- **由于分配的内存不是一个数组类型，因此不能对动态数组调用 begin 或 end，也不能使用范围 for 语句来处理动态数组中的元素**
- **是否可以这样理解：标准库函数 begin，end 以及范围 for 语句在处理数组时，数组退化为指针的行为不会发生**

3、初始化动态分配对象的数组：

- 默认情况下，new 分配的对象，不管是单个分配还是在数组中，都是默认初始化的，**所以类类型必须要有默认构造函数，没有默认构造函数的类类型无法分配动态数组**
- 可以对数组中的元素进行值初始化，方法是在方括号后跟一堆空括号
  - int *p=new int[10]();
- 在新标准中还可以提供元素初始化器的花括号列表
  - int *p=new int [2]{1,2};
  - 如果初始化器数目小于元素数目，剩余元素值初始化
  - 如果初始化器数目大于元素数目，new 表达式失败

4、动态分配一个空数组是合法的

- 因为使用动态数组的原因之一就是我们不知道需要分配多大的数组，因此分配空数组是合法的特性可以增加代码的健壮性

5、释放动态数组：

- delete [] p;
- 数组中的元素按逆序销毁，即最后一个元素最先被销毁

- 如果在 delete 一个数组指针时忘记了方括号，或者在 delete 一个单一对象的指针使用了方括号，编译器很可能不会发出警告，但会在执行过程中产生异常的行为

6、智能指针和动态数组：

1、unique_ptr

- 标准库提供了一个可以管理 new 分配数组的 unique_ptr 版本，为了用一个 unique_ptr 管理动态数组，我们必须在对象类型后面跟一对空方括号
- unique_ptr<T []> u //u 可以指向一个动态分配的数组
- unique_ptr<T []> u(p) //u 指向内置指针指向的动态分配的数组
- u[i] //返回 u 拥有的数组中位置 i 处的对象

2、shared_ptr

- shared_ptr 不能直接支持管理动态数组
- shared_ptr 未定义下标运算符，而且不支持指针算数运算，为了访问数组中的元素，必须用 get 获取一个内置指针，然后用它来访问数组
- 如果希望 shared_ptr 管理一个动态数组，必须提供自定义的删除器
- shared_ptr<int> p(new int[10],[] (int *p) {delete[] p;});

### 12.2.2. allocator 类

1、new 有些灵活性上的局限

- 它将内存分配和对象构造组合在了一起
- delete 将对象析构和内存释放组合在了一起
- 没有默认构造函数的类就不能动态分配数组了!!!

2、allocate 类：

- 定义在<memory>头文件中
- 帮助我们将内存分配和对象构造分离开来
- 语法：
  - allocator<T> a：定义了一个名为 a 的 allocator 对象，可以为类型 T 的对象分配内存
  - a.allocate(n)：分配一段原始的、为构造的内存，保存 n 个类型为 T 的对象
  - a.deallocate(p,n)：释放从 T*指针 p 中地址开始的内存，这块内存保存了 n 个类型为 T 的对象；p 必须是一个先前由 allocate()返回的指针，且 n 必须是 p 创建时所要求的大小；在调用 deallocate 之前，用户必须对每个在这块内存中创建的对象调用 destroy(这只是要求而已，其实不对每个构造了对象的内存调用 destroy，也能执行 deallocate)
  - a.construct(p,args)：p 必须是类型为 T*的指针，指向一块原始内存，args 被传递给类型为 T 的构造函数，用来在 p 指向的内存中构造一个对象
  - a.destroy(p)：p 为 T*类型的指针，此算法对 p 指向的对象执行析构函数

3、allocator 分配未构造的内存

- allocator 分配的内存是未构造的，我们需要在此内存中构造对象
- 在新标准库中，construct 成员函数接受一个指针和零个或多个额外参数，在给定位置构造一个元素，额外参数用来初始化构造的对象
- 为了使用 allocate 返回的内存，我们必须用 construct 构造对象，使用未构造的内存，其行为是未定义的

- 我们只能对真正构造了的元素调用 destroy
- 传递给 deallocate 的指针不能为空，它必须指向由 allocate 分配的内存，而且传递给 deallocate 的大小参数必须与调用 allocate 分配时提供的内存大小参数具有一样的值

4、拷贝和填充未初始化内存的算法

- uninitialized_copy(b,e,b2)：从迭代器[b,e)指出的输入范围中拷贝元素到迭代器 b2 指定的未构造的原始内存中，b2 指向的内存必须足够大能够容纳输入序列中元素的拷贝
- uninitialized_copy_n(b,n,b2)：从迭代器 b 指向的元素开始，拷贝 n 个元素到 b2 开始的内存中
- uninitialized_fill(b,e,t)：在迭代器(或指针)[b,e)指定的原始范围中创建对象，对象值均为 t 的拷贝
- uninitialized_fill_n(b,n,t)：在迭代器(或指针)b 指向的内存地址开始创建 n 个对象，b 必须指向足够大的未构造原始内存，能够容纳给定数量的对象

# Chapter 13. 拷贝控制

1、拷贝控制操作

- 拷贝构造函数
- 拷贝赋值运算符
- 移动构造函数
- 移动赋值运算符
- 析构函数

## 13.1. 拷贝、赋值与销毁

### 13.1.1. 拷贝构造函数

1、定义：

- 如果一个构造函数第一个参数是**自身类型的引用，且任何额外的参数都有默认值**
- 拷贝构造函数在几种情况下会被隐式调用，因此拷贝构造函数不应该是 explicit 的

2、合成拷贝构造函数

- 如果没有为一个类定义拷贝构造函数，编译器会为我们定义一个"合成拷贝构造函数"
- 一般情况下，合成的拷贝构造函数会将其参数的成员逐个拷贝到正在创建的对象中，编译器从给定对象依次将每个非 static 成员拷贝到正在创建的对象中
- 每个成员的类型决定了它将如何拷贝
  - 对类类型成员：会使用拷贝构造函数来拷贝
  - 内置类型的成员：直接拷贝
  - **虽然不能直接拷贝数组，但是合成的拷贝构造函数会逐元素地拷贝一个数组类型的成员**

3、拷贝初始化

- 当我们使用直接初始化时，我们实际上是要求编译器使用普通的函数匹配来选择我们提供的参数最匹配的构造函数
- 当我们使用拷贝初始化时，我们要求编译器将右侧运算对象拷贝到正在创建的对象中，如果需要的话还需要类型转换
- 拷贝初始化**通常**使用拷贝构造函数来完成，但有时会使用移动构造函数或者是非 explict 的构造函数等

4、拷贝初始化发生的情况：

- 将一个对象作为实参传递给一个非引用类型的形参
- 从一个返回类型为非引用类型的函数返回一个对象
- 用花括号列表初始化一个数组中的元素或一个聚合类中的元素
- 标准容器调用 push 或者 insert 等方法时，与之相对 emplace 成员创建的元素都是直接初始化

### 13.1.2. 拷贝赋值运算符

1、与拷贝构造函数一样，如果未定义自己的拷贝赋值运算符，编译器会为它合

成一个

2、重载赋值运算符

- 本质上是函数，其名字由 operator 关键字后接表示要定义的运算符的符号组成
- 因此赋值运算符就是一个名为 operator=的函数
- =、[]、()、->这四个运算符必须定义为成员函数

3、定义

- 拷贝赋值运算符接受一个与其类型相同的参数(const T&)
- 赋值运算符通常返回一个指向其左侧运算对象的引用
- **标准库通常要求保存在容器中的类型要具有赋值运算符**

4、合成拷贝赋值运算符

- 如果一个类未定义拷贝赋值运算符，编译器会为它生成一个"合成拷贝赋值运算符"
- 一般情况下，它会将右侧运算对象的每个非 static 成员赋予左侧运算对象的对应成员，这一工作是通过成员类型的拷贝赋值运算符来完成的
- 对于数组成员，逐个复制数组元素
- 合成拷贝赋值运算符返回一个指向其左侧运算对象的引用

### 13. 1. 3. 析构函数

1、析构函数执行与构造函数相反的操作：构造函数初始化对象的非 static 数据成员，析构函数释放对象使用的资源，并销毁对象的非 static 数据成员

2、定义：

- 析构函数是类的一个成员函数
- 名字由波浪号接类名构成
- 没有返回值
- 不接受参数
- Java 中没有析构函数，有一个垃圾回收器，自动回收释放的资源
- 由于析构函数没有参数，因此不能被重载，一个给定类只会有唯一的析构函数

3、析构函数完成的工作

- 与构造函数首先执行初始化部分不同，析构函数首先执行函数体，然后销毁成员，成员按初始化顺序的逆序销毁
- 函数体可执行类设计者希望执行的任何收尾工作
- 析构函数不存在类似构造函数中初始化列表的东西来控制成员如何销毁，析构部分是隐式的
- **内置类型没有析构函数，销毁内置类型什么也不需要做**
- **隐式销毁一个内置类型的指针，不会 delete 它所指向的对象**

4、调用析构函数的情况：

- 对象被销毁
  - 变量离开作用域
  - 对象被销毁，其成员被销毁
  - 容器被销毁，元素被销毁
  - 对于动态分配的对象，delete 时
  - 对于临时对象，当创建它的完整表达式结束时被销毁

- 析构函数自动运行，无需担心何时释放资源

5、合成的析构函数：

- 当一个类未定义自己的析构函数时，编译器为他定义一个"合成析构函数"

### 13.1.4. 三五法则

1、五个操作：拷贝构造函数、拷贝赋值运算符、移动构造函数、移动赋值运算符、析构函数

2、**通常将其作为整体，需要定义其中一个，就定义所有！**

### 13.1.5. 使用=default

1、通过将五类操作中的一个操作定义为=default 来显式地要求编译器生成合成的版本

2、当我们在类内用=default 修饰成员的声明时，合成的函数将隐式地声明为内联的，如果不希望合成的成员是内联函数，应该只对成员的类外定义使用=default

3、我们只能对具有合成版本的成员函数使用=default(即拷贝控制成员)

### 13.1.6. 阻止拷贝

1、大多数类应该定义拷贝构造函数和拷贝赋值运算符

2、对于某些类这些操作没有合理的意义，这种情况下，必须采用某种机制阻止拷贝或赋值

3、定义删除的函数

- 在新标准下，我们可以通过将拷贝控制成员定义为删除的函数来阻止拷贝
- 删除的函数是这样一种函数：我们虽然声明了它，但不能以任何方式使用它
- 在函数的参数列表后面加上=delete 来指出我们将它定义为删除的
- **与=default 不同，=delete 必须出现在函数第一次声明的时候(对于成员函数只能在类内声明)**
- **与=default 不同，我们可以对任何函数指定=delete，但主要用于禁止拷贝控制成员**

4、**析构函数不能是删除的**

5、**合成的拷贝控制成员**可能是删除的

- 如果累的某个成员的析构函数是删除或不可访问的(private)，则类的合成析构函数被定义为删除的
- 如果类的某个成员的拷贝构造函数是删除的或不可访问的，则类的合成拷贝构造函数被定义为删除的
- 如果类的某个成员的析构函数是删除的或不可访问的，则类的合成拷贝构造函数被定义为删除的
- 如果累的某个成员的拷贝赋值运算符是删除的或不可访问的，或类有一个 const 的或引用成员，则类的合成拷贝赋值运算符被定义为删除的
- 如果累的某个成员的析构函数是删除的或不可访问的，或者类有一个引用成员，它没有类内初始化器，或是类有一个 const 成员，它没有类内初始化器且其类型未定义默认构造函数，则该类型的默认构造函数定义为删除的
- **本质的含义：如果一个类有数据成员不能默认构造、拷贝、复制或销毁，**

则对应的拷贝控制成员将被定义为删除的

6、private 拷贝控制

- 在新标准发布之前，类是通过将其拷贝构造函数和拷贝赋值运算符声明为 private 来阻止拷贝
- **声明但不定一个成员函数是合法的，试图访问未定义的成员将导致一个链接时错误**
- **通过声明(但不定义)private 的拷贝控制成员，我们可以预先阻止任何拷贝该类型对象的企图：试图拷贝对象的用户代码将在编译阶段被标记为错误**，成员函数或友元中的拷贝操作将会导致链接时的错误

## 13.2. 拷贝控制和资源管理

1、拷贝语义

- 行为像值
  - 意味着它应该也有自己的状态，拷贝一个像值的对象时，副本和原对象是完全独立的，改变副本不会对原来对象有任何影响
- 行为像指针
  - 行为像指针的类则共享状态，当我们拷贝这种类的对象时，副本和原对象使用相同的底层数据

### 13.2.1. 行为像值的类

1、类值拷贝赋值运算符

- 赋值运算符通常组合了析构函数和拷贝构造函数的操作
  - 赋值操作会销毁左侧运算对象的资源
  - 赋值操作会从右侧运算对象拷贝数据
  - 必须保证自赋值的正确性
  - 必须是异常安全的：当发生异常时，能将左侧运算对象置于一个有意义的状态
- 一个好的方法是：在销毁左侧运算对象之前拷贝右侧运算对象

### 13.2.2. 行为像指针的类

1、引用计数：

- 除了初始化对象以外，每个构造函数还要创建一个引用计数，用来记录有多少对象正在创建的对象共享状态。
- 当我们创建一个对象时，只有一个对象共享状态，因此计数值为 1
- 拷贝构造函数不分配新的计数器，而是拷贝给定对象的数据成员，包括计数器，拷贝构造函数递增共享计数器，指出给定对象的状态又被另一个新用户共享
- 析构函数递减计数器，如果计数器变为 0，析构函数释放状态
- 拷贝赋值运算符递增右侧运算对象的计数器，递减左侧运算对象的计数器
- 难点是在哪里放引用计数：
  - 将计数器保存在动态内存中
  - 当创建一个对象时，我们也分配一个新的计数器，当拷贝或赋值对象时，我们拷贝指向计数器的指针

## 13.3. 交换操作

1、除了定义拷贝控制成员，管理资源的类通常还定义一个名为 swap 的函数
2、对于那些与重排元素顺序的算法一起使用的类，定义 swap 是非常重要的，这类算法在需要交换两个元素时会调用 swap(有点 Java 的感觉了，类似接口方法)
3、与拷贝控制成员不同，swap 并不是必要的，但是对于分配了资源的类，定义 swap 可能是一种很重要的优化手段
4、例子：

```
class HasPtr{
    friend void swap(HasPtr&,HasPtr&);
private:
    string * ps;
    int i;
};
inline
void swap(HasPtr &lhs, HasPtr &rhs){
    using std::swap;
    swap(lhs.ps,rhs.ps);
    swap(lhs.i,rhs.i);
}
```

- 首先将 swap 定义为 friend，以便能访问 HasPtr 的数据成员
- 将其声明为 inline 函数，优化代码(由于 swap 的存在就是为了优化代码)
- swap 的函数体给对给定对象的每个数据成员调用 swap
- **swap 应该调用 swap 而不是 std::swap**
- 每个 swap 调用都应该是未加限定的，即调用应该是 swap 而不是 std::swap，如果存在特定类型的 swap 版本，其匹配程度会优于 std 中定义的版本，如果不存在特定的 swap 版本，则会调用 std 中的版本

5、在赋值运算符中使用 swap

- 定义 swap 的类通常用 swap 来定义他们的赋值运算符，这些运算符使用了一种名为"拷贝交换"的技术，将左侧运算对象与右侧运算对象的一个副本进行交换
- **在这个版本中，传递的参数并不是一个引用！因此传入的对象是右侧运算对象的一个副本，只有这样才能拿来交换，否则就会改变右侧运算对象的状态了**
- 这个技术的有趣之处在于它自动处理自赋值情况且天然就是异常安全的
  - 通过在改变左侧运算对象之前拷贝了右侧运算对象(值传递，传入的是一个拷贝)，保证了自赋值的正确性
  - 它保证异常安全的方法也与原来的赋值运算符实现一样

## 13.4. 拷贝控制事例

## 13.5. 动态管理内存

## 13. 6. 对象移动

1、新标准一个最主要的特性是可以移动而非拷贝对象的能力
2、很多情况下都会发生对象拷贝，在某些情况下对象拷贝后就被销毁了，若采用移动可以大幅度提升性能
3、另一个使用移动而不是拷贝的是类似 unique_ptr 这样的类，这些类都包含不能被共享的资源，因此这些类型的对象可以被移动但是不能被拷贝
4、**新标准中，容器可以保存不可拷贝的类型，只要它们可以移动即可**
5、**最开始我感觉就是改动一下名字所对应的地址。其实不是，是在新的对象的内存中，存放着源对象的内容，移动过后，源对象内存中的内容是任意的了**

### 13. 6. 1. 右值引用

1、右值引用：
- 所谓右值引用就是必须绑定到右值的引用
- 我们通过&&而不是&来获取右值引用
- **右值引用有一个重要特性：只能绑定到一个将要销毁的对象，因此我们可以自由地将一个右值引用的资源"移动到"另一个对象中**
- 类似任何引用，一个右值引用也不过是某一个对象的另一个名字

2、左值右值
- 左值和右值是表达式的属性
- 一个左值表达式表示的是一个对象的身份
- 一个右值表达式表示的是对象的值
- **我们不能将左值绑定到要求转换的表达式、字面常量或是返回右值的表达式，右值有着相反的绑定特性，但不能将右值引用绑定到左值上**
- 返回左值引用的函数，赋值，下标，解引用，前置递增/递减符号都是返回左值
- 左值持久；右值短暂

3、变量是左值
- 变量可以看做只有一个运算对象而没有运算符的表达式
- 变量表达式也有左右/右值属性，**变量表达式都是左值**
- 我们不能将一个右值引用绑定到一个右值引用类型的变量上

4、标准库 move 函数
- 我们不能将一个右值引用绑定到左值上，但是我们可以显式得将一个左值转换为对应的右值引用类型
- 我们还可以通过调用一个名为 move 的新标准库函数来获得绑定到左值上的右值引用，定义在<utility>中
- move 告诉编译器：我有一个左值，但我希望像一个右值一样处理它
- **调用 move 就意味着：除了赋值或销毁它之外，我们将不再使用它，在调用 move 后，我们不能对移后源对象的值做任何假设**
- **不提供 using 声明，直接调用 std::move**
- **std::move 的含义：强制类型转换，即(T&&)**

### 13. 6. 2. 移动构造函数和移动赋值运算符

1、移动构造函数

- 移动构造函数的第一个参数是该类类型的一个引用
- 不同于拷贝构造函数，这个引用参数在移动构造函数中是右值引用
- **与拷贝构造函数一样，额外的参数必须有默认实参**
- 除了完成资源移动，移动构造函数还必须保证移后源对象处于这样一种状态:销毁它是无害的
- 一旦资源完成移动，源对象必须不再指向被移动的资源--这些资源的所有权已经归属新创建的对象

2、移动操作、标准库容器和异常

- 移动操作窃取资源，通常不分配任何资源，因此移动操作通常不会抛出任何异常
- 当编写一个不抛出异常的移动操作时，应该将此事通知标准库
- 一种通知标准库的方法是在我们的构造函数中声明 noexcept，在一个函数的参数列表后面指定 noexcept，对于构造函数，noexcept 出现在参数列表和初始化列表开始的冒号之间
- 不抛出异常的移动构造函数和移动赋值运算符必须标记为 noexcept
- 理解标准库与我们自定义的类型如何交互：
  - 移动操作通常不抛出异常，但是也可以抛出异常
  - 标准库容器能对异常发生时其自身的行为提供保证：即 vector 保证，如果调用 push_back()发生了异常，vector 自身不会发生改变
  - 移动一个对象通常改变它的值，如果在移动了部分元素后抛出了一个异常，就会产生问题，旧空间中移动源元素已经被改变，而新空间中未构造的元素可能尚不存在，在这种情况下，vector 将不能满足资深保持不变的要求
  - 如果 vector 使用了拷贝构造函数且发生了异常，它可以很容易地满足要求，即释放新分配的内存并返回，vector 原有的元素仍然存在
  - 为了避免这种潜在的问题，除非 vector 知道元素类型的移动构造函数不会抛出异常，否则在重新分配内存的过程中，必须使用拷贝构造函数而不是移动构造函数

3、移动赋值运算符

- 如果移动赋值运算符不抛出任何异常，我们就应该将其标记为 noexception
- 类似拷贝赋值运算符，移动赋值运算符必须正确处理自赋值(if(this!=&t))

4、移后源对象必须可析构

- 当我们编写移动操作时，必须确保移后源对象进入一个可析构的状态
- 移动操作还必须保证对象仍然是有效的，即可以安全地为其赋予新值或者可以安全地使用而不依赖其当前值
- 我们的程序不应该依赖于移后源对象中的数据

5、合成的移动操作：

- 以下情况，编译器不会合成移动操作：
  - 如果一个类定义了拷贝构造函数，拷贝赋值运算符或析构函数
- 只有当一个类没有定义任何自己版本的拷贝控制成员，且类的每个非 static 数据成员都可以移动时，编译器才会为它合成移动构造函数或移动赋值运算符
  - 对于内置类型，编译器可以移动它

- 对于类类型，只有该类型定义了对应的移动操作，编译器才能移动这个成员
- 与拷贝操作不同，移动操作永远不会隐式定义为删除函数
- 如果我们显式地要求编译器生成=default 的移动操作，且编译器不能移动所有成员时，编译器会将移动操作定义为删除的函数

6、**总的法则就是：如果需要定义拷贝操作的其中一个，就定义 5 个！**

7、移动右值，拷贝左值

- 对于一个既定义了移动构造函数又定义了拷贝构造函数的类，编译器使用普通的函数匹配规则来确定使用哪个构造函数，即左值拷贝，右值移动
- 如果没有移动构造函数，那么右值也会被拷贝
- 用拷贝构造函数代替移动构造函数几乎肯定是安全的(赋值运算符也类似)

### 13. 6. 3. 右值引用和成员函数

1、标准库容器的 push_back 提供两个版本

- void push_back(**const X&**);
- void push_back(**X&&**);
- 一般来说，我们不需要为函数操作定义接受一个 const X&&或者 X&
  - 因为当我们希望窃取数据时，通常传递一个右值引用，为了达到这个目的，实参不能是 const，否则就无法窃取了
  - 类似，从一个对象进行拷贝的操作不应该改变该对象，因此通常不需要定义一个接受 X&参数的版本
- 区分移动和拷贝的重载函数通常一个版本接受 const T&，另一个版本接受 T&&

2、右值和左值引用成员函数

- **通常，我们在一个对象上调用成员函数，而不管该对象是一个左值还是一个右值**
- void function(args) const &;
- void function(args) &&;
- 声明和定义都需要加上，很好理解，不然无法知道函数定义对应哪个函数

### 13. 6. 4. 对象移动的理解，参考暖神

1、所谓"拷贝"和"赋值"仅仅是赋值表达式的语义而已

```
class T{
    T& operator=(const T& arg);   //拷贝赋值
    T& operator=(T&& arg);   //移动赋值
}
```

- 本质上，这就是两个不同的函数，但碰巧都写作赋值表达式
- 这两个函数的语义不同
  - 第一个的意思是把参数 arg 及其内部的资源复制到当前对象
  - 第二个是剥夺 arg 内部的资源，把它移动到当前对象里来
  - **对于字符串这种没有副作用的对象来说，可以完全用拷贝来实现移动。但正是因为移动赋值函数允许摧毁 arg，使得程序员允许用更高效的方法实现移动赋值**
  - C++11 里的"拷贝赋值"和"移动赋值"根本是两个不同的函数，它们有不

同的语义，**只是 C++允许"移动赋值"进行破坏性的操作**。正是因为 C++允许破坏性操作，使得程序猿可以利用破坏来更高效地实现移动语义。**但真正怎么实现移动，怎么"破坏"才更高效，就看程序猿自己实现了。*C++只是提供了语法***

2、对象移动的两种情况：

- 对于像 int 这样的类型，完全可以用拷贝来实现移动，"移动后"源对象依然存在
- 对于含有**类似指针类型的数据成员**的类型，对该类型的对象移动就像接管指针一样，接管资源后，挂起源对象的资源，如果是指针的话，就赋值为 nullptr，这种类型的移动会销毁源对象，但比拷贝要更高效，因为不必再次分配资源

# Chapter 14. 重载运算符与类型转换

## 14.1. 基本概念

1、重载的运算符是具有特殊名字的函数：它们的名字由关键字 operator 和其后要定义的运算符号共同组成，重载运算符也包含返回类型，参数列表以及函数体
2、重载运算符的参数数量与该运算符作用的运算对象数量一样多
3、**如果一个运算符函数是成员函数，则它的第一个运算对象绑定到隐式的 this 指针上，因此成员函数的参数数量比运算符的运算对象总数少一个**
4、**对于一个运算符函数来说，它或者是类的成员，或者至少含有一个类类型对象，也就是不能为内置类型重定义运算符**
5、**我们只能重载已有的运算符，不能发明新的符号**
6、不能被重载的运算符：

- ::
- .*
- .
- ?:

7、对于一个重载的运算符来说，优先级和结合律与对应的内置类型保持一致
8、不建议重载的运算符

- **逻辑与**和**逻辑或**运算符，运算对象求值顺序和短路属性无法保留，因此不建议重载它们
- **逗号**运算符和**取址**运算符也不建议重载，因为 C++语言，已经定义了这两种运算符作用于类类型对象时的特殊含义

9、符合赋值运算符应该继承而非违背其内置版本的含义，如果该类含有算数运算符，那么最好也提供符合赋值运算符
10、必须作为成员函数的运算符

- 赋值运算符:=
- 下标运算符:[]
- 函数调用运算符:()
- 成员访问箭头运算符:->

11、建议作为成员的运算符

- 符合赋值运算符
- 递增递减和解引用

12、建议作为非成员的运算符

- **具有对称性的运算符可能转换任意一端的运算对象，因此建议定义为非成员的运算符**
- 算数运算符
- 相等性运算符
- 关系运算符
- 位运算符

## 14.2. 输入输出运算符

- IO 标准库分别使用>>和<<执行输入输出操作

### 14.2.1. 重载输出运算符<<

1、通常情况下，输出运算符的第一个形参是一个非常量 ostream 对象的引用，因为向流写入数据会改变其状态，而且形参是引用是因为我们没法直接复制一个 ostream 对象，**该运算符通常返回给定流的引用**

- ostream& operator<<(ostream& os,const T& t);

2、第二个形参一般来说是一个常量引用，该常量是我们想要打印的类类型，采用引用是希望避免复制实参，采用常量是因为打印对象不会改变其内容

3、输出运算符尽量减少格式化操作，将格式化操作留给用户控制

4、**输出运算符必须是非成员函数，因为第一个运算对象必须是 iostream 的对象**

### 14.2.2. 重载输入运算符>>

1、通常情况下，输入运算符的第一个形参是运算符将要读取的流的引用，第二个形参是将要读入到的对象的引用，**该运算符通常会返回某个给定流的引用**，第二个形参必须是非常量是因为输入运算符本身目的就是将数据读入到这个对象中

- istream& operator>>(istream &is, T &t);

2、**输入运算符必须要处理可能失败的情况(输入运算符应该负责从错误中恢复)，输出运算符不需要**

3、输入运算符失败的情况

- 当流含有错误类型的数据时读取操作可能失败
- 当读取操作达到文件尾或者遇到输入流的其他错误时也会失败

## 14.3. 算数和关系运算符

1、通常情况下，把算数和关系运算符定义成非成员函数以允许左侧或右侧运算对象进行交换，这些运算符一般不需要改变运算对象的状态，因此形参都是常量引用

- T operator+(const T& t1,const T& t2);

2、如果定义了复合赋值运算符，通常情况下应该使用复合赋值运算符来实现算数运算符

### 14.3.1. 相等运算符

1、通常情况下，关系运算符一般不需要改变运算对象的状态，因此形参都是常量引用

- bool operator==(const T&t1,const T& t2);

2、通常，如果定义了 operator==,也应该定义 operator!=

3、相等关系和不等关系的其中一个应该把工作委托给另一个，这意味着其中一个运算符应该负责实际比较对象的工作，另一个只是调用真正工作的运算符

### 14.3.2. 关系运算符

1、关联容器和一些算法会用到小于运算符，所以定义 operator<会比较有用

## 14.4. 赋值运算符

1、赋值运算符必须定义为成员函数
2、符合赋值运算符也倾向于定义为成员函数
3、返回左侧对象的引用

- T& operator=(const T& t);
- T& operator+=(const T& t);

## 14.5. 下标运算符

1、下标运算符必须是成员函数
2、通常以访问元素的引用作为返回值，这样做的好处是，下标可以出现在赋值运算符的任意一端
3、最好同时定义下标运算符的常量版本和非常量版本，**当作用于一个常量对象时，下标运算符返回常量引用(VS15 会强制让你加上 const，之前可以不加 const，导致可以对返回值进行复制)，以确保不会给返回对象赋值**

## 14.6. 递增和递减运算符

1、定义递增递减运算符的类应该同时定义前置版本和后置版本
2、这些运算符通常被定义为类的成员函数
3、**前置运算符应该返回递增或递减后对象的引用**
4、**后置运算符应该返回对象的原值**
4、区分前置后置运算符

- 后置运算符接受一个额外的(但不使用)的 int 类型的参数
- T& operator++();
- T operator++(int);

## 14.7. 成员访问运算符

1、箭头运算符必须是类的成员
2、解引用运算符通常也是类的成员
3、我们通常将这两个运算符定义成 const 成员

## 14.8. 函数调用运算符

1、函数调用运算符必须是类的成员
2、**定义了函数调用运算符的类，我们能像使用函数一样使用该类的对象(是一个闭包吗???)**

### 14.8.1. lambda 是函数对象

1、当我们编写一个 lambda 后，编译器将该表达式翻译成一个未命名类的未命名对象
2、lambda 表达式产生的类中含有一个重载的函数调用运算符，**默认情况下，由**

**lambda 产生的函数调用运算符是一个 const 成员函数**，如果 lambda 被声明为可变的，则函数调用运算符就不是 const 的了

3、捕获列表：

- 当一个 lambda 表达式通过引用捕获变量时，将由程序确保 lambda 执行时引用所引的对象确实存在，因此编译器可以直接使用该引用而无须在 lambda 产生的类中将其存储为数据成员
- 当一个 lambda 表达式通过值捕获变量时，将在 lambda 产生的类中为每个值捕获的变量建立对应的数据成员，同时创建构造函数，令其使用捕获的变量的值来初始化数据成员

### 14.8.2. 标准库定义的函数对象

1、标准库定义了一组表示算数运算符、关系运算符和逻辑运算符的类，每个类分别定义了一个执行命名操作的调用运算符

- plus 类定义了一个函数调用运算符用于对运算对象执行+操作
- modules 类定义了一个函数调用运算符执行二元的%操作
- 这些都定义成了模板形式

### 14.8.3. 可调用对象与 function

1、C++中的可调用对象：函数、函数指针、lambda 表达式、bind 创建的对象，以及重载了函数调用运算符的类

## 14.9. 重载、类型转换与运算符

1、类型转换

- 含有一个形参的非 explicit 构造函数定义了一种类型转换：这种构造函数可以将实参类型的对象转换成类类型
- 定义类型转换运算符也能够实现这一目的，这被称为用户定义的类型转换

### 14.9.1. 类型转换运算符

1、一般形式

- operator T() const;
- 其中 T 可以是处理 void 之外的任意类型，只要能作为函数的返回类型即可(因此不能转换成数组或者函数类型，但是可以转换成数组或函数的指针或引用类型)
- **必须定义成类的成员函数**
- **类型转换运算符是隐式执行的，所以无法给函数传递实参，因此类型转换运算符行参列表为空**
- **类型转换运算符不能声明返回类型，但是函数体内都会返回一个对应类型的值**
- 类型转换运算符通常不改变待转换对象的内容，因此一般定义成 const

2、没有充分的理由，不要用类型转换运算符

- int i=1;
- cin<<i; //由于 cin 没有定义<<，但是 cin 定义了从 istream 到 bool 的类型转换，这个语句会变成 bool 类型的位移语句

3、显式的类型转换运算符

- **explicit** operator T() const;
- 使用显式的类型转换：static_cast<T>()
- 如果表达式被作用于条件，编译器会将显式的类型转换自动应用与它
  - if while do 语句的条件部分
  - for 语句的条件表达式
  - 逻辑运算符! || &&的运算对象
  - ?:语句的条件表达式
  - 向 bool 的类型转换通常用在条件部分，因此 operator bool 一般定义成 explicit

### 14.9.2. 避免有二义性的转换

1、如果类中包含一个或多个类型转换，则必须确保在类类型和目标类型之间只存在唯一一种转换方式

### 14.9.3. 函数匹配与重载运算符

# Chapter 15. 面向对象程序设计

## 15.1. OOP 概述

1、面向对象程序设计的核心思想是数据抽象、继承和动态绑定

- 通过数据抽象，可以将类的接口与实现分离
- 继承，可以定义想死的类型并对其相似关系建模
- 动态绑定，在一定程度上忽略相似类型的区别，而以统一的方式使用它们的对象

2、继承

- 通过继承联系在一起的类构成一种层次关系
- 通常在层次关系的根部有一个基类
- 其他类则直接或间接地从基类继承而来，这些继承得到的类称为派生类
- 基类负责定义层次关系中所有类共同拥有的成员，每个派生类定义各自特有的成员
- 基类将类型相关的函数与派生类不做改变直接继承的函数区分对待，对于某些函数，**基类希望它的派生类各自定义适合自身的版本，此时基类就将这些函数声明成虚函数(与 Java 不同，Java 没有虚函数的概念，所有的非 private 方法都是动态的)**
- 派生类必须通过使用派生列表明确指出它从哪个基类继承而来(Java 的继承使用 extends 关键字和 implements 关键字)
- **派生类必须在其内部对所有重新定义(也可以不重新定义直接继承)的虚函数进行声明，派生类可以在这样的函数之前加上 virtual 关键字(但不是非得这样做)**
- C++11 新标准允许派生类显式地注明它将使用哪个成员函数改写基类的虚函数，具体措施实在该函数的形参列表之后增加 override 关键字(Java 使用的是@Override 注解)

3、动态绑定

- 在运行时选择函数的版本，动态绑定又被称为运行时绑定
- C++中，当我们使用基类的引用或指针调用一个虚函数时发生动态绑定(Java 是引用，因为 Java 只能定义类类型的引用)

## 15.2. 定义基类和派生类

### 15.2.1. 定义基类

1、**基类通常都应该定义一个虚析构函数，即使该函数不执行任实际操作**

2、成员函数与继承

- 派生类可以继承基类的成员，也可以提自己的新定义以覆盖从基类继承而来的旧定义
- C++中，基类必须将它的两种成员函数区分开来，一种是基类希望其派生类进行覆盖的函数，另一种是基类希望派生类直接继承而不要改变的函数(Java 不区别对待，要直接继承，就不要自己定义，想要自定义新版本就自己定义)
  - 对于前者，通常将其定义为虚函数

- 基类通过在成员函数的声明语句前加上 virtual 关键字使得该函数执行动态绑定
- 任何构造函数之外的非静态函数都可以是虚函数
- 关键字 virtual 只能出现在类内部的声明而不能用于类外部的函数定义
- 如果基类把一个函数声明为虚函数，那么在派生类中，该函数隐式地是虚函数(无论加不加 virtual 关键字)

➢ 如果函数没有被定义成虚函数，其解析过程发生在编译时而非运行时

3、访问控制与继承

➢ **派生类可以继承定义在基类中的成员，但是派生类的成员函数不一定有权访问从基类继承而来的成员**

➢ 与其他使用基类的代码一样，派生类能访问共有成员，而不能访问私有成员

➢ 基类还有一种成员，基类希望它们的派生类有权访问该成员，同时禁止其他用户访问，使用 protected 访问运算说明符

### 15. 2. 2. 定义派生类

1、派生类必须通过使用派生类列表明确指出它从哪个(哪些，(Java 只支持到单继承，不支持多重继承))基类继承而来

2、派生类列表的形式：

➢ 首先是一个冒号:

➢ 后面紧跟以逗号分隔的基类列表

➢ 每个基类列表可以有以下三种访问说明符中的一种**(Java 没有这种复杂的继承访问说明，继承就是继承，因此继承关系比较容易理清)**

- public
- protected
- private

➢ ***访问说明符的作用是控制派生类从基类继承而来的成员是否对派生类的用户可见***

3、派生类中的虚函数

➢ 派生类经常(但不总是)覆盖它继承的虚函数

➢ 如果派生类没有覆盖其基类中的某个虚函数，该虚函数的行为类似于其他的普通成员，派生类会直接继承其在基类中的版本**(但是该函数仍然支持动态绑定，只是与静态绑定没有区别，因为函数就是基类中定义的)**

➢ 派生类可以在它覆盖的函数前使用 virtual 关键字(但不是非得这么做)，C++11 新标准允许派生类显式得注明它使用某个成员函数覆盖了它继承的函数，具体做法是：在{前加一个 override 关键字

4、派生类对象及派生类向基类的类型转换

➢ 一个派生类对象包含多个组成部分

- 一个含有派生类自己定义(非静态)成员子对象
- 一个派生类继承的基类对应的子对象

➢ 因为在派生类对象中含有与其基类对应的组成部分，所以我们能把派生类的对象当成基类对象来使用，我们能将基类的指针或引用绑定到派生类对象中的基类部分

➢ 这种转换成为**派生类到基类**的类型转换

- **派生类对象中包含与其基类对应的组成部分，这一事实是继承的关键(Java 也是如此，对于 private 的域，派生类无法直接调用，但是事实上却是包含的，因为可以用非 private 的函数来使用它)**

5、派生类构造函数

- **派生类对象中含有从基类继承而来的成员，但是派生类并不能直接初始化这些成员，和其他创建了基类，派生类必须使用基类的构造函数来初始化它的基类部分(与 Java 一样)**
- 每个类控制它自己的成员的初始化过程，C++中，派生类构造函数通过在构造函数初始化列表显式调用基类构造函数来对基类成员进行初始化，"类名(args)"**(Java 必须在函数体中第一句调用 super(args)来显式调用基类的构造函数**

6、派生类使用基类的成员

- 派生类(内部)可以访问基类共有成员和受保护成员，但是不能访问私有成员(但事实上是含有的，只是不能访问，与 Java 一致)

7、继承与静态成员

- 如果基类定义了一个静态成员，则在整个继承体系中只存在该成员的唯一定义，不论从基类派生出多少个派生类，对于每个静态成员来说都只存在唯一实例
- 静态成员遵循访问控制规则，派生类无法访问基类中的 private 的静态成员

8、派生类的声明

- 派生类的声明与其他差别不大，声明中包含类名但是不包含派生列表

9、被用作基类的类

- 如果想将某个类用作基类，则该类必须已经定义而非仅仅声明
- 隐式的意思：一个类不能派生它本身

10、防止继承的发生

- 不希望一个类成为基类，在类名后面加 final
- class ClassName final{};

### 15. 2. 3. 类型转换与继承

1、通常情况下，我们想把一引用或指针绑定到一个对象上，则引用或指针的类型应与对象的类型一致，或者对象的类型含有一个可接受 const 类型转换规则

2、**可以将基类的指针或引用绑定到派生类对象有一层极为重要的含义：当使用基类的引用或指针时，实际上我们并不清楚该引用或指针所绑定对象的真实类型，可能是基类也可能是派生类**

3、静态类型与动态类型

- 表达式的静态类型在编译时总是可知的，它是变量声明时的类型或表达式生成的类型
- 动态类型则是变量或表达式表示的内存中的对象的类型，动态类型直到运行时才可知
- **如果表达式既不是引用也不是指针，它的动态类型永远与静态类型一致**

4、不存在从基类向派生类类型的转换，很好理解，派生类有自己额外的东西，无法转换，但是从派生类向基类转换，只需要丢弃这些额外的部分即可(你只要保证你不会用这些额外的东西就行了)

5、在对象之间不存在类型转换
- 派生类向基类的自动转换只对指针或引用类型有效
- 在派生类和基类类型之间不存在这样的转换

## 15.3. 虚函数

1、对虚函数的调用可能在运行时才被解析
2、派生类中的虚函数
- 当我们在派生类中覆盖了某个虚函数时，可以再一次使用 virtual 关键字指出该函数的性质，但是无论加不加 virtual，在派生类中它都是虚函数
- 同样派生类中的虚函数的返回类型也必须与基类函数匹配，但存在一个例外，当类的虚函数返回类型是**类本身的指针或引用时**，上述规则无效

3、final 和 override 说明符
- 派生类如果定义了一个函数与基类中的虚函数的名字相同但是行参列表不同，这仍然是合法的行为，因为编译器认为这个函数与基类原有的函数是完全独立的。
- 但如果是书写发生了错误，而本意是想要覆盖基类中的虚函数，这就会导致问题，可以使用 override 来显式告知编译器该函数覆盖了基类中的某个虚函数，当书写错误时，编译器将报错
- 可以将函数定义为 fianl(写在在{之前)，来防止被覆盖，任何覆盖操作都将引发错误

4、虚函数与默认实参
- 和其他函数一样，虚函数也可以拥有默认实参
- 如果某次函数调用使用了默认实参，则该实参值由本次调用的静态类型决定
  - 如果以动态的方式调用该函数，那么该实参为基类版本中的实参，而无视派生类版本的实参
  - 如果以静态的方式调用，如果是派生类对象调用该函数，那么默认实参将会采用派生类中的默认实参
- 建议，如果虚函数使用默认实参，那么基类和派生类最好一致，免得引起误会

5、虚函数回避机制(仍然用引用或者指针的情况下)
- 某些情况下，我们希望对虚函数的调用不要进行动态绑定，而是强迫其使用某个指定的版本：
  - p->ClassName::func(args);
  - cite.ClassName::func(args);
- 如果派生类虚函数的定义代码中包含了对基类该虚函数的调用，必须加上"基类名::"，以免循环调用派生类的函数(Java 使用的是 super.func(name))

## 15.4. 抽象基类

1、纯虚函数
- 通过在分号之前书写"=0"将一个虚函数定义为纯虚函数
  - T1 func(args) **=0**;

- 我们可以为纯虚函数提供定义，但函数体必须在类外，也就是说，我们不能在类的内部为一个=0 的函数提供定义

2、抽象基类

- 含有纯虚函数的类是抽象基类
- 抽象基类负责定义类的接口，而后续的其他类覆盖该接口(实现它)
- 我们不能直接创建抽象基类的对象
- 可以创建抽象基类的引用以及指针

## 15.5. 访问控制与继承

1、每个类分别控制自己的成员初始化过程

2、每个类分别控制着其成员对于派生类来说是否可访问

1、受保护的成员 protected

- 与私有成员类似，受保护的成员对于类的用户来说是不可访问的
- 与共有成员类似，受保护的成员对于派生类的成员和友元是可访问的
- **派生类的成员或友元只能通过派生类对象来访问基类的受保护的成员，派生类对于一个基类对象中的受保护成员没有任何访问权限**
- **派生类的成员和友元只能访问派生类对象中的基类部分的受保护成员，对于普通的基类对象中的成员不具有特殊的访问权限**
  - 派生类的成员和友元可以访问基类中公有和受保护的成员，因为该成员或友元也在派生类的作用域当中，而该派生类的作用域包含了基类的部分，这就是通过派生来访问基类中的受保护成员
  - Java 不同，protected 主要针对的是处于不同包中的派生类，对于同同一个包中的非 private 成员，均能访问，因为默认的访问权限是 friendly(包访问权限)

2、共有、私有和受保护继承

- 某个类对其继承而来的成员的访问权限收到两个因素的影响
  - 在基类中该成员的访问说明符
  - 在派生类的派生列表中的访问说明符
- 派生访问说明符对于派生类的成员(及友元)能否访问直接基类的成员没有影响
- 对于基类成员的访问权限只与基类的访问说明符有关
- ***派生访问说明符的目的：控制派生类用户(包括派生类的派生类)对于派生类中的基类部分的访问权限***

3、**派生类向基类转换的可访问性(转换的可访问性)**

- 派生类向基类的转换是否可访问由使用该转换的代码决定，同时派生类的派生访问说明符也会有影响
- 假定 D 继承 B
- 只有当 D 以 public 继承 B 时，用户代码才能使用派生类向基类的转换
- 无论 D 以什么方式继承 B，**D 的成员函数和友元**都能使用派生类向基类的类型转换
- 如果 D 继承 B 的方式是 public 或 protected 的，则 **D 的派生类的成员和友元**可以使用 D 向 B 的类型转换，若以私有方式，则不行

4、友元与继承

- 友元关系不能传递，友元关系也不能继承
- 基类的友元在访问派生类成员时不具有特殊性，但是可以访问派生类中的基类部分
- 派生类的友元也不能随意访问基类成员
- 每个类负责控制各自成员的访问权限
  - 该类的友元，能够访问该类的所有成员，以及该类的派生类中的属于该类的部分
  - 该类的友元，对于不属于该类的部分(例如派生类自己定义的额外成员)，没有访问权限

5、改变个别成员的可访问性

- 利用 using 声明可以达到这一目的
  - using Base::i;
- 派生类只能为那些它可访问的名字提供 using 声明，言下之意，不可改变基类中 private 的成员的可访问性

6、默认的继承保护级别

- 使用 class 关键字定义的派生类是私有继承的
- 使用 struct 关键字定义的派生类是公有继承的

### 15.5.1. 成员访问说明符与派生访问说明符的理解

1、成员访问说明符

- *控制该类的用户(该类的派生类以及该类的对象)对该类成员的访问权限*
- *言下之意，成员访问说明符无法控制该类的友元以及成员函数对于该类的成员的访问权限，该类的友元与成员函数对于该类的访问权限是至高的*

2、派生访问说明符

- *控制派生类的用户(派生类的派生类，或者派生类的对象)对于派生类中基类部分的访问权限*
- *言下之意，派生访问说明符无法控制派生类的友元以及成员函数对于派生类中的基类部分的访问权限，这部分访问权限受到基类的成员访问说明符的控制*

## 15.6. 继承中的类作用域

1、每个类定义自己的作用域

- *当存在继承关系时，派生类的作用域嵌套在基类作用域之内(这就是派生类含有基类的成员(无论是否有访问权限))*
- 恰恰因为类作用域有继承嵌套的关系，派生类才能像使用自己的成员一样使用基类的成员(当然这些成员是否被派生类访问取决于基类的访问说明符)
- 如果一个名字在派生类的作用域内无法正确解析，编译器就将继续在外层的基类作用域中寻找该名字的定义

2、名字冲突与继承

- 和其他作用域一样，派生类也能重用定义在其直接基类或间接基类中的名

字，此时内层作用域(派生类)的名字将隐藏外层作用域(基类)的名字
- 可以使用作用域运算符(::)来使用一个被隐藏的名字
- 派生类最好不用重用其他定义在基类中的名字

3、函数调用过程的解析
- 对于 p->men()的调用
- 首先确定 p 的静态类型
- 在 p 的静态类型对应的类中查找 men，如果找不到，则依次在直接基类中不断查找直到达到继承链的顶端，如果找遍了该类以及基类仍然找不到，则编译器报错
- 一旦找到了 men，就进行常规的类型检查，以确认对于当前找到的 men，本次调用是否合法
- 假设调用合法，则编译器将根据调用的是否是虚函数产生不同代码
  - 如果 men 是虚函数且我们通过引用或指针进行的调用，则编译器产生的代码将在运行时确定到底运行该虚函数的哪个版本，依据是对象的动态类型
  - 反之，如果 men 不是虚函数或我们通过对象调用，则编译器将产生一个常规的函数调用(与 Java 不同，因为在 Java 中，只能创建类类型的引用，因此全部都是动态调用)

4、覆盖重载的函数
- 和其他函数一样，成员函数无论是否是虚函数都能被重载
- 派生类可以覆盖重载函数的 0 个或多个实例
- 如果派生类希望所有的重载版本对它来说都是可见的，必须覆盖所有的版本，或者一个都不覆盖
  - **内层作用域的函数并不会重载声明在外层作用域的函数，在不同的作用域无法重载函数名，在内层作用于声明名字将隐藏外层作用域的同名实体(即使形参列表不一致，因为名字查找优先于类型的检查)**
  - 因为如果只覆盖其中几个，那么只有被覆盖的函数才会在内层作用域(派生类)被查找到，**而这个重载函数名将会隐藏在基类中的其他版本**
- 我们可以为重载的成员提供一条 using 声明语句，这样就无须重载不需要重载的基类函数了，这条 using 语句将所有重载的实例添加到派生类的作用域当中

## 15. 7. 构造函数与拷贝控制

### 15. 7. 1. 虚析构函数

1、继承关系对基类拷贝控制最直接的影响是基类通常应该定义一个虚析构函数，这样就能动态分配继承体系中的对象了

2、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
- 因此将虚构函数定义成虚函数以确保执行正确的析构函数版本

3、与其他虚函数一样，析构函数的虚函数属性也会被继承

4、三五法则的例外：对于继承体系中的基类，都是需要定义析构函数的，但是

是否需要拷贝或者复制操作就不确定了
5、如果一个类定义了析构函数，即使它通过=default 的形式使用了合成的版本，编译器也不会为这个类合成移动操作

### 15. 7. 2. 合成拷贝控制与继承

1、基类或派生类的合成拷贝控制成员的行为与其他合成的构造函数，赋值运算符或析构函数类似

- 它们对类本身的成员依次进行初始化、赋值或销毁操作
- 此外这些合成的成员还负责使用直接基类中对应的操作对一个对象的直接基类部分进行初始化、赋值或销毁操作

### 15. 7. 3. 派生类的拷贝控制成员

1、派生类构造函数在其初始化阶段不但要初始化派生类自己的成员，还要负责初始化派生类对象的基类部分
2、派生类的拷贝和移动构造函数在拷贝和移动自有成员的同时，也要拷贝和移动基类部分的成员

- Drive(const Drive&d):Base(d)...{}
- Drive(Drive&& d):Base(std::move(d))...{}

3、派生类的赋值运算符也必须为其基类部分的成员赋值

- Drive& Drive::operator=(const Drive&rhs){Base::operator=(rhs);/*...*/}

4、析构函数只负责销毁派生类自己分配的资源，对象的成员是被隐式销毁的

- 基类的析构函数~Base 会被自动调用
- 对象销毁的顺序与创建顺序相反，派生类的析构函数首先执行，然后是基类的析构函数，直至集成体系的最后

5、如果构造函数或析构函数的函数体中调用了某个虚函数，则我们应该执行与构造函数或析构函数所属类型相对应的虚构函数版本(而不是与动态类型相同的版本)

- 情景：当基类的构造函数包含了对某个虚函数的调用
- 当构造派生类的对象时，动态类型是派生类
- 当初始化派生类中基类部分时，执行了基类函数体中的那个虚函数，此时这个虚函数的版本应该与它所属的那个构造函数类型一致，也就是基类的版本
- **这个虚函数在被调用的时候，基类部分已经初始化完毕，而派生类部分尚未初始化，如果调用的是派生类的版本，则可能访问派生类的未初始化的成员，这样会导致程序的崩溃**
- 对于其他函数，则调用的虚函数版本与动态类型一致

### 15. 7. 4. 继承的构造函数（略）

1、在 C++11 新标准中，**派生类能够重用其直接基类定义的构造函数，但需要明确，这些构造函数并非以常规的方式继承而来**
2、一个类只能初始化它的直接基类，处于同样的原因，一个类也只继承其直接基类的构造函数
3、派生类继承其基类构造函数的方式是提供一条注明了基类名的 using 语句

- using Base::Base

- 对于基类的每个构造函数，编译器都生成一个与之对应的派生类的构造函数
- 换句话说，对于基类的每个构造函数，编译器都在派生类中生成一个形参列表完全相同的构造函数
- 如果派生类含有自己的成员，这些成员默认初始化

## 15.8. 容器与继承

1、当我们使用容器存放继承体系中的对象时，通常必须采取间接存储的方式，因为不允许保存不同类型的元素

2、**在容器中放置指针而非对象，注意，容器不能存放引用**

3、**实现动态的方式有两种，引用和指针，但是结合容器实现动态只有指针**

## 15.9. 派生类中重载方法

1、**C++如果在派生中定义与基类名字一样的函数(注意，仅仅名字一样)，那么该函数就会屏蔽所有从基类继承而来的同名函数**

# Chapter 16. 模板

1、模板是 C++中范型编程的基础
2、一个模板就是一个创建类或函数的蓝图或者说公式

## 16.1. 定义模板

### 16.1.1. 函数模板

1、我们可以定义一个通用的函数模板，而不是为每个类型都定义一个新函数，一个函数模板就是一个公式，可以用来生成针对特定类型的函数版本
2、模板定义以关键字 template 开始，后跟一个模板参数列表(这是一个逗号分隔的一个或多个模板参数，列表用<>号包围起来)
3、在模板定义中，列表参数不能为空
4、模板的参数列表很像函数的参数列表
5、模板参数表示在类或函数定义中用到的类型或值，当使用模板时，我们隐式地或显式地指定模板实参，将其绑定到模板参数上
6、实例化函数模板

- 当我们调用一个函数模板时，编译器(通常)用函数实参来为我们推断模板实参
- 编译器用推断出的模板参数来为我们实例化一个特定版本的函数

7、模板类型参数

- 我们可以将类型参数看做类型说明符，就像内置类型或类类型说明符一样使用
- 类型参数可以用来指定返回类型或函数的参数类型，以及在函数体内用于变量声明或类型转换
- 类型参数前必须加上关键字 class 或 typename，推荐用 typename
- template <typename T>

8、非类型模板参数

- 我们还可以在模板中定义非类型参数，一个非类型参数表示一个值而非一个类型
- 当一个模板被实例化时，非类型参数被一个用户提供的货编译器推断出的值锁代替，这些值必须是常量表达式，从而允许编译器在编译时实例化模板
- 一个非类型参数可以是一个整型，或是一个指向对象或函数类型的指针或引用。
  - 绑定到非类型整型参数的实参必须是一个常量表达式
  - 绑定到指针或引用非类型参数必须具有静态的生命周期
  - 在模板定义内，模板非类型参数是一个常量值，在需要常量表达式的地方，可以使用非类型参数，例如指定数组大小

9、inline 和 constexpr 的函数模板

- 函数模板可以声明为 inline 或 constexpr 的
- inline 和 constexpr 说明符放在模板参数列表之后，返回类型之前

10、模板程序应该尽量减少对模板类型的要求：比如尽量不要在模板函数体内调用模板参数类型的某个特定的方法

11、模板编译

- 当编译器遇到一个模板定义时，它并不生成代码，只有当我们实例化出模板的一个特定版本时，编译器才会生成代码
- 当我们使用而不是定义模板时，编译器才生成代码，这个特性影响了我们如何组织代码以及错误何时被检测到
- **通常当我们调用一个函数时，编译器只需要掌握函数的声明，类似的，当我们使用类类型对象时，类定义必须是可用的，但是成员函数的定义不必已经出现，因此我们可以将类定义和函数声明放在头文件中，而普通函数和类的成员函数的定义放在源文件中**
- **模板则不同，为了生成一个实例化版本，编译器需要掌握函数模板或类模板成员函数的定义，因此与非模板代码不同，模板的头文件通常既包括声明也包括定义**

12、大多数编译错误在实例化期间报告

- 第一个阶段是编译模板本身时，在这个阶段，编译器通常不会发现很多错误，编译器可以检查语法错误，例如忘记分号或者变量名拼写错误
- 第二个阶段是编译器遇到模板使用时，在这个阶段，编译器仍然没有很多可以检查，对于函数模板的调用，编译器通常会检查实参数目是否正确，它还能检查参数类型是否匹配，对于类模板，编译器可以检查用户是否提供了正确数目的模板参数，仅限于此
- 第三个阶段是模板实例化时，只有这个阶段才能发现类型相关的错误，依赖于编译器如何管理实例化，这类错误可能在链接时才报告

13、保证传递给模板的实参支持模板所要求的操作，以及这些操作在模板中能正确工作，是调用者的责任

```
template<typename T>
void f(const T& t){
    t.func(); //调用者要保证，传递给模板的实参支持这种调用
}
```

### 16. 1. 2. 类模板

1、类模板是用来生成类的蓝图的，与函数模板不同的是，编译器不能为类模板推断模板参数类型

2、为了使用模板，我们必须在模板明后的尖括号中提供额外信息

3、定义类模板

- 类似函数模板，类模板以关键字 template 开始，后跟模板参数列表
- 在类模板(及其成员)的定义中，我们将模板参数当做替身，代替使用模板时用户所需要提供的类型或值

4、实例化类模板

- 当使用一个类模板时，我们必须提供额外的信息，我们现在知道这些额外的信息是显式模板实参列表，它们被绑定到模板参数，编译器使用这些模板实参来实例化出特定的类
- 一个类模板的每个实例都形成一个独立的类

5、在模板作用域中引用模板类型

- 一个类模板中的代码如果使用了另外一个模板，通常不将一个实际类型(或值)的名字用作其模板实参，相反，我们通常将模板自己的参数当做被

使用的模板实参???

6、类模板的成员函数

- 与其他任何类相同，我们既可以在类模板内部，也可以在类模板外部定义成员函数，且定义在类模板内的成员函数被隐式声明为内联函数
- 类模板的成员函数本身是一个普通函数，但是类模板的每个实例都有自己版本的成员函数，因此类模板的成员函数具有和模板相同的模板参数
- 因此，定义在类模板之外的成员函数就必须以关键字 template 开始，后接模板参数列表

7、类模板成员函数的实例化

- 默认情况下，一个类模板的成员函数只有当程序用到它时才进行实例化
- **如果一个成员函数没有被使用，则它不会被实例化，这一特性使得即使某种类型不能完全符合模板操作的要求，我们仍然能使用该类**
  - 一个成员函数调用了 T 类型的 func1 函数，另一个成员函数调用了 T 类型的 func2 函数，我们可以传入一个类型，它只定义了 func1 函数而没有定义 func2 函数

8、在类代码内简化模板类名的使用

- 当我们使用一个类模板时必须提供模板实参，但这有一个例外，在类模板自己的作用域中，我们可以直接使用模板名而不提供实参，比如需要返回一个定义类型本身的参数，可以直接写成 ClassName，而不必写成 ClassName<T>

9、在类模板外使用类模板名

- 当我们在类模板外定义其成员时，我们并不在类的作用域中，直到遇到类名才表示进入类的作用域
- template<typename T>
- returnType ClassName<T>::func(args){/*something*/}  //这里必须提供模板**实参**
  - typename 后跟的 T 是模板参数
  - ClassName<>中的 T 是模板实参!!!

10、类模板和友元

- 当一个类包含一个友元声明，类与友元各自是否包含模板是互相无关的
- 如果一个类模板包含一个非模板友元，则友元被授权可以访问所有模板实例，如果友元自身是模板，类可以授权给所有友元模板实例，也可以授权给特定实例
- 一对一好友关系
  - 类模板与另一个模板间友好关系最常见的形式是建立实例及友元间的友好关系
  - 为了引用(类或函数)模板的一个特定实例，我们必须首先声明模板自身

```
template<typename> class F;//必须要前置声明
template<typename T> class A{
    friend class F<T>;//为类模板 F 提供模板实参，需要前置声明
}
```

- 通用和特定的模板友好关系
  - 一个类也可以将另一个模板的每个实例都声明为自己的友元，或者限定

特定的实例为友元

```
template <typename T> class F1;
class C1{
    friend class F1<C1>;//需要前置声明，因为提供了模板实参
    template <typename T> friend class F2;//不需要前置声明
}

template <typename T> class F1;
template<typename T> class C2{
    friend class F1<T>;//需要前置声明，因为提供了模板实参
    template<typename X> friend class F2;//不需要提供前置声明，但是由于 C2 是类模板，已经占用了 T 符号，因此必须换一个，这里用的 X
    friend class F3;
}
```

- 让所有两个类模板的所有实例互为友元，在声明友元时，必须提供不同的模板参数

11、模板令自己的类型参数成为友元

- 新标准中，可以将模板类型参数声明为友元

12、模板类型别名

- 类模板的一个实例定义了一个类类型，与任何其他类类型一样，我们可以定义一个 typedef 来引用实例化的类
  - typedef vector<int> vint;
  - using vint=vector<int>;
- 由于模板不是一个类型，我们不能定义一个 typedef 引用一个模板
  - ~~typedef vector<T> vT; //错误~~
  - ~~using vT=vector<int>; //错误~~
- 新标准允许我们为类模板定义个类型别名，也可以固定一个或多个模板参数
  - template<typename T> using twin=pair<T, T>
  - template<typename T> using partint=pair<T,int>

13、类模板的 static 成员

- 类模板可以声明 static 成员
- 与任何其他 static 数据成员相同，模板类的每个 static 数据成员必须有且仅有一个定义
- **但是类模板的每个实例都有一个独有的 static 对象，因此与定义模板的成员函数类似，我们将 static 数据成员也定义为模板**
  - template<typename T>
  - int C<T>::i=0;
  - 因此当用一个特定的模板实参类型实例化 C 时，会为该类类型实例化一个独立的 i，并将其初始化为 0
- 类似任何其他成员函数，一个 static 成员函数只有在使用时才会被实例化

### 16.1.3. 模板参数

1、类似函数参数的名字，一个模板参数的名字没有什么内在意义，通常将模板

参数命名为 T，但实际上可以使用任何名字

2、模板参数与作用域

- 模板参数遵循普通的作用域规则
- 一个模板参数名的可用范围在其声明之后，至模板声明或定义结束前
- 与其他任何名字一样，模板参数会隐藏外层作用域中声明的相同名字
- 但是与大多数上下文不同，在模板内不能重用模板参数名

3、模板声明

- 模板声明必须包含模板参数
- 与函数参数相同，声明中的模板参数名字不必与定义中相同
- 一个给定不安的每个声明和定义必须有相同数量和种类(即类型和非类型)的参数

4、使用类的类型成员

- 我们使用作用域运算符::来访问 **static** 和类型成员
- 在普通(非模板)代码中，编译器掌握类的定义，因此它知道通过作用域运算符访问的是名字还是 static 成员，不会有问题
- 但对于模板代码就存在困难，假定 T 是一个模板类型参数，当编译器遇到类似 T::men 这样的代码时，它不会知道 men 是一个类型成员还是一个 static 数据成员，直到实例化时才会知道，但是为了处理模板，编译器必须知道名字是否表示一个类型
  - 例如 T::size_type*p;
  - 编译器需要知道我们是在定义一个名为 p 的变量，还是将一个名为 size_type 的 static 数据与名为 p 的变量相乘
- 默认情况下，C++假定通过作用域运算符访问的名字不是类型，因此如果我们希望使用一个模板类型参数的类型成员，就必须现显式告诉编译器该名字是一个类型，使用 typename 来实现这一点
  - typename T::value_type ...

5、默认模板实参

- 新标准中，我们可以为函数和类模板提供默认模板实参
- 与函数默认实参一样，对于一个模板参数，只有当它右侧的所有参数都有默认实参时，它才可以有默认实参

6、模板默认实参与类模板

- 无论何时使用一个类模板时，我们都必须在模板名后接上尖括号
- 如果一个类模板为其所有模板参数提供了默认实参，且我们希望使用这些默认实参，必须在模板名后跟一个空尖括号<>(与 Java 空<>意义不同！)

### 16. 1. 4. 成员模板

1、一个类(无论是普通还是类模板)可以包含本身是模板的成员函数，这种成员被称为是成员模板，成员模板不能是虚函数

2、普通(非模板)类的成员模板：没什么特别的，就跟定义函数模板一样定义

3、类模板的成员模板

- 我们可以为类模板定义成员模板，在此情况下，类和成员各自有自己的独立的模板参数
- 与类模板的普通函数成员不同，成员模板是函数模板，我们在类模板外定义一个成员模板时，必须同为为类模板和成员模板提供模板参数列表，类

模板的参数列表在前，后跟成员自己的模板参数列表

- template<typename T>
- template<typename X>
- returnType C<T>::func(args){/*do something*/}

4、实例化与成员模板

- 为了实例化一个类模板的成员模板，我们必须同时提供类和函数模板的实参
- 与往常一样，我们在哪个对象上调用成员模板，编译器就根据对象的类型来推断类模板参数的实参
- 与普通函数模板相同，编译器通常根据传递给成员模板的函数参数来推断它的模板实参

### 16.1.5. 控制实例化

1、当模板被使用时才会进行实例化，这一特性意味着，相同的实例可能出现在多个对象文件中，当两个或多个独立编译的源文件使用了相同的模板，并且提供了相同的模板参数时，每个文件中都会有该模板的一个实例

2、在大系统中，在多个文件中实例化相同模板的额外开销非常严重，在新标准中，我们可以通过显式实例化来避免这种开销

- extern template *declaration* ***//实例化声明(含有 extern)***
- template *declaration* ***//实例化定义(没有 extern)***
- *declaration* 是一个类或函数的声明，其中所有的模板参数已被替换为模板实参
- extern template class **std::**vector<int>; **//Xcode 不知道为什么必须写 std::，即使有 using 指示，VS 上可以略去 std**
- template int compare(const int&,const int&);
- 当编译器遇到 extern 模板声明时，不会在本文件中生成实例化代码
- 将一个实例化声明为 extern 就表示承诺在程序其他位置有该实例化的一个非 extern 声明(定义)
- 对于一个给定的实例化版本，可以有多个 extern 声明，但必须只有一个定义
- 对于每个实例化声明，在程序中每个位置必须有其显式实例化定义
- **一个类模板的实例化定义会实例化该模板的所有成员，包括内联的成员函数，当编译器遇到一个实例化定义时，它不了解程序使用哪些成员函数，因此与处理类模板的普通实例化不同，编译器会实例化该类的所有成员**
- 因此我们用于显式实例化一个类模板的类型，必须能用于模板的所有成员

### 16.1.6. 效率与灵活性

1、shared_ptr 与 unique_ptr 之间的明显不同是它们管理所保存的指针的策略---前者给予我们共享指针所有权的能力，后者独占指针

2、shared_ptr 与 unique_ptr 之间另一个差异是它们允许用户重载默认删除器的方式

- 我们可以很容易地重载一个 shared_ptr 的删除器，只要在创建或 reset 指针时传递给他一个可调用对象即可
- 与之相反，删除器类型是一个 unique_ptr 对象的类型一部分，用户必须在

定义时以显式模板实参的形式提供删除器的类型，因此对于 unique_ptr 的用户来说，提供删除器就更为复杂

3、在运行时绑定删除器：shared_ptr

- shared_ptr 必须能直接访问其删除器，即删除器必须保存为一个指针或一个封装了指针的类
- 我们可以确定 shared_ptr 不是将删除器直接保存为一个成员，因为删除器的类型是运行时才知道，实际上，在一个 shared_ptr 的生存周期中，我们可以随时改变其删除器的类型
- 考察删除器如何工作：
  - 假定 shared_ptr 将它管理的指针保存在一个成员 p 中，且删除器是通过一个名为 del 的成员来访问的，则 shared_ptr 的析构函数必须包含类似下面的语句
  - del==nullptr? del(p):delete p; //del 的值只有在运行时才知道，通过一个指针来调用它(del(p)时需要跳转到 del 的地址)
  - 由于删除器是间接保存的，调用 del(p)需要一次运行时的跳转操作，转到 del 中保存的地址来执行对应的代码

4、在编译时绑定删除器：unique_ptr

- 在 unique_ptr 这个类中，删除器的类型是类类型的一部分，即 unique_ptr 有两个模板参数，一个表示它所管理的指针，另一个表示删除器的类型
- 由于删除器的类型是 unique_ptr 的一部分，因此删除器成员的类型在编译时时知道的，从而删除器可以直接保存在 unique_ptr 对象中
- unique_ptr 的析构函数与 shared_ptr 的析构函数类似，也是对其保存的指针调用用户提供的删除器或执行 delete
  - del(p);//del 在编译时绑定，直接调用实例化的删除器，无运行时额外开销

5、通过在编译时期绑定删除器，unique_ptr 避免了间接调用删除器的运行时开销，通过运行时绑定删除器，shared_ptr 使用户重载删除器更为方便

## 16.2. 模板实参推断

1、对于函数模板，编译器利用调用中的函数实参来确定其模板参数，从函数实参类确定模板实参的过程被称为**模板实参推断**

2、在模板实参推断过程中，编译器利用函数调用中的实参类型来寻找模板实参，用这些模板实参生成的函数版本与给定的函数调用最为匹配

### 16.2.1. 类型转换与模板类型参数

1、我们在一次调用中传递给函数模板的实参被用来初始化函数的形参，**如果一个函数的形参类型使用了模板参数类型，那么它就采用特殊的初始化规则，只有有限的几种类型转换会自动应用于这些实参**，编译器通常不对实参进行类型转换，而是生成一个新的模板实例

2、与往常一样，顶层 const 无论是在形参中还是在实参中，都会被忽略

3、会发生的类型转换：

- 底层 const 转换：可以将一个非 const 对象的引用(或指针)传递给一个 const

的引用(或指针)

- 数组或函数指针转换：如果函数形参不是引用类型，则可以对数组或函数类型的实参应用正常的指针转换，一个数组实参可以转换为一个指向其首元素的指针，类似的，一个函数实参可以转换为一个该函数类型的指针

4、其他类型转换，包括算数类型转换，派生类向基类的转换以及用户定义的转换都不能应用于模板

5、正常类型转换应用于普通函数实参

### 16. 2. 2. 函数模板显式实参

1、某些情况下，编译器无法推断出模板实参的类型(当函数返回类型与参数列表中任何类型都不同时)，我们希望允许用户控制模板实例化

2、指定显式模板实参

- 提供显式模板实参的方式与定义类模板实例的方式相同
- 显式模板实参在尖括号中给出，位于函数名之后，实参列表之前**(Java 中的泛型函数定义时，泛型参数在返回类型之前给出)**
- fuc<int>(i,j);
- 模板实参按从左到右的顺序与对应的模板参数匹配，只有尾部参数的显式模板实参才可以被忽略，而且前提是它们可以从函数参数列表推断出来

3、正常转换应用于显式指定的参数

- 对于普通类型定义的函数参数，允许进行类型转换，处于同样的原因，对于模板类型参数已经显式指定了函数实参，也能进行正常的类型转换

### 16. 2. 3. 尾置返回类型与类型转换

1、当我们希望用户确定返回类型时，显式模板实参表示模板函数的返回类型是很有效的，但在其他情况下，要求显式指定模板实参会给用户增添额外的负担，而且不会带来什么好处

2、例如这样的函数

```
template<typename It>
??? & func(It beg,It end){
    //处理代码
    return *beg
}
```

- 我们并不知道返回结果的准确类型，但知道所需类型是所处理序列元素的类型

```
template<typename It>
auto func(It beg,It end) -> decltype(*beg) {
//处理代码
return *beg
}
```

3、进行类型转换的标准库模板类

P606

### 16. 2. 4. 函数指针和实参推断

1、当我们用一个函数模板初始化一个函数指针或为一个函数指针赋值时，编译

器使用指针的类型来推断模板实参

2、当参数是一个函数模板实例的地址时，程序上下文必须满足：对每个模板参数，能唯一确定其类型或值

```
template <typename T> int compare(const T&,const T&);
int (*pf1) (const int&,const int&) =compare;
```

### 16. 2. 5. 函数模板实参推断和引用

1、对于"template <typename T> void f(T &p);"编译器会应用正常的引用绑定规则：

- const 是底层的不是顶层的

2、从左值引用函数参数推断类型

- 当一个函数模板的参数类型是 T&
  - 正常的绑定规则告诉我们，只能传递给它一个左值(变量或返回引用的表达式)
  - 当实参是 const 类型，则函数模板推断出的模板参数类型 **T=const X**
  - 当实参不是 const 类型，则函数模板推断出的模板参数类型 **T=X**
- 当一个函数模板的参数类型是 const T&
  - 正常的绑定规则告诉我们，可以传递给它任何类型的实参：常量左值(const 对象)、非常量左值(非 const 对象)、常量右值(字面值)、非常量右值(临时对象)
  - 函数模板推断出的模板参数类型 **T=X**
  - **当函数参数本身是 const 时，T 的类型推断不会是一个 const 类型**，因为 const 已经是函数参数类型的一部分，因此它不会是模板参数类型的一部分
    - **唯一特例：当实参类型是指向常量的指针时，模板参数推断会保留 const ==>T=const X***
- *无论是 T&还是 const T&，T 的推断都不会带有&，但是最终的形参还是带有&的，这里要注意，因为形参的类型是 T&或 const T&而不是 T*

3、从右值引用函数参数推断类型

- 当一个函数模板的参数类型是 T&&
  - 正常的绑定规则告诉我们，可以给他传递一个右值(右值常量以及非常量(字面值类型))
- 引用折叠
  - C++在正常绑定规则之外定义了两个例外规则，这两个例外是 move 这种标准库设施正确工作的基础
    - 第一个例外绑定规则影响右值参数的推断如何进行：当我们将一个左值传递给**函数模板的右值引用参数(普通的右值引用是不行的)**，编译器推断模板类型参数为实参的左值引用类型
    - 第二个例外绑定规则允许我们**通过类型别名**或**通过模板类型参数**间接定义一个引用的引用：这些引用形成了"折叠"
      - X& &、X& &&、X&& & ==> X&
      - X&& &&==>X&&
  - 因此我们可以将任意类型传递给 T&&类型的参数
  - 如果实参是一个**非常量左值**，则函数模板推断出的模板参数类型 **T=X&**，**从而使得形参的类型为 X& &&==>X&，将本作为变量表达式的左值属性**

添加到了形参的类型中

- 如果实参是一个常量左值，则函数模板推断出的模板实参为 T=const X&，从而使得形参类型为 const X& &&==>const X&，将本作为变量表达式的左值属性添加到了形参的类型中
- 如果实参是一个常量右值或非常量右值，则函数模板推断出的模板实参为 T=X，从而使得形参类型为 X &&==> X&&，将本作为变量表达式的右值属性添加到了形参的类型中
- 注意讨论引用没有意义，因为引用是一个左值

4、C++标准规定，在推导模板参数时，如果函数形参是引用类型(T&、const T&、T&&)，那么推倒模板参数不会执行隐式转换，包括数组和函数的退化。

### 16.2.6. 理解 std::move

1、标准库是这样定义 move 的：

```
template<typename T>
typename remove_reference<T>::type&& move(T&& t){
    return static_cast<typename remove_reference<T>::type&&>(t);
}
```

- 首先 move 函数参数 T&&是一个指向模板类型参数的右值引用
  - 当传递左值时(常量或非常量)，推断类型为 T=X&或 T=const X&
  - 当传递给右值时，推断类型为 T=X
- 通过 typename remove_reference 去除引用属性
  - 得到 X 或 const X
- 最后通过 static_cast 强制转换为右值引用类型

2、通常情况下，static_cast 只能用于其他合法的类型转换，但存在一条针对右值引用的特许规则：

- 虽然不能隐式地将一个左值转换为右值引用，但可以用 static_cast 显式地将一个左值转换为一个右值引用

### 16.2.7. 转发

1、某些函数需要将其一个或多个实参不变地(包含实参所有的信息)转发给其他函数，在此情况下，我们必须要保证被转发的实参的所有信息

1) 实参类型(不包括引用类型)
2) 是否 const
3) 实参表达式的左右值属性

2、通过将一个函数参数定义为一个指向模板类型参数的右值引用(T&&)，我们可以保持其对应实参的所有信息

3、概念

- 表达式的属性：左值/右值
- 变量本身的属性(变量可以看做一个表达式)包括：左值/右值
- 类型信息包括：常量/非常量，引用/非引用，指针/非指针，以及类型
  - 其中只有引用可以分为左值引用或右值引用，这里的左右值属性与表达式的左右值属性完全不同
  - 引用的左右值属性表示的是，可以绑定到左值或右值的表达式
  - 表达式的左右值属性表示的是表达式本身的左右值属性

4、在调用中使用 std::forward 保持类型信息，与 std::move 一样定义在<utility>中

- ***forward 返回该显式模板实参类型的右值引用，即 forward<Type>返回 Type&&***
  - 若 Type 是 X&类型，那么 Type&&是 X& &&=X&类型
  - 若 Type 是 X&&类型，那么 Type&&是 X&& &&=X&&类型

5、例子说明

```
template<typename F, typename T>
void func(F f, T&& t){
    f(std::forward<T>(t));
}
```

1) 首先将实参的所有信息保存到形参 t 中，包括类型(不包括引用类型)，const 属性，实参表达式的左右值属性
2) 通过 std::forward<T>(t)将保存在形参中的左右值引用类型变成表达式的左右值属性

- 如此一来 std::forward<T>(t)所含的信息与实参完全相同
- **如果没有 std::forward<T>()函数，那么 f(t)便会调用 f 的左值引用版本**

## 16.3. 重载与模板

1、函数模板可以被另一个模板或一个普通非模板函数重载

2、涉及函数模板时，函数的匹配规则：

- 对于一个调用，其候选函数包括所有模板实参推断成功的函数模板实例
- 候选的函数模板总是可行的，因为模板实参推断会排除任何不可行的模板
- 可行函数按类型转换来排序，当然用于函数模板调用的类型转换非常有限(const 以及数组函数指针)
- 如果恰有一个函数提供比任何其他函数都要更好的匹配，则选择此函数
- 如果有多个函数提供同样好的匹配：
  - 如果同样好的的函数中只有一个非模板函数，则选择此函数
  - 如果同样好的函数中没有非模板函数，但有多个函数模板，且其中一个模板比其他模板更特例化，则选择此模板
  - 否则产生二义性

3、简而言之，匹配同样好的情况下

- 优先级排序：非模板>特例化的函数模板>函数模板

## 16.4. 可变参数模板

1、一个可变参数模板就是一个接受可变数目参数的模板函数或模板类

2、可变数目的参数被称为参数包

- 模板参数包：表示零个或多个模板参数
- 函数参数包：表示零个或多个函数参数
  - template<typename T, typename...Args>
  - void func(const T&t, const Args&...rest);

3、我们可以用 sizeof...运算符计算包中的元素个数，类似 sizeof，sizeof...也返回常量表达式

### 16.4.1. 编写可变参数函数模板

### 16.4.2. 扩展包

1、对于一个参数包，除了获取其大小外，我们能对它做的唯一的事情就是扩展
2、当扩展一个包的时候，我们还要提供用于每个扩展元素的模式
3、扩展一个包就是将它分解为构成的元素，对每个元素应用模式，获得扩展后的列表
4、我们通过在模式右边放一个省略号来触发扩展操作
5、理解扩展包

- print(os,rest...);//模式就是直接解开
- print(os,debug_rep(rest)...);//模式就是对每个元素调用 debug_rep()
- print(os,debug_rep(rest...));//模式就是直接展开

### 16.4.3. 转发参数包

1、在新标准下，我们可以组合使用**可变参数模板**与 **forward 机制**来编写函数，实现将其实参不变地传递给其他函数
2、例子，emplace_back(args)的实现

```
template<typename...Args>
void Vec::emplace_back(Args&&...args){
        alloc.construct(first_free++,std::forward<Args>(args)...);
}
```

- 这里的模式就是：对每个元素调用 std::forward<T>()

## 16.5. 模板特例化

1、某些情况下，通用模板的定义对特定类型是不适合的，通用定义可能编译失败或者不正确，这时我们可以定义类或函数模板的一个特例化版本
2、定义函数模板特例化

- 当我们特例化一个函数模板时，必须为原模板中每个模板参数都提供实参
- 为了指出我们正在实例化一个模板，应使用关键字 template 后跟一个空尖括号<>

3、类模板特例化

# Chapter 17. 标准库特殊设施

## 17.1. tuple 类型

1、定义在<tuple>中
2、用法
- tuple<T1,T2,...> t
- make_tuple(v1,v2,...,vn)
- get<dex>(t)
- get 是一个函数模板，必须制定显式模板实参，表明想要访问第几个成员

## 17.2. bitset 类型

1、bitset 是一个类模板，我们定义 bitset 时必须声明它包含多少个二进制位
2、用法
- bitset<n> b;
- bitset<n> b(u); //b 是 unsigned long long 值 u 的低 n 位的拷贝，如果 n 大于 u 的大小，则超出部分为 0
- b.any()
- b.all()
- b.none()
- b.count()
- b.size()
- b.test(pos)
- b.reset(pos)
- b.reset()

## 17.3. 正则表达式

1、头文件：<regex>
2、正则表达式库组件
- regex：表示一个由正则表达式的类
- regex_match：将一个字符序列与一个正则表达式匹配(完全匹配返回 true)
- regex_search：寻找第一个与正则表达式匹配的子序列(有匹配项返回 true)
- regex_replace：使用给定格式替换一个正则表达式
- sregex_iterator：迭代器适配器，调用 regex_search 来遍历一个 string 中所有匹配的子串
- smatch：容器类，保存在 string 中搜索的结果
- ssub_match：string 中匹配的子表达式结果

3、regex_search 与 regex_match
- 如果整个输入序列与表达式匹配，则 regex_match 函数返回 true
- 如果输入序列中一个子串与表达式匹配，则 regex_search 函数返回 true

- 这两个函数的参数
  - (seq,m,r,mft)
  - (seq,r,mft)
  - 在字符序列 seq 中查找 regex 对象 r 中的正则表达式，seq 可以是一个 **string**，表示范围的一对迭代器以及一个指向空字符结尾的字符数组指针
  - m 是一个 match 对象，用来保存匹配结果的相关细节，m 和 seq 必须具有兼容性
  - mft 是一个可选的 regex_constants::match_flag_type 值

4、概念
- 子表达式：正则表达式中以"()"括起来的部分
- 子串：一个字符串的其中连续一部分

### 17. 3. 1. 使用正则表达式库

1、regex 的相关用法
- regex r(re)：re 表示一个正则表达式，可以是一个 string，一个表示字符范围的迭代器对，一个指向空字符结尾的字符数组的指针，一个字符指针和一个计数器，一个花括号包围的字符列表
- regex r(re,f)：f 指出对象如何处理的标志，默认为 ECMAScript
- r1=re：将 r1 中的正则表达式替换为 re，re 表示一个正则表达式，可以是另一个 regex 对象，一个 string，一个指向空字符结尾的字符数组的指针或者一个花括号包围的字符列表
- r1.assign(re,f)：与使用赋值运算符效果相同，可选标识符 f 与 regex 构造函数中相同
- r.mark_count()：r 中子表达式的数目
- r.flags()：返回 r 的标志集
- regex 指定的标志
  - icase：匹配时忽略大小写
  - nosubs：不保存匹配的子表达式
  - optimize：执行速度优先于构造速度
  - ECMAScript：使用 ECMA-262 指定的语法
  - basic：使用 POSIX 基本的正则表达式语法
  - extended：使用 POSIX 扩展的正则表达式语法
  - awk：使用 POSIX 版本的 awk 语言的语法
  - grep：使用 POSIX 版本的 grep 语法
  - egrep：使用 POSIX 版本的 egrep 语法

2、一个正则表达式的语法是否正确是在运行时解析的

3、如果我们编写的正则表达式存在错误，则在运行时标准库会抛出一个类型为 regex_error 的异常
- error_collate：无效的元素校对请求
- error_ctype：无效的字符类
- error_escape：无效的转义字符或无效的尾置转义
- error_backref：无效的向后引用
- error_back：不匹配的方括号("["或"]")

- error_paren：不匹配的小括号("("或")")
- error_brace：不匹配的花括号("{"或"}")
- error_badbracd："{}"中无效的范围
- error_range：无效的字符范围(如[z-a])
- error_space：内存不足，无法处理此正则表达式
- error_badrepeat：重复字符("*","?","+","{")之前没有有效的正则表达式
- error_complexity：要求的匹配过于复杂
- error_stack：栈空间不足，无法处理匹配

4、正则表达式类库

| 输入序列类型 | 使用正则表达式类 |
|---|---|
| string | regex、smatch、ssub_match、sregex_iterator |
| const char* | regex、cmatch、csub_match、cregex_iterator |
| wstring | wregex、wsmatch、wssub_match、wsregex_iterator |
| const wchar_t* | wregex、wcmatch、wcsub_match、wcregex_iterator |

### 17.3.2. 匹配与 Regex 迭代器类型

1、每个不同输入序列都有对应的特殊 regex 迭代器类型

2、sregex_iterator 操作，其他类似

- sregex_iterator it(b,e,r)：一个 sregex_iterator，遍历迭代器 b 和 e 表示的 string
  - 它调用 sregex_search(b,e,r)将 it 定位到输入中第一个匹配的位置
- sregex_iterator end：尾后迭代器
- *it、it->：根据最后一个调用 regex_search 的结果，返回一个 **smatch 对象的引用或一个指向 smatch 对象的指针**
- ++it，it++：从输入序列当前匹配位置开始调用 regex_search，前置返回递增后的迭代器，后置返回原值
- it1==it2
- it1!=it2

3、smatch 操作

- m.ready()：如果已经通过调用 regex_search 或 regex_match 设置了 m，则返回 true，否则返回 false，如果返回 false，那么对 m 的操作是未定义的
- m.size()：如果匹配失败，则返回 0，否则返回最近一次匹配的正则表达式中子表达式的数目
- m.empty()：若 m.size()为 0，则返回 true
- m.prefix()：一个 ssub_match 对象，表示当前匹配之前的序列
- m.suffix()：一个 ssub_match 对象，表示当前匹配之后的部分
- m.format(...)
- m.length(n)：第 n 个与子表达式匹配的子串的长度
- m.position(n)：第 n 个与子表达式匹配的子串距序列开始的距离
- m.str(n)：第 n 个与子表达式匹配的子串 string
- m[n]：对应第 n 个子表达式的 ssub_match 对象
- m.begin(),m.end()：表示 m 中 sub_match 元素范围的迭代器
- m.cbegin(),m.cend()：

4、例子

```
const regex BEAN_REGEX("<bean([^>]+)>([\\s\\S]*?)</bean>");

for (sregex_iterator it(content.begin(), content.end(), BEAN_REGEX), eof; it != eof;
++it) {
    string inside_info = it->operator[](1);
    string content_info = it->operator[](2)
    //...后续处理
}
```

### 17.3.3. 使用子表达式

1、正则表达式中的模式通常包含一个或多个子表达式(subexpression)，一个子表达式是模式的一部分，本身也具有意义，正则表达式语法通常用括号表示子表达式

2、子表达式的序号：该子表达式(不包括"()")前面的"("的个数

- AB(C)(D(E))
- group 0：ABCDE
- group 1：C
- group 2：DE
- group 3：E

3、子匹配操作，以 ssub_match 为例

- matched：public bool 数据成员，指出 ssub_match 是否匹配
- first、second：public 数据成员，指向匹配序列首元素和尾后位置的迭代器
- length()：匹配子序列的长度
- str()：返回一个匹配子序列的 string，若 matched 为 false，则返回空 string
- s=ssub：将 ssub_match 对象 ssub 转化为 string 对象 s，等价于 s=ssub.str()

### 17.3.4. 使用 regex_replace

1、例子

```
const regex ANNOTATION_REGEX("<!--[\\s\\S]*?-->");
content = regex_replace(content, ANNOTATION_REGEX, "");
```

- 除去 xml 文件中的注释<!--  -->
- 注意[\\s\\S]*可以匹配换行符，而.*不可以

## 17.4. 随机数

1、新标准出现之前，C 和 C++都依赖于一个简单的 C 库函数 rand 来生成随机数

2、问题：一些程序需要非均匀分布的随机数，程序员为了解决这些问题而试图转换 rand 生成的随机数范围、类型或分布时，常常引入非随机性

3、随机数引擎类和随机数分布类定义在头文件 random 中

### 17.4.1. 随机数引擎和分布

1、随机数引擎操作：

- Engine e
- Engine e(s);  //使用整型值 s 作为种子

- e.seed(s)
- e.min();  //此引擎可生成的最小值
- e.max();  //此引擎可生成的最大值
- Engine::result_type
- e.discard(u);  //将引擎推进 u 步，u 为 unsigned long long
- **对于大多数场合，随机数引擎的输出是不能直接用的，问题出在随机数的值范围通常与我们需要的不符合**

2、分布类型和引擎

- uniform_int_distribution<unsigned> u(0,9);
- default_random_engine e;
- u(e);
- 随机数发生器：分布对象和引擎对象的组合
- **一般来说如果在一个非全局作用域中定义分布类型和引擎类型的对象，要加上 static，使其成为静态的，可以保持随机数的状态，以免每次都生成一样的序列**

### 17. 4. 2.  其他随机数分布

1、生成随机实数

- 最常用但不正确的从 rand 获得一个随机浮点数的方法是用 rand()初一 RAND_MAX，这种方法不正确的原因是随机整数的精度通常地域随机浮点数，这样有一些浮点数值就永远不会生成了
- 新标准库设施定义了一个 uniform_real_distribution 类型的对象
- default_random_engine e;
- uniform_real_distribution<double> u(0,1);
- u(e);

2、分布类型的操作

- d(e);//用相同的 e 连续调用 e，会根据 d 的分布时类型生成一个随机数序列
- d.min();
- d.max()
- d.reset();//重建 d 的状态，随后 d 的使用不依赖于 d 已经生成的值，相当于重置一下种子

3、使用分布的默认结果类型

- **分布类型都是模板**，具有单一的模板类型参数，表示分布生成的随机数的类型，只有一个例外 bernoulli_distribution 类
- 每个分布模板都有默认实参
  - 生成浮点数的默认生成 double
  - 生成整型值的默认生成 int

4、生成非均匀分布的随机数

- lround 函数可以将浮点数四舍五入到最近的整数，包含了从浮点数到整数的类型转换(定义在头文件 cmath 中)
- normal_distribution<> n(4,1.5) //正态分布，均值 4，方差 1.5，生成 double
- bernoulli_distribution b; //默认 true 和 false 各自 0.5 的概率
- bernoulli_distribution b(p);//true 的概率是 p

5、分布介绍

- 均匀分布
  - **uniform_int_distribution<IntT> u(m,n)**
  - **uniform_real_distribution<RealT> u(x,y);**
- 伯努利分布
  - **bernoulli_distribution b(p) ==>以给定概率生成 true，默认 0.5**
  - binomial_distribution<IntT> b(t,p) ==>分布是按采样大小为整形值 t，概率为 p 生成的，t 的默认值为 1，p 的默认值为 0.5
  - geometric_distribution<IntT> g(p) ==>每次试验成功的概率为 p，默认 0.5
  - negative_binomial_distribution<IntT> nb(k,p) ==>k 此试验成功的概率为 p，k 默认 1，p 默认 0.5
- 泊松分布
  - **possion_distribution<IntT> p(x) ==>均值为 double 值 x 的分布**
  - **exponential_distribution<RealT> e(lam) ==>指数分布，参数 lambda 通过浮点数 lam 给出，lam 默认 1.0**
  - gamma_distribution<RealT> g(a,b) ==>alpha(形状参数)为 a，beta(尺度参数)为 b，两者默认 1.0
  - weibull_distribution<RealT> w(a,b) ==>形状参数为 a，尺度参数为 b 的分布，两者默认 1.0
  - extreme_value_distribution<RealT> e(a,b) ==>a 默认 0.0，b 默认 1.0
- 正态分布
  - **normal_distribution<RealT> n(m,s) ==>均值 m，标准差 s，m 默认 0.0，s 默认 1.0**

## 17.5. IO 库再探

### 17.5.1. 格式化输入与输出

1、标准库定义了一组**操纵符**来修改流的格式状态

- 一个操纵符是一个函数或是一个对象，会影响流的状态，并能用作输入或输出运算符的运算对象
- 类似输入和输出运算符，操纵符也返回它所处理的流的对象
- **当操纵符改变流的格式状态时，通常改变后的状态对所有后续 IO 都生效**

2、操纵符

- endl：输出一个换行符并刷新缓冲区
- 布尔值
  - boolalpha：使得 true/false 输出格式从 1/0 转为 true/false，其他的数字还是不变的，仅仅转变 true/false 的打印格式
  - noboolalpha：取消
- 进制控制
  - hex：十六进制
  - oct：八进制
  - dec：十进制
  - showbase：在输出中指出进制
  - noshowbase：取消

- 浮点数精度控制
  - 可以调用 IO 对象的成员 precision 或者使用 setprecision 操纵符
    - precision 成员是重载的，一个版本接受 int 值，将精度设置为此值，并返回旧精度值，另一个版本不接受参数，返回当前精度值
    - cout.precision(12);
    - cout<<setprecision(13)<<...
- 打印小数点
  - 一般情况下，当一个浮点值的小数部分为 0 时，不显示小数点
  - showpoint
  - noshowpoint
- 输出补白
  - setw：指定下一个数字或字符串的最小空间
  - left：左对齐
  - right：右对齐，这是默认格式
  - internal：控制负数的符号位置，它左对齐符号，右对齐值，用空格填满所有中间空间
  - setfill：允许指定一个字符代替默认的空格来补白输出
- 定义在 iomaip 中的操纵符
  - setfill(ch)
  - setprecision(n)
  - setw(w)
  - setbase(b)：将整数输出为 b 进制

# Chapter 18. 用于大型程序的工具

## 18.1. 异常处理

1、异常处理机制允许程序中独立开发的部分能够在运行时就出现的问题进行通信并作出相应的处理
2、异常使得我们能够将问题的检测与解决过程分离开

### 18.1.1. 抛出异常

1、在 C++语言中，我们通过抛出一条表达式来引发一个异常，被抛出的表达式的类型，以及当前调用链共同决定了哪段处理代码将被用来处理该异常
2、当执行一个 throw 时，跟在 throw 后面的语句将不再被执行，相反将程序的控制权从 throw 转移到与之匹配的 catch 模块
3、控制权转义的含义

- 沿着调用链的函数可能会提早退出
- 一旦程序开始执行异常处理代码，沿着调用链创建的对象将被销毁

4、栈展开

- 栈展开的过程沿着嵌套函数的调用链不断查找，直到找到了与异常匹配的 catch 子句或者退出主函数
- 一个异常如果没有被捕获，它将终止当前程序(调用标准库函数 terminate)
- 在栈展开过程中，位于调用链上的语句块可能会提前退出，块退出后，在这些块中的局部对象将被销毁

### 18.1.2. 捕获异常

1、catch 子句

- 子句中的异常声明看起来像是只包含一个形参的函数形参列表
- 声明的类型决定了处理代码所能捕获的异常类型，这个类型必须是完全类型，可以是左值引用，但不能是右值引用
- 如果 catch 接受的异常与某个继承体系有关，则最好将该 catch 的参数定义为基类引用

2、查找匹配的处理代码

- 与函数匹配不同，catch 查找匹配的未必是最佳匹配，而是第一个与异常匹配的 catch 子句，因此越是专门的 catch 越应该置于整个 catch 列表的前端
- catch 语句是按照其出现的顺序逐一进行匹配的
- 大多数的类型转换不允许，但允许以下几类：
  - 允许从非常量向常量转换
  - 允许从派生类向基类类型转换
  - 数组和函数允许转换成指针
  - 比模板参数推断时要宽松一点，比它多了派生类向基类的转换

3、重新抛出

- throw;

4、捕获所有异常的代码 catch(...)

## 18.2. 命名空间

### 18.2.1. 定义命名空间

1、语法

```
namespace std{
}
```

2、每个命名空间都是一个作用域

3、命名空间可以是不连续的，因此我们可以将几个独立的接口和实现文件组成一个命名空间，此时命名空间的组织方式类似于我们管理自定义类及函数的方式

- 命名空间的一部分成员的作用是定义类，以及生命作为类接口的函数及对象，这些成员应该置于头文件中，这些头文件包含在使用了这些成员的文件中
- 命名空间成员的定义部分应该置于另外的源文件中

4、在程序中某些实体只能定义一次：非内联函数，静态数据成员，变量等，命名空间中的名字也要满足这一要求，这种接口和实现分离的机制确保我们所需的函数和其他名字只定义一次，而只要是用到这些尸体的地方都能看到对于实体名字的声明

5、全局命名空间

6、嵌套的命名空间

- 定义在其他命名空间中的命名空间
- 使用时需要多个::运算符

7、内联命名空间

- 与普通嵌套的命名空间不同
- 内联命名空间中的名字可以被外层命名空间直接使用
- 内联命名空间的定义方式是在关键字 namespace 前面天剑 inline

8、未命名的命名空间

- 关键字 namespace 后紧跟花括号
- **未命名的命名空间中定义的变量拥有静态生命周期**
- **一个未命名的命名空间可以在某个给定的文件内不连续，但是不能横跨多个文件，每个文件定义自己的未命名的命名空间**
- 和其他命名空间不同，未命名的命名空间仅在特定的文件内部有效，其作用范围不会横跨多个不同的文件
- 如果未命名的命名空间在文件的最外层作用域中，那么该命名空间中的名字要与全局作用域中的名字有所区别

### 18.2.2. 使用命名空间成员

1、语法

- namespace_name::member_name

2、命名空间别名：

- namespace a=b;

3、using 声明：扼要概述

- 一条 using 声明语句一次只能引入命名空间中的一个成员

- using 声明引入的名字遵守与过去一样的作用域规则：它的有效范围从 using 声明的地方开始，一直到 using 声明所在的作用域结束为止，外层作用域的同名实体将被隐藏
- 未加限定的名字只能在 using 声明所在的作用域以及其内层作用域中使用
- 在有效作用域结束后，我们必须使用完整的经过限定的名字

4、using 指示

- using 指示与 using 声明不同，我们无法控制那些名字是可见的，因为会导入所有的名字
- using 指示可以出现在全局作用域、局部作用域和命名空间作用域中
- **using 指示具有将命名空间成员提升到包含命名空间本身和 using 指示的最近的作用域的能力**
- 头文件里别用 using 指示

## 18.3. 多重继承与虚继承

1、多重继承(multiple inheritance)是指从多个直接基类中产生派生类的能力。多重继承的派生类继承了所有父类的属性

### 18.3.1. 多重继承

```
class Bear: public ZooAnimal{...}
class Panda: public Bear, public Endangered{...}
```

1、在派生列表中可以包含多个基类

2、每个基类包含一个可选的访问说明符，如果访问说明符被忽略了，那么关键字 class 对应的默认访问说明符是 private，关键字 struct 对应的是 public

3、与只有一个基类的继承一样，多重继承的派生列表也只能包含已经被定义过的类，而且这些类不能是 final 的。对于派生类能够继承的基类个数，C++并没有进行特殊规定，但是在某个给定的派生列表中，同一个基类只能出现一次

4、**在多重继承关系中，派生类的对象包含有每个基类的子对象**

5、构造一个派生类的对象将同时构造并初始化它的所有基类子对象。与从一个基类进行的派生一样，多重继承的派生类的构造函数初始值也只能初始化它的直接基类

6、派生类的构造函数初始值列表将实参分别传递给每个直接基类，其中基类的构造顺序与派生列表中基类的出现顺序保持一致，而与派生类构造函数初始值列表中基类的顺序无关

7、在 C++11 新标准中，允许派生类从它的一个或几个基类中继承构造函数，但是如果从多个基类继承了相同的构造函数(形参列表完全相同)，则程序将产生错误。因此，如果一个类从它的多个基类中继承了相同的构造函数，则这个类必须为该构造函数定义它自己的版本

8、派生类的析构函数只负责清除派生类本身分配的资源，派生类的成员以及基类都是自动销毁的

### 18. 3. 2. 类型转换与多个基类

1、在只有一个基类的情况下，派生类的指针或引用能自动转换成一个可访问基类的指针或引用。多个基类的情况类似，我们可以令某个可访问基类的指针或引用直接指向一个派生类对象

2、与只有一个基类的继承一样，对象、指针和引用的静态类型决定了我们能够使用哪些成员

### 18. 3. 3. 多重继承下的类作用域

1、在只有一个基类的情况下，派生类的作用域嵌套在直接基类和间接基类的作用域中，查找过程沿着继承体系自底向上进行，直到找到所需的名字，派生类的名字将隐藏基类的同名成员

2、在多重继承的情况下，相同的查找过程在所有直接基类中同时进行，如果名字在多个基类中都被找到，则对该名字的使用将产生二义性(对于一个派生类来说，从它的几个基类中分别继承名字相同的成员是完全合法的，只不过在使用这个名字时必须明确指出它的版本)

### 18. 3. 4. 虚继承

1、尽管在派生列表中同一个基类只能出现一次，但实际上派生类可以多次继承同一个类，派生类可以通过它的两个直接基类分别继承同一个间接基类，也可以直接继承某个基类，然后通过另一个基类再一次间接继承该类

2、例如 IO 标准库的 istream 和 ostream 分别继承了一个共同的名为 base_ios 的抽象基类。该抽象基类负责保存流的缓冲内容并管理流的条件状态。iostream 是另外一个类，从 istream 和 ostream 直接继承过来，可以同时读写流的内容。因为 istream 和 ostream 都继承自 base_ios，所以 iostream 继承了 base_ios 两次，一次是通过 istream，另一次是通过 ostream

3、在默认情况下，派生类中包含有继承链上每个类对应的子部分，如果某个类在派生过程中出现了多次，则派生类中将包含该类的多个子对象

4、在 C++中，我们通过虚继承的机制解决上述问题。虚继承的目的是令某个类做出声明，承诺愿意共享它的基类。其中，共享的基类子对象称为徐积累，在这种机制下，不论虚基类在继承体系中出现了多少次，在派生类中都只包含唯一一个共享的虚基类子对象

5、虚派生只影响了从指定虚基类的派生类中进一步派生出的类，它不会影响派生类本身

6、在实际编程过程中，位于中间层次的基类将其继承声明为虚继承一般不会带来什么问题。通常情况下，使用虚继承的类层次是由一个人或一个项目组一次性设计完成的，对于一个独立开发的类来说，很少需要基类中的某个是虚基类

7、支持向基类的常规类型转换

8、不论基类是不是虚基类，派生类对象都能被可访问基类的指针或引用操作

9、在每个共享的虚基类中只有唯一一个共享的子对象，所以该基类的成员可以被直接访问，并且不会产生二义性。此外，如果虚基类的成员只被一条派生路径覆盖，则我们仍然可以直接访问这个被覆盖的成员。但是如果成员被多于一个基类覆盖，则一般情况下派生类必须为该成员定义一个新的版本，例如

```
struct B{
    int b;
```

```
    ...
};
struct D1: public virtual B{...};
struct D2: public virtual B{...};
struct D:public D1,public D2{...};
```

1) 如果 D1 和 D2 中都没有 x 的定义，则 x 将被解析为 B 的成员，此时不存在二义性，一个 D 对象中只含有 x 的一个实例
2) 如果 x 是 B 的成员，同时是 D1 和 D2 中某一个的成员，则同样没有二义性，派生类的 x 比共享虚基类 B 的 x 优先级更高
3) 如果在 D1 和 D2 中都有 x 的定义，则直接访问 x 将产生二义性

- 与非虚的多重继承体系一样，解决这种二义性问题最好的方法是在派生类中为成员自定义新的实例

### 18. 3. 5. 构造函数与虚继承

1、在虚派生中，**虚基类是由最底层的派生类初始化的(虚基类可以不是该派生类的直接基类，这样做的目的是为了避免多次初始化)**，如果采用普通规则，那么虚基类将会在多条继承路径上被重复初始化

2、在继承体系中的每个类都可能在某个时刻成为"最底层的派生类"，只要我们能创建虚基类的派生类对象，该派生类的构造函数就必须初始化它的虚基类

3、虚继承的对象的构造方式：含有虚基类的对象的构造顺序与一般的顺序稍有不同，首先使用提供给最底层派生类构造函数的初始值初始化该对象的虚基类子部分，接下来按照直接基类在派生列表中出现的次序依次对其进行初始化

4、一个类可以有多个虚基类，这些虚的子对象按照它们在派生列表中出现的顺序从左向右依次构造

# Chapter 19. 特殊工具与技术

## 19.1. 控制内存分配

### 19.1.1. 重载 new 和 delete

1、当我们使用一条 new 表达式时，实际执行了三个步骤

1) new 表达式调用一个名为 operator new 或 operator new[]的标准库函数，该函数分配一块足够大的、原始的、未命名的内存空间以便存储特定类型的对象(或对象数组)
2) 编译器运行相应的构造函数以构造这些对象，并为其传入初始值
3) 对象被分配了空间并构造完成，返回一个该对象的指针

2、当我们使用一条 delete 表达式删除一个动态分配的对象时，实际执行了两个步骤

1) 第一步对指针所指向的对象或对象数组中的元素执行对应的析构函数
2) 编译器调用名为 operator delete 或 operator delete[]的标准库函数释放内存空间

3、如果程序希望控制内存分配的过程，则它们需要定义自己的 operator new 函数和 operator delete 函数。即使在标准库中已经存在这两个函数的定义，我们仍然可以定义自己的版本。编译器不会对这种重复的定义提出异议，相反，编译器将使用我们自定义的版本替换标准库定义的版本

4、当自定义了全局的 operator new 函数和 operator delete 函数后，我们就担负起了控制动态内存分配的职责，这两个函数必须是正确的：因为它们是程序整个处理过程中至关重要的一部分

5、应用程序可以在全局作用域中定义 operator new 和 operator delete 函数，也可以将它们定义成成员函数

1) 当编译器发现一条 new 表达式或 delete 表达式后将在程序中查找可供调用的 operator 函数
2) 如果被分配(释放)的对象是类类型，则编译器首先在类及其基类作用域中查找
3) 此时如果该类含有 operator new 成员或 operator delete 成员，则相应的表达式将调用这些成员
4) 否则，编译器在全局作用域查找匹配的函数
5) 此时如果编译器找到了用户自定义的版本，则使用该版本执行 new 表达式或 delete 表达式
6) 如果没有找到，则使用标准库定义的版本

6、我们可以使用作用域运算符令 new 表达式或 delete 表达式忽略定义在类中的函数，直接执行全局作用域中的版本，例如::new 只在全局作用域中查找匹配的 operator 函数，::delete 与之类似

#### 19.1.1.1. operator new 接口和 operator delete 接口

1、标准库定义了 operator new 函数和 operator delete 函数的 8 个重载版本，前四个可能抛出 bad_alloc 异常，后四个不会抛出异常

- 以下是 4 个可能抛出异常的版本

1) void *operator new(size_t);

2) void *operator new[](size_t);
3) void *operator delete(void*) noexcept;
4) void *operator delete[](void*) noexcept;

- 以下是 4 个承诺不会排除异常的版本

1) void *operator new(size_t, nothrow_t&) noexcept;
2) void *operator new[](size_t, nothrow_t&) noexcept;
3) void *operator delete(void*, nothrow_t&) noexcept;
4) void *operator delete[](void*, nothrow_t&) noexcept;

- nothrow_t 是定义在头文件中的一个 struct，这个对象不包含任何成员，new 头文件还定义了一个名为 nothrow 的 const 对象，用户可以通过这个对象请求 new 的非抛出版本(仅仅用于静态分派)
- 与析构函数类似，operator delete 也不允许抛出异常，当我们重载这些运算符时，必须使用 noexcept 异常说明符指定其不抛出异常

2、自定义的版本必须位于全局作用域或者类作用域中，作为类的成员时，是隐式静态的，无须显式声明 static，但是声明也无妨

3、对于 operator new 函数或者 operator new[]函数来说，它的返回类型必须是 void*，第一个形参类型必须是 size_t 且该形参不能含有默认实参

- 当编译器调用 operator new 时，把存储指定类型对象所需的字节数传给 size_t 形参
- 当编译器调用 operator new[]时，传入函数的则是存储数组中所有元素所需的空间

4、下面这个全局函数不能被重载(**即::placement new**)

**void *operator new(size_t, void*);**

5、对于 operator delete 函数或者 operator delete[]函数来说，它们的返回类型必须是 void，第一个形参类型必须是 void*

- 当 operator delete 或 operator delete[]作为类型静态成员时，该函数可以包含另外一个类型为 size_t 的形参，该形参的初始值是第一个形参所指对象的字节数。size_t 形参可用于删除继承体系中的对象，如果基类有一个虚析构函数，则传递给 operator delete 的字节数将因待删除指针所指对象的动态类型不同而有所区别。
- ~~而且实际运行的 operator delete 函数版本也由对象的动态类型决定???扯淡吧~~

6、delete 版本确定的测试

```
struct AAA{
    void* operator new(size_t size){
        cout<<"AAA's operator new"<<endl;
        return ::operator new(size);
    }
    void operator delete(void*p){
        cout<<"AAA's operator delete"<<endl;
        ::operator delete(p);
    }
};
```

```
struct BBB:public AAA{
    void* operator new(size_t size){
        cout<<"BBB's operator new"<<endl;
        return ::operator new(size);
    }
    void operator delete(void*p){
        cout<<"BBB's operator delete"<<endl;
        ::operator delete(p);
    }
};

int main(){
    void *p1=(void*) new AAA();
    delete p1;

    void *p2=(void*) new BBB();
    delete p2;

    AAA* p3=new BBB();
    delete p3;
}
```

- 输出如下

```
AAA's operator new
BBB's operator new
BBB's operator new
AAA's operator delete
```

- **结论：delete 通过指针的静态类型来决定调用版本**

7、我们重载的是 operator new 和 operator delete 等一系列运算符，而非 new 和 delete 关键字。new 和 delete 关键字是无法被重载的，其含义不可改变

#### 19.1.1.2. malloc 函数和 free 函数

1、我们可以使用名为 malloc 和 free 的函数，C++从 C 语言中继承了这些函数，并将其定义在 cstdlib 头文件中

- malloc 函数接受一个表示待分配字节数的 size_t，返回指向分配空间的指针或者返回 0 表示分配失败
- free 函数接受一个 void*，它是 malloc 返回的指针副本，free 将相关内存返回给系统
- 调用 free(0)没有任何意义

### 19.1.2. 定位 new 表达式

1、在 C++早期版本中，allocate 类还不是标准库的一部分，应用程序如果想把内存分配与初始化分离开来，需要调用 operator new 和 operator delete，这两个函数的行为与 allocator 的 allocate 成员和 deallocate 成员非常类似，负责分配或释放内存空间，但是不会构造或销毁对象

2、与 allocator 不同的是，对于 operator new 分配的内存空间，我们无法使用 construct 函数构造对象。相反，我们应该使用 new 的定位 new(placement new)形式构造对象。new 的这种形式为分配函数提供了额外的信息，我们可以使用定位 new 传递一个地址

new (place_address) type
new (place_address) type (initializers)
new (place_address) type [size]
new (place_address) type [size] {braced initializer list}

- 其中 place_address 必须是一个指针
- 当仅通过一个地址调用时，定位 new 使用 operator new(size_t,void*)分配它的内存。该函数不分配任何内存，它只是简单地返回指针实例，然后由 new 表达式负责在指定的地址初始化对象以完成整个工作
- 定位 new 允许我们在一个特定的、预先分配的内存地址上构造对象

3、定位 new 与 allocator 的 construct 成员非常类似，但是有一个重要区别

- 我们传给 construct 的指针必须指向同一个 allocator 对象分配的空间
- 传给 new 的指针无须指向 operator new 分配的内存，甚至不需要指向动态内存

#### 19.1.2.1. 显式的析构函数调用

1、就像定位 new 与使用 allocator 的成员 allocate 类似一样，对析构函数的显式调用也与使用 allocator 的成员 destroy 很类似

2、与 allocator 的成员 destroy 类似，调用析构函数可以清除给定对象，但是不会释放该对象的内存空间，如果需要的话，我们可以重新使用该空间

### 19.1.3. 测试

1、测试一

```
struct test{
    int i,j;
    double d;
    static int id;
    //在对象的内存前面多分配一个 int 大小的内存，存储额外的 id
    void* operator new(size_t size){
        void* p=::operator new(size+sizeof(int));
        char* raw=(char*)p;
        int* pi=(int*)p;
        *pi=++id;
        return (void*)(raw+sizeof(int));
    }

    void operator delete(void* p){
        char* raw=(char*)p-sizeof(int);
        ::operator delete(raw);
    }
```

```
    //在为数组中每个对象的的内存前面多分配一个 int 大小的内存，存储额外的 id
    //但是在使用 test* p 指针进行元素访问时，内存偏移量并不会将我们的额外 int 考虑进去，因此直接用 p[i]是访问不到正常对象的
    //正确的做法是，只在数组头元素前面存一些额外信息，这样在使用指针进行内存偏移时也不会出错
    void* operator new[](size_t size){
        size_t num=size/sizeof(test);
        size_t size_one=sizeof(test)+sizeof(int);
        cout<<"size: "<<size<<endl;
        cout<<"num: "<<num<<endl;
        cout<<"size_one: "<<size_one<<endl;
        void* p=::operator new(size_one*num);
        char* raw=(char*)p;
        for(size_t i=0;i<num;i++){
            //为每一个对象都存一个额外的 id
            int* pi=(int*)(raw+size_one*i);
            *pi=++id;
        }
        return (void*)(raw+sizeof(int));
    }

    void operator delete[](void* p){
        char* raw=(char*)p-sizeof(int);
        ::operator delete[](raw);
    }
};

int test::id=100;

int get_id(void* p){
    return *(int*)((char*)p-sizeof(int));
}

int main(){
    int N=15;
    cout<<"N: "<<N<<", sizeof(test): "<<sizeof(test)<<endl;
    test* ary=new test[N];
    for(int i=0;i<N;i++){
        if(i>0){
            cout<<"difference between ary[i] and ary[i-1]: "
            <<(char*)&ary[i]-(char*)&ary[i-1]<<endl;
        }
        cout<<"id :"<<get_id((char*)&ary[i]+sizeof(int)*i)<<endl;
```

```
        }
    }
```

- 输出

```
N: 15, sizeof(test): 16
size: 240
num: 15
size_one: 20
id :101
difference between ary[i] and ary[i-1]: 16
id :102
difference between ary[i] and ary[i-1]: 16
id :103
difference between ary[i] and ary[i-1]: 16
id :104
difference between ary[i] and ary[i-1]: 16
id :105
difference between ary[i] and ary[i-1]: 16
id :106
difference between ary[i] and ary[i-1]: 16
id :107
difference between ary[i] and ary[i-1]: 16
id :108
difference between ary[i] and ary[i-1]: 16
id :109
difference between ary[i] and ary[i-1]: 16
id :110
difference between ary[i] and ary[i-1]: 16
id :111
difference between ary[i] and ary[i-1]: 16
id :112
difference between ary[i] and ary[i-1]: 16
id :113
difference between ary[i] and ary[i-1]: 16
id :114
difference between ary[i] and ary[i-1]: 16
id :115
```

2、测试 2

```
struct A{
        int a1;
        int a2;
};

struct B{
```

```
        virtual void funb(){cout<<"B.funb()"<<endl;}
        int b1;
        int b2;
};

struct C{
        virtual void func(){cout<<"C.func()"<<endl;}
        int c1;
        int c2;
};

struct D:public A,public B,public C{
        int d1;
        int d2;
        virtual void fund(){cout<<"D.fund()"<<endl;}
};

int main()
{
        A a;
        cout<<"sizeof(A): "<<sizeof(A)<<endl;
        cout<<(char*)&a.a1-(char*)&a<<endl;
        cout<<(char*)&a.a2-(char*)&a<<endl<<endl;

        B b;
        cout<<"sizeof(B): "<<sizeof(B)<<endl;
        b.funb();
        cout<<(char*)&b.b1-(char*)&b<<endl;
        cout<<(char*)&b.b2-(char*)&b<<endl<<endl;

        C c;
        cout<<"sizeof(C): "<<sizeof(C)<<endl;
        c.func();
        cout<<(char*)&c.c1-(char*)&c<<endl;
        cout<<(char*)&c.c2-(char*)&c<<endl<<endl;

        D d;
        cout<<"sizeof(D): "<<sizeof(D)<<endl;
        d.funb();
        d.func();
        d.fund();
        cout<<(char*)&d.a1-(char*)&d<<endl;
        cout<<(char*)&d.a2-(char*)&d<<endl;
        cout<<(char*)&d.b1-(char*)&d<<endl;
```

```
    cout<<(char*)&d.b2-(char*)&d<<endl;
    cout<<(char*)&d.c1-(char*)&d<<endl;
    cout<<(char*)&d.c2-(char*)&d<<endl;
     cout<<(char*)&d.d1-(char*)&d<<endl;
    cout<<(char*)&d.d2-(char*)&d<<endl<<endl;

    cout<<"&d-addr: "<<&d<<endl;
    A* pa=&d;
    cout<<"pa-addr: "<<pa<<", diff: "<<(char*)pa-(char*)&d<<endl;
    cout<<(char*)&pa->a1-(char*)pa<<endl;
    cout<<(char*)&pa->a2-(char*)pa<<endl<<endl;

    B* pb=&d;
    cout<<"pb-addr: "<<pb<<", diff: "<<(char*)pb-(char*)&d<<endl;
    cout<<(char*)&pb->b1-(char*)pb<<endl;
    cout<<(char*)&pb->b2-(char*)pb<<endl<<endl;

    C* pc=&d;
    cout<<"pc-addr: "<<pc<<", diff: "<<(char*)pc-(char*)&d<<endl;
    cout<<(char*)&pc->c1-(char*)pc<<endl;
    cout<<(char*)&pc->c2-(char*)pc<<endl<<endl;

    return 0;
}
```

- 输出

```
sizeof(A): 8
0
4

sizeof(B): 16
B.funb()
8
12

sizeof(C): 16
C.func()
8
12

sizeof(D): 48
B.funb()
C.func()
D.fund()
16
```

```
20
8
12
32
36
40
44

&d-addr: 0x7fff5fbff500
pa-addr: 0x7fff5fbff510, diff: 16
0
4

pb-addr: 0x7fff5fbff500, diff: 0
8
12

pc-addr: 0x7fff5fbff518, diff: 24
8
12
```

3、测试 3(情况 2 的变体)

```
struct A{
    virtual void funa(){cout<<"A.funa()"<<endl;}
    int a1;
    int a2;
};

struct B{
    virtual void funb(){cout<<"B.funb()"<<endl;}
    int b1;
    int b2;
};

struct C{
    virtual void func(){cout<<"C.func()"<<endl;}
    int c1;
    int c2;
};

struct D:public A,public B,public C{
    int d1;
    int d2;
    virtual void fund(){cout<<"D.fund()"<<endl;}
```

```
};

int main()
{
	A a;
	cout<<"sizeof(A): "<<sizeof(A)<<endl;
	cout<<(char*)&a.a1-(char*)&a<<endl;
	cout<<(char*)&a.a2-(char*)&a<<endl<<endl;

	B b;
	cout<<"sizeof(B): "<<sizeof(B)<<endl;
	b.funb();
	cout<<(char*)&b.b1-(char*)&b<<endl;
	cout<<(char*)&b.b2-(char*)&b<<endl<<endl;

	C c;
	cout<<"sizeof(C): "<<sizeof(C)<<endl;
	c.func();
	cout<<(char*)&c.c1-(char*)&c<<endl;
	cout<<(char*)&c.c2-(char*)&c<<endl<<endl;

	D d;
	cout<<"sizeof(D): "<<sizeof(D)<<endl;
	d.funa();
	d.funb();
	d.func();
	d.fund();
	cout<<(char*)&d.a1-(char*)&d<<endl;
	cout<<(char*)&d.a2-(char*)&d<<endl;
	cout<<(char*)&d.b1-(char*)&d<<endl;
	cout<<(char*)&d.b2-(char*)&d<<endl;
	cout<<(char*)&d.c1-(char*)&d<<endl;
	cout<<(char*)&d.c2-(char*)&d<<endl;
	cout<<(char*)&d.d1-(char*)&d<<endl;
	cout<<(char*)&d.d2-(char*)&d<<endl<<endl;

	cout<<"&d-addr: "<<&d<<endl;
	A* pa=&d;
	cout<<"pa-addr: "<<pa<<", diff: "<<(char*)pa-(char*)&d<<endl;
	cout<<(char*)&pa->a1-(char*)pa<<endl;
	cout<<(char*)&pa->a2-(char*)pa<<endl<<endl;

	B* pb=&d;
	cout<<"pb-addr: "<<pb<<", diff: "<<(char*)pb-(char*)&d<<endl;
```

```
    cout<<(char*)&pb->b1-(char*)pb<<endl;
    cout<<(char*)&pb->b2-(char*)pb<<endl<<endl;

    C* pc=&d;
    cout<<"pc-addr: "<<pc<<", diff: "<<(char*)pc-(char*)&d<<endl;
    cout<<(char*)&pc->c1-(char*)pc<<endl;
    cout<<(char*)&pc->c2-(char*)pc<<endl<<endl;

    return 0;
}
```

- 输出

```
sizeof(A): 16
8
12

sizeof(B): 16
B.funb()
8
12

sizeof(C): 16
C.func()
8
12

sizeof(D): 56
A.funa()
B.funb()
C.func()
D.fund()
8
12
24
28
40
44
48
52

&d-addr: 0x7fff5fbff4f0
pa-addr: 0x7fff5fbff4f0, diff: 0
8
12
```

```
pb-addr: 0x7fff5fbff500, diff: 16
8
12

pc-addr: 0x7fff5fbff510, diff: 32
8
12
```

1、基类 object 为不含有虚函数的类
- 当 object 子类也不含有虚函数，那么在对象地址前面放置的额外内存可以正常访问到
- 当 object 子类具有虚函数，由于虚函数表的存在，导致发生内存偏移

2、当基类 object 为含有虚函数的类
- route 不继承自 object，但是含有虚函数
- 子类 route_tcp 继承自 object 和 route，将 route_tcp*转为 route*然后调用 route*的成员会发生错误

## 19.2. 运行时类型识别

1、运行时类型识别(run-time type identification,RTTI)的功能由两个运算符实现
1) typeid 运算符，用于返回表达式的类型
2) 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

2、当我们将这两个运算符用于某种类型的指针或引用，并且该类型含有虚函数时，运算符将使用指针或引用所绑定的动态类型

3、这两个运算符特别适用于以下情况
- 当我们想使用基类对象的指针或引用执行某个派生类操作并且该操作不是虚函数(一般来说，只要有可能我们应该尽量使用虚函数)

4、当我们无法使用虚函数时，则可以使用一个 RTTI 运算符。另一方面，与虚成员函数相比，使用 RTTI 运算符蕴含着更多潜在的风险：程序员必须清楚地知道转换的目标类型并且必须检查类型转换是否被成功执行

### 19.2.1. dynamic_cast 运算符

1、dynamic_cast 运算符(dynamic_cast operator)的使用形式如下
1) dynamic_cast<type*>(e) ==>返回 type*类型
2) dynamic_cast<type&>(e) ==>返回 type&类型
3) dynamic_cast<type&&>(e) ==>返回 type&&类型
- 其中 type 必须是一个类类型，并且通常情况(好像是必须)下该类型应该含有虚函数

2、在上面的所有形式中，e 的类型必须符合以下三个条件中任意一个
1) e 的类型是目标 type 的公有派生类
2) e 的类型是目标 type 的共有基类
3) e 的类型就是目标 type 的类型
- 如果符合，则类型转换成功，否则，转换失败

- 如果一条 dynamic_cast 语句的转换目标是指针类型并且失败了，结果为 0
- 如果一条 dynamic_cast 语句的转换目标是引用类型，将抛出一个 bad_cast 异常，该异常定义在 typeinfo 标准库头文件中

### 19.2.2. typeid 运算符

1、typeid 表达式的形式是 typeid(e)，其中 e 可以是任意表达式或类型的名字
2、typeid 操作的结果是一个常量对象的引用，该对象的类型是标准库类型 type_info 或者 type_info 的共有派生类。type_info 类定义在 typeinfo 头文件中
3、typeid 运算符可以作用于任意类型的表达式

- 和往常一样，顶层 const 被忽略
- 如果表达式是一个引用，则 typeid 返回该引用所引对象的类型
- 当 typeid 作用域数组或函数时，并不会执行向指针的标准类型转换，即对数组 a 执行 typeid(a)，则所得的结果是数组类型而非指针类型

4、当运算对象不属于类类型或者是一个不包含任何虚函数的类时，typeid 运算符指示的是运算对象的静态类型；当运算对象至少定义了一个虚函数的类的左值时，typeid 的结果直到运行时才会求
5、当 typeid 作用于指针时(而非指针所指对象)，返回的结果是该指针的静态编译时类型
6、typeid 是否需要运行时检查决定了表达式是否会被求值

- 只有当类型含有虚函数时，编译器才会对表达式求值
- 如果当类型不含有虚函数时，typeid 返回表达式的静态类型，即编译器无须对表达式求值也能知道表达式的静态类型

7、如果表达式的动态类型可能与静态类型不同，则必须在运行时对表达式求值以确定返回的类型，这条规则适用于 typeid(*p)的情况

- 如果指针 p 所指的类型不含有虚函数，则 p 不必非得是一个有效的指针
- 如果指针 p 所指的类型含有虚函数，则 p 必须是一个有效指针，否则 typeid(*p)将会抛出一个名为 bad_typeid 的异常

### 19.2.3. type_info 类

1、type_info 类的精确定义随着编译器的不同而略有差异
2、C++标准规定 type_info 类必须定义在 typeinfo 头文件中，并至少提供如下操作

- t1==t2，如果 type_info 对象 t1 和 t2 表示同一种类型，返回 true，否则返回 false
- t1!=t2，如果 type_info 对象 t1 和 t2 表示不同类型，返回 true，否则返回 false
- t.name()，返回一个 C 风格字符串，表示类型名字的可打印形式，类型名字的生成方式因系统而异
- t1.before(t2)，返回一个 bool 值，表示 t1 是否位于 t2 之前。before 所采用的顺序是依赖于编译器的

## 19.3. 枚举类型

19.4. 类成员指针

19.5. 嵌套类

19.6. union：一种节省空间的类

19.7. 局部类

19.8. 固有的不可移植性

# Chapter 20. 宏

## 20.1. #define

### 20.1.1. #define 的概念

1、仅在当前文件可见，一般将其定义在头文件中，若需要使用该宏，则需要包含该头文件

2、#define 命令是 C 语言中的一个宏定义命令，它用来将一个标识符定义为一个字符串，该标识符被称为宏名，被定义的字符串称为替换文本

3、该命令有两种格式：一种是简单的宏定义，另一种是带参数的宏定义

1) 简单的宏定义

#define <宏名> <字符串>

2) 带参数的宏定义

#define <宏名>(<参数表>) <宏体>

4、一个标识符被宏定义后，该标识符便是一个宏名。这时，在程序中出现的是宏名，在该程序被编译前，先将宏名用被定义的字符串替换，这称为宏替换，替换后才进行编译，宏替换是简单的替换

### 20.1.2. 宏替换发生的时机

1、为了能够真正理解#define 的作用，让我们来了解一下对 C 语言源程序的处理过程。当我们在一个集成的开发环境如 Turbo C 中将编写好的源程序进行编译时，实际经过了预处理、编译、汇编和连接几个过程。其中预处理器产生编译器的输出，它实现以下的功能：

1) 文件包含：可以把源程序中的#include 扩展为文件正文，即把包含的.h 文件找到并展开到#include 所在处
2) 条件编译：预处理器根据#if 和#ifdef 等编译命令及其后的条件，将源程序中的某部分包含进来或排除在外，通常把排除在外的语句转换成空行
3) 宏展开：预处理器将源程序文件中出现的对宏的引用展开成相应的宏 定义，即本文所说的#define 的功能，由预处理器来完成

2、经过预处理器处理的源程序与之前的源程序有所有不同，在这个阶段所进行的工作只是纯粹的替换与展开，没有任何计算功能，所以在学习#define 命令时只要能真正理解这一点，这样才不会对此命令引起误解并误用

### 20.1.3. #define 中的三个特殊符号：#，##，#@

1、x##y：表示 x 连接 y

2、#@x：表示给 x 加上单引号，结果返回一个 const char

3、#x：给 x 加上双引号

# Chapter 21. 编译过程

## 21.1. 概念：

1、**编译**：编译器对源代码进行编译，是将以文本形式存在的源代码翻译为机器语言形式的目标文件的过程
2、**编译单元**：对于 C++来说，**每一个 cpp 文件就是一个编译单元**。个编译单元互相不可知(独立性)(Java 中的编译单元是.java 文件)
3、**目标文件(.o 文件)**：由编译所生成的文件，以机器码的形式包含了编译单元里所有的代码和数据，以及一些其他的信息。

## 21.2. 表

1、编译器把一个 cpp 编译为目标文件的时候，除了要在目标文件里写入 cpp 里包含的数据和代码，至少还要提供三个表：

- **未解决符号表**：提供了所有在该编译单元里引用但是定义并不在本编译单元里的符号及其出现的地址
- **导出符号表**：提供了本编译单元具有定义，并且愿意提供给其他编译单元使用的符号及其地址
- **地址重定向表**：提供了本编译单元所有对自身地址的引用的记录，因为每个编译单元的地址都是从 0 开始 的，所以最终拼接起来的时候地址会重复。所以链接器会在拼接的时候对各个单元的地址进行调整

## 21.3. 链接过程

1、链接器进行链接的时候，首先决定各个目标文件在最终可执行文件里的位置
2、然后访问所有目标文件的地址重定向表，对其中记录的地址进行重定向(即加上该编译单元实际在可执行文件里的起始地址)。
3、然后遍历所有目标文件的未解决符号表，并且在所有导出符号表里查找匹配的符号，并在未解决符号表中记录的位置上填写实际的地址(也要加上拥有该符号定义的编译单元实际在可执行文件里的起始地址)。
4、最后把目标文件的内容写在各自的位置上，再做一些别的工作，一个可执行文件就出炉了。

## 21.4. 经典错误

1、unresolved external link...：连接器发现一个未解决符号，但是出现在导出符号表里没有找到对应的项
2、duplicated external simbols...：导出符号表里出现了重复项，因此连接器无法确定应该使用哪一个。

## 21.5. C++提供的特性：

1、extern：告诉编译器，这个符号在别的编译单元里定义，也就是要把这个符号

放到未解决符号表里去(外部链接)
2、static：如果该关键字位于全局函数或者变量声明的前面，**表示该编译单元不导出这个函数/变量的符号**，因此无法再别的编译单元里使用。(内部链接)。如果 static 是局部变量，则该变量的存储方式和全局变量一样，但仍然不导出符号。

## 21.6. 默认连接属性

1、函数和变量：默认外部连接
2、**const 变量：默认内部连接**
3、可以通过 extern 和 static 改变连接属性

## 21.7. 内外部链接各自的利弊：

1、**外部链接**：外部链接的符号，可以在整个程序范围内使用(因为导出了符号)。但是同时要求其他编译单元不能导出相同的符号(不然就是 duplicated external simbols)
2、**内部链接**：内部链接的符号，不能在别的编译单元里使用。但是不同的编译单元可以拥有相同名称的内部链接符号。

## 21.8. 各类问题

### 21.8.1. 为什么常量默认内部链接

1、这就是为了能够在头文件里如 const int n = 0 这样的定义常量。由于常量是只读的，因此即使每个编译单元都拥有一份定义也没有关系。
2、如果一个定义于头文件里的变量拥有内部链接，那么如果出现多个编译单元都定义该变量，则其中一个编译单元对该变量进行修改，不会影响其他单元的同一变量

### 21.8.2. 为什么函数默认是外部链接

1、虽然函数是只读的，但是和变量不同，函数在代码编写的时候非常容易变化，如果函数默认具有内部链接，则人们会倾向于把函数定义在头文件里，那么一旦函数被修改，所有包含了该头文件的编译单元都要被重新编译。另外，函数里定义的静态局部变量也将被定义在头文件里。

### 21.8.3. 为什么类的静态变量不可以就地初始化

1、不允许这样做得原因是，由于 class 的声明通常是在头文件里，如果允许这样做，其实就相当于在头文件里定义了一个非 const 变量。

### 21.8.4. 头文件定义一个 const 对象会怎么样

1、一般不会怎么样，这个和 C 里的在头文件里定义 const int 一样，每一个包含了这个头文件的编译单元都会定义这个对象。但由于该对象是 const 的，所以没什么影响。但是：有 2 种情况可能破坏这个局面：

- 如果涉及到对这个 const 对象取地址并且依赖于这个地址的唯一性，那么在不同的编译单元里，取到的地址可以不同。(但一般很少这么做)

- 如果这个对象具有 mutable 的变量，某个编译单元对其进行修改，则同样不会影响到别的编译单元。

### 21.8.5. 为什么类的静态常量也不可以就地初始化

1、因为这相当于在头文件里定义了 const 对象。**作为例外，int/char 等可以进行就地初始化，是因为这些变量可以直接被优化为立即数，就和宏一样**。

### 21.8.6. 内联函数：

1、**C++里的内联函数由于类似于一个宏**，因此不存在链接属性问题。

### 21.8.7. 为什么公共使用的内联函数要定义于头文件里

1、因为编译时编译单元之间互相不知道，如果内联函数被定义于.cpp 文件中，**编译其他使用该函数的编译单元的时候没有办法找到函数的定义，因此无法对函数进行展开**。所以说如果内联函数定义于.cpp 文件里，那么就只有这个 cpp 文件可以使用这个函数。

### 21.8.8. 头文件里内联函数被拒绝会怎样

1、如果定义于头文件里的内联函数被拒绝，那么编译器会自动在每个包含了该头文件的编译单元里定义这个函数并且不导出符号。

# Chapter 22. 初始化

## 22.1. 默认初始化

1、内置类型：如果变量未被显式初始化，它的值由定义的位置决定
- 定义域任何函数之外的变量都被初始化为 0
- 定义在函数内部的内置类型将不被初始化，其值是未定义的
- 那些依赖编译器合成默认构造函数的内置类型，如果在类内未被初始化，值也是未定义的

2、类类型：
- 调用默认构造函数(编译器合成或者自定义)
- 如果存在类内初始值，用它来初始化成员(只能用花括号，或者拷贝初始化，不能用直接初始化，即不能用圆括号)
- 否则，默认初始化成员

3、其他：
- 默认情况下，动态分配的对象是默认初始化的

```
int *p=new int;
```

## 22.2. 值初始化

1、对于定义了自己的构造函数的类类型，要求值初始化是没有意义的，不管采用什么形式，对象都会用默认构造函数来初始化

2、对于内置类型，值初始化的内置类型对象有着良好定义的值，而默认初始化的对象的值是未定义的

3、值初始化常见例子
- 容器模板类型里(array 不是)，内置类型的默认初始化都是值初始化，是一个有意义的值

```
C<int> c(10);     <==10 个为 0 的 int
```

- pair、tuple 的默认初始化也是值初始化
- 列表初始化数组的时候，当列表中的元素小于数组元素个数的时候，数组中剩下的元素进行值初始化

```
string *ps1=new string(); //与默认初始化无区别
int *pi1=new int(); //与默认初始化区别很大
auto p=make_shared<int>();
```

## 22.3. 直接初始化：

1、当我们使用直接初始化时，我们实际上是要求编译器使用普通的函数匹配来选择我们提供的参数最匹配的构造函数

2、变量名后跟上圆括号，类类型对象，调用构造函数

3、explicit 的构造函数只能用直接初始化，不能用于含有隐式转换的拷贝初始化 string s(10); 等等

## 22.4. 拷贝初始化：

1、当我们使用拷贝初始化时，我们要求编译器将右侧运算对象拷贝到正在创建的对象中，如果需要的话还需要类型转换

2、**是否调用拷贝构造函数与拷贝初始化并不等价**

- **调用拷贝构造函数不一定是拷贝初始化**

```
string s;
string s1(s);   //调用拷贝构造函数，但是这是直接初始化
string s2=s;   //拷贝初始化
```

- **此外，拷贝初始化也未必调用拷贝构造函数，比如一个非 explicit 的构造函数，允许将参数类型向该类型进行转换**

3、用(=号)初始化变量(**用该类型的一个对象的拷贝初始化另一个对象**)

```
Int i=5;
vector<int> vec1=vec2；
```

4、explict 的构造函数不能拷贝初始化(接受单一实参的构造函数是 explict 的是指：不能在需要该类型的地方用接受一个实参的构造函数的形参类型替代!!! )

## 22.5. 列表初始化：

1、变量名后跟上花括号

```
vector<int> vec{1,2,3,4,5};
```

## 22.6. 类内特殊成员的初始化

1、对于 const 和引用类型

- 这两种类型被要求必须初始化
- **而类初始化的唯一地方就是初始化列表**
- 构造函数函数体内的代码并非初始化，而是赋值

2、对于 istream、ostream、ifstream、ofstream 等流对象

- 其特点是不允许拷贝，即拷贝构造函数是 delete 的
- 这类成员函数可以在构造函数函数体内进行**赋值**
  - 只允许调用**移动构造函数**以及**移动赋值运算符**
- **可以在初始化列表中进行初始化，但是也不能调用拷贝构造函数**

# Chapter 23. static

## 23.1. 类别

### 23.1.1. 函数内部声明的 static 变量，可作为对象间的一种通信机制

1、如果一局部变量被声明为 static，那么将只有唯一的一个静态分配的对象，它被用于在该函数的所有调用中表示这个变量。这个对象将只在执行线程第一次到达它的定义使初始化。

### 23.1.2. 局部静态对象

1、对于局部静态对象，构造函数是在控制线程第一次通过该对象的定义时调用。在程序结束时，局部静态对象的析构函数将按照他们被构造的相反顺序逐一调用，没有规定确切时间。

### 23.1.3. 静态成员和静态成员函数

1、如果一个变量是类的一部分，但却不是该类的各个对象的一部分，它就被成为是一个 static 静态成员。一个 static 成员只有唯一的一份副本，而不像常规的非 static 成员那样在每个对象里各有一份副本。同理，一个需要访问类成员，而不需要针对特定对象去调用的函数，也被称为一个 static 成员函数。
类的静态成员函数只能访问类的静态成员(变量或函数)。
2、**类的静态成员函数只能访问类的静态成员(变量或函数)**

## 23.2. 作用

### 23.2.1. 隐藏

1、当我们同时编译多个文件时，所有未加 static 前缀的全局变量和函数都具有全局可见性。
2、如果加了 static，就会对其它源文件隐藏。利用这一特性可以在不同的文件中定义同名函数和同名变量，而不必担心命名冲突
3、对于函数来讲，static 的作用仅限于隐藏。static 函数与普通函数的区别：static 函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝.
4、而对于变量，static 还有下面两个作用。

### 23.2.2. 保持变量内容的持久

1、存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。
2、共有两种变量存储在静态存储区：全局变量和 static 变量，只不过和全局变量比起来，static 可以控制变量的可见范围，说到底 static 还是用来隐藏的。

### 23.2.3. 默认初始化为 0

1、其实全局变量也具备这一属性，因为全局变量也存储在静态数据区
2、在静态数据区，内存中所有的字节默认值都是 0x00，某些时候这一特点可以减少程序员的工作量。比如初始化一个稀疏矩阵，我们可以一个一个地把所有元素都置 0，然后把不是 0 的几个元素赋值。如果定义成静态的，就省去了一开始

置 0 的操作。再比如要把一个字符数组当字符串来用，但又觉得每次在字符数组末尾加'\0'太麻烦。如果把字符串定义成静态的，就省去了这个麻烦，因为那里本来就是'\0'。

## 23.3. 选择建议

1、若全局变量仅在单个 C 文件中访问，则可以将这个变量修改为静态全局变量，以降低模块间的耦合度
2、若全局变量仅由单个函数访问，则可以将这个变量改为该函数的静态局部变量，以降低模块间的耦合度
3、设计和使用访问动态全局变量、静态全局变量、静态局部变量的函数时，需要考虑重入问题
4、如果我们需要一个可重入的函数，那么，我们一定要避免函数中使用 static 变量(这样的函数被称为：带"内部存储器"功能的的函数)
5、函数中必须要使用 static 变量情况:比如当某函数的返回值为指针类型时，则必须是 static 的局部变量的地址作为返回值，若为 auto 类型，则返回为错指针(**一个类的静态方法返回这个类的唯一实例，即单例模式，用局部静态变量非常好，不用额外增加判断是否唯一实例的条件**)

## 23.4. 含义的发展

1、术语 static 有着不寻常的历史
- 起初，在 C 中引入关键字 static 是为了表示退出一个块后仍然存在的局部变量
- 随后，static 在 C 中有了第二种含义：用来表示不能被其它文件访问的全局变量和函数。为了避免引入新的关键字，所以仍使用 static 关键字来表示这第二种含义。
- 最后，C++重用了这个关键字，并赋予它与前面不同的第三种含义：表示属于一个类而不是属于此类的任何特定对象的变量和函数(与 Java 中此关键字的含义相同)

## 23.5. 全局变量、静态全局变量、静态局部变量和局部变量

1、按存储区域分：
- 全局变量、静态全局变量和静态局部变量都存放在内存的静态存储区域
- 局部变量存放在内存的栈区

2、按作作用域分：
- 全局变量在整个工程文件内都有效
- 静态全局变量只在定义它的文件内有效
- 静态局部变量只在定义它的函数内有效，只是程序仅分配一次内存，函数返回后，该变量不会消失
- 局部变量在定义它的函数内有效，但是函数返回后失效

3、全局变量(外部变量)的说明之前再冠以 static 就构成了静态的全局变量。全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。这两者在存

储方式上并无不同。这两者的区别虽在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。 而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其它源文件中引起错误。从以上分析可以看出：

- 把局部变量改变为静态变量后是改变了它的存储方式即改变了它的生存期。
- 把全局变量改变为静态变量后是改变了它的作用域，限制了它的使用范围。

# Chapter 24. 右值引用

## 24.1. Introduction

1、右值引用为了解决两类问题
   1) 实现移动语义(move)
   2) 完美转发
2、左值右值的最初始定义来源于 C
   1) 出现在"="左边的就是左值
   2) 出现在"="右边的就是右值
3、后来由于出现了自定义对象(Object)左右值的概念有了细微的变化，下面给出左右值的定义(尽管可能存在争议，但是便于理解)
   1) 左值表达式有其内存空间，允许我们使用取值运算符&
   2) 右值表达式是无法获取地址的

## 24.2. Move Semantics

1、假设类型 C 持有一个指针或者一些资源，记为 m_pResource
2、拷贝赋值运算符如下

```
X& X::operator=(X const & rhs){// const X&与 X const&是一样的
    //[...]
    //克隆一份 rhs.m_pResource 指向的资源，记为 clone
    //销毁 this->m_pResource 指向的资源
    //将 m_pResource 指向 clone
    //[...]
}
```

3、同样对于拷贝构造函数也是如此

```
X foo();
X x;
x=foo();
```

- 最后一句包括以下三个阶段
  1) 克隆一份 foo()返回的临时对象所指向的资源，记为 clone
  2) 销毁 x 所指向的资源，并用 clone 代替
  3) 销毁 foo()返回的临时对象
- 这样做虽然没问题，但是效率极差
- 我们可以交换 x 与 foo()返回对象的 m_pResource 指针，然后让 foo()返回的临时对象销毁 x 的初始资源即可。换句话说，在这种情况下"="右边就是一个右值，我们希望拷贝操作像下面这样运转，这便是移动语义

```
//[...]
//交换 this->pResource 和 rhs.m_pResource
//[...]
```

4、我们希望拷贝构造函数可以以如下的形式定义

```
X& X::operator=(<mystery type> rhs){
    //[...]
```

```
    //交换 this->m_pResource 和 rhs.m_pResource
    //[...]
}
```

- 这个 mystery type 本质上一定是个引用，因为我们想要移动它的资源
- 当我们有两个重载(overloads)时，一个接受普通的引用(左值引用)，另一个接受我们的 mystery type，于是右值会选择 mystery type 的版本，左值会选择普通的版本

## 24.3. Rvalue References

1、如果 X 是一种类型，那么 X&&叫做 X 的右值引用，相对地，X&叫做 X 的左值引用

2、右值引用类型，其行为和左值引用非常相似，但有几处不同

1) 最重要的就是：重载

```
void foo(X& x);
void foo(X&& x);

X x;
x foobar();

foo(x);//参数是左值，调用 foo(X&);
foo(foobar());//参数是右值，调用 foo(X&&);
```

3、左右值引用的主旨：允许一个函数调用在编译时分派(通过重载)，"我(被调用的函数)通过一个左值还是右值被调用"

4、这种右值引用的参数可以用在任何函数中，但是绝大多数用在对拷贝构造函数以及拷贝赋值运算符实现重载，即

```
X& X::operator=(X const & rhs);
X& X::operator=(X&& rhs);

X(const X& rhs);
X(X&& rhs);
```

## 24.4. Forcing Move Semantics

1、非移动语义的 swap

```
template<class T>
void swap(T&a,T&b){
    T tmp(a);
    a=b;
    b=tmp;
}
X a,b;
swap(a,b);
```

2、C++标准库 std::move 函数可以将一个参数转变成右值

```
template<class T>
void swap(T&a, T&b){
    T tmp(std::move(a));
    a=std::move(b);
    b=std::move(a);
}
X a,b;
swap(a,b)
```

- 如果没有重载接受右值引用的版本，那么将会调用接受左值引用的版本

3、许多标准算法能从移动语义获得增益

1) 例如原值排序，将会有大量元素交换的操作

4、在早期，STL 标准库要求元素可以拷贝。后来发现，在很多情况下，可移动就行了，不需要可拷贝，于是后来这些可移动不可拷贝的元素可以作为 STL 标准库容器的元素

5、考虑以下语句

```
a=b;
a=std::move(b);
```

- 如果移动语义通过 swap 来实现(下面的讨论基于的前提)，那么在上述赋值语句后，a 和 b 所持有的资源仅仅交换了而已，a 的资源并没有被立即销毁，而是在 b 结束生命周期时才被销毁(除非 b 又被 move 了，即又被传递到其他地方了)，因此我们并不知道 a 在赋值前所持有的资源在何时被销毁
- 一个变量被赋值了，但是它在赋值前所持有的资源可能还存在于某个地方
  - 如果析构函数是 trivial 的，那么是没有关系的
  - 如果析构函数不是 trivial 的，那么必须在赋值前销毁原先持有的资源，例如以下

```
X& X::operator=(X&& rhs){
    //在赋值前将 X 类型中那些重要的资源进行清理，让 this 指向的实例处于一种可赋值的状态
    //交换 this 与 rhs 的资源
    return *this;
}
```

## 24.5. Is an Rvalue Reference an Rvalue?

1、考虑这样一种情形

```
void foo(X&& x){
    X anotherX=x;
    //...
}
```

- 在 foo 内部调用 X 的拷贝构造函数
- 此时，x 是一个变量，被声明为右值引用类型，通常(不一定非得)指向了一个右值
- 因此希望 x 本身也是一个右值好像是合理的，即 X(X&& rhs)应该被调用，

换句话说，我们希望一个被声明为右值引用的变量其本身也是右值

2、**右值引用的设计者采用了一个十分微妙的解决方法**

- **被声明为右值引用的东西可以作为左值也可以作为右值**
- **取决于这个东西是否有名字，有名字就是左值，没名字就是右值**
- 这种设计背后的理由是：允许将默认语义应用于具有名称的东西
- **这也是 std::move 的原理!!!，std::move 返回的是一个无名的右值引用，因此会作为右值**
- 以下举几个简单的例子

```
void foo(X&& x){
    X anotherX=x;//调用 X(X const & rhs)
}

X&& goo();
X x=goo();//调用 X(X&& rhs)
```

3、std::move(x)

- 其声明返回一个右值引用，并且没有名字，因此就是右值
- std::move 通过将类型转变为右值引用，并且隐藏其名字来实现返回右值的承诺

4、继承体系中的细节

```
Base(Base const & rhs){
    //...
}
Base(Base&& rhs){
    //...
}

Derived(Derived const &rhs):Base(rhs){
    //...
}

Derived(Derived&& rhs):Base(rhs){//Wrong
    //...
}

Derived(Derived&& rhs):Base(std::move(rhs)){
    //...
}
```

## 24.6. Move Semantics and Compiler Optimizations

1、考虑下面的情况

```
X foo(){
    X x;
```

```
    //...
    return x;
}
```

- 你可能会觉得 foo()返回参数与 x 之间存在一次拷贝，于是你会采用以下方式来进行优化

```
X foo(){
    X x;
    //...
    return std::move(x);//making it worse!
}
```

- 然而，任何一款现代编译器会进行返回值优化(return value optimization)：**即在返回值的内存地址上去构造 x 对象，因此效率要比移动更高**

## 24.7. Perfect Forwarding:The Problem

1、右值引用的另一目的就是解决完美转发的问题，考虑下面的情况

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg arg){
    return shared_ptr<T>(new T(arg));
}
```

- 该函数的目的就是将参数 arg 从工厂方法传递到类型 T 的构造函数中
- 如果 T 的构造函数依赖于传入参数的引用，那么将会有问题
- 完美转发(perfect worwarding)：一切看起来就像没有 factory 这个中间层一样，实参传递给了 T 的构造函数

2、改进版本

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg& arg){
    return shared_ptr<T>(new T(arg));
}
```

但是以下调用会失败

```
factory<X>(hoo());//error
factory<X>(41);//error
```

3、通过增加一个常量引用版本，可以解决上述问题

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg const & arg){
    return shared_ptr<T>(new T(arg));
}
```

- 同时会造成两个问题

1) 如果工厂方法接受几个参数，则必须为各种参数的非 const 和 const 引用的组合提供重载，因此此方法对于具有多个参数的功能非常差
2) 这种传递方案不可能完美，因为它屏蔽了移动语义，在 factory 内的参数 arg 是一个左值，因此移动语义永远不会调用

4、右值引用在不增加重载的前提下就可以实现完美转发

## 24.8. Perfect Forwarding:The Solution

1、C++11 引入了引用折叠

1) A& &->A&
2) A& &&->A&
3) A&& &->A&
4) A&& &&->A&&

2、引入了一种特殊的模板参数推导机制

```
template<typename T>
void foo(T&& t);
```

1) 若果实参为 A 类型的左值，那么 T 将会推导为：A&，根据引用折叠规则，形参 t 的类型就是 A&
2) 如果实参为 A 类型的右值，那么 T 将会推导为 A，于是形参 t 的类型为 A&&

3、有了这两个武器，我们就可以解决上一节遇到的问题

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg&& arg){
    return shared_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

**其中<std::forward>定义如下**

```
template<class S>
S&& forward(typename remove_reference<S>::type& a) noexcept{
    return static_cast<S&&>(a);
}
```

- 考虑以下调用

```
X x;
factory<A>(x);
```

- 根据特殊参数推导机制，Arg 的类型为 X&，于是实例化的函数模板如下

```
shared_ptr<A> factory(X& &&arg){
    return shared_ptr<A>(new A(std::forward<X&>(arg)));
}
X& && forward(typename remove_reference<X&>::type& a) noexcept{
    return static_cast<X& &&>(a);
}
```

- 通过 remove_reference 以及引用折叠规则之后，变为如下

```
shared_ptr<A> factory(X& arg){
    return shared_ptr<A>(new A(std::forward<X&>(arg)));
}
X& forward(typename X& a) noexcept{
    return static_cast<X&>(a);
}
```

- 通过以上步骤完美转发左值：factory 的参数通过两次间接的传递将实参传递给 A 的构造函数，两次都是通过左值引用

- 考虑以下调用

```
X foo();
factory<A>(foo());
```

- 根据特殊参数推导机制，Arg 的类型为 X，于是实例化的函数模板如下

```
shared_ptr<A> factory(X&& arg){
    return shared_ptr<A>(new A(std::forward<X>(arg)));
}
X&& forward(X& a) noexcept{
    return static_cast<X&&>(a);
}
```

- 右值的完美转发：factory 的参数通过两次间接的传递将实参传递给 A 的构造函数，两次都是通过引用
- 值得注意的是，std::forward 的唯一目的就是保留移动语义，在没有 std::forward 时，一切类型信息都是完备的，除了 A 的构造函数会将 arg 视为一个左值，因为这个东西具有名字，因此它是左值
- remove_reference 在 forward 的定义中是必须的吗？**不是必须的(因为 forward 的模板参数 S 是显式指定而非推导出来的，最后返回的就是 S&&)**，如果用 S&代替 remove_reference<S>::type&，完美转发仍然成立。所以 remove_reference 的目的就是让我们一定要这样做，明确一下

```
template<class S>
S&& forward(S& a) noexcept{
    return static_cast<S&&>(a);
}
```

4、std::move 的定义如下

```
template<class T>
typename remove_reference<T>::type&&
std::move(T&& a) noexcept{
    typedef typename remove_reference<T>::type&& RvalRef;
    return static_cast<RvalRef>(a);
}
```

- 对于左值调用

```
X x;
std::move(x);
```

- 根据特殊参数推导机制，T 的类型为 X&，于是实例化的函数模板如下

```
typename remove_reference<X&>::type&&
std::move(X& && a) noexcept{
    typedef typename remove_reference<X&>::type&& RvalRef;
    return static_cast<RvalRef>(a);
}
```

- 通过 remove_reference 以及引用折叠规则之后，变为如下

```
X&& std::move(X& a) noexcept{
    return static_cast<X&&>(a);
}
```

- move 将左值变为一个无名右值引用类型，即右值

- 对于右值调用

```
X foo();
std::move(foo());
```

- 根据特殊参数推导机制，T 的类型为 X，于是实例化的函数模板如下

```
typename remove_reference<X>::type&&
std::move(X && a) noexcept{
    typedef typename remove_reference<X>::type&& RvalRef;
    return static_cast<RvalRef>(a);
}
```

- 通过 remove_reference 以及引用折叠规则之后，变为如下

```
X&& std::move(X a) noexcept{
    return static_cast<X&&>(a);
}
```

- move 将仍然返回右值

## 24.9. Rvalue References and Exceptions

1、在重载拷贝构造函数和拷贝赋值运算符的移动版本时，需要注意以下两点

1) 尽量以无异常的方式定义这两个函数，因为实现移动语义通常所作的事情就是交换两个指针
2) 如果你成功地以无异常的方式实现你的移动版本，那么加上关键字 noexcept

2、如果不这么做，那么你所希望的移动语义可能不会发生，例如 std::vector 在进行扩容时，会将现有的元素挪到新的区域，如果 1 或 2 不满足，则移动语义不会发生

## 24.10. The Case of the Implicit Move

1、当程序员不提供构造和赋值的移动版本时，编译器不会为我们自动生成，因为它会对现有的代码造成相当大的破坏

2、在日常编程中，需要注意以下三点

1) 移动语义最好仅仅用在构造函数和赋值运算符的实现上，最好不要定义如下的重载函数(难道我们称它为接受左值的版本和接受右值的版本??)

```
void foo(X& x);
void foo(X&& x);
```

2) std::move 将参数转变为一个右值
3) std::forward 帮助你实现完美转发

# Chapter 25. 并发

## 25.1. 简介

1、C++11 开始支持多线程编程，之前多线程编程都需要系统的支持，在不同的系统下创建线程需要不同的 API 如 pthread_create()，Createthread()，beginthread()等，使用起来都比较复杂

2、C++11 提供了新头文件<thread>、<mutex>、<atomic>、<future>等用于支持多线程

- thread 定义在 std 命名空间中

3、C++11 支持 Lambda 表达式，因此一个新线程的**回调函数**也可以是有一个 Lambda 表达式的形式

- 但是注意如果使用 Lambda 表达式，最好不要使用引用的方式，应该使用值传递的方式来访问数据，避免在多线程中使用时造成混乱。

4、例子

```
int main(){
    vector< thread> threads;
    for(int i = 0; i < 5; ++i){
        threads.push_back(std::thread([](){
            cout << "Hello from lamda thread " << this_thread::get_id() << endl;
        }));
    }
    for(auto& thread : threads){
        thread.join();
    }
    cout<<"Main Thread"<<"\t"<< this_thread::get_id()<<endl;
    return 0;
}
```

## 25.2. thread 回调成员函数的方法

1、例子

```
class threadTest{
public:
    void f(int i,int j,int k){
        cout<<"i= "<<i<<", j= "<<j<<", k="<<k<<endl;
    }
};
int main(int argc, const char * argv[]) {
    threadTest tt;
    thread t(&threadTest::f,tt,1,2,3);
    t.join();
    return 1;
```

}

- 第一个参数是方法指针，第二个参数是实例对象

## 25.3. 锁机制

1、与 Java 不同，Java 将同步机制建立在 Object 对象上，即任意 Java 对象都可以作为公共资源；C++必须额外采用独特的类型，即 mutex(互斥量)来进行线程之间的同步

2、几个核心类

1) mutex、recursize_mutex、timed_mutex、recursive_timed_mutex
2) lock_buard、unique_lock、
3) condition_variable

### 25.3.1. mutex

1、方法介绍

1) void lock();
2) void unlock();
3) bool try_lock();
4) void native_handle();

### 25.3.2. unique_lock

1、方法介绍

1) explicit unique_lock<mutex>(_Mutex& _Mtx):_Pmtx(&_Mtx),_Owns(false)
   {
   _Pmtx->lock();
   _Owns=true;
   }
   - 可以看出，一旦 unique_lock 一旦构造完成，即调用传入的 mutex 的 lock 方法进行锁定
2) _Mutex *mutex();//返回持有的 mutex
3) _Mutex *release();//释放锁??返回原先持有的 mutex
4) void lock();//锁定
5) void unlock();//解锁

### 25.3.3. condition_variable

1、方法介绍

1) void wait(unique_lock<mutex>&);// 挂起当前线程，并等待指定的 unique_lock 对象
2) void notify_one();随机唤醒等待锁(什么锁?)的一个线程
3) void notify_all();唤醒所有等待锁(什么锁?)的线程

2、注意

- 一个 condition_variable 可以对不同的 unique_lock<mutex>调用 wait，在这种情况下，如果调用了 void notify_one()/void notify_all()，那么会唤醒等待所有这些传入 wait 的锁的线程中的一个或者全部

- 我以为，一个 condition_variable 在一般情况下，仅对同一个 mutex 操作会比较好

# Chapter 26. 杂项

## 26.1. 变量、变量名、地址

1、变量就是一个内存空间，它有地址，也有变量名
2、**所有的变量名，编译后在可执行程序里都是单纯的内存地址**
3、**放在栈中的变量就是单纯的进栈出栈，里面的地址是连续的，不需要申请也不需要释放，移动栈顶指针就OK**
4、**放在堆中的地址就需要申请和释放**
5、比如 int a=3 //这个变量的地址是 0x12345678，它的名字是 a，值是 3，名字 a 只是该变量的代号，当编译器看到 a 时就会找到这个变量的地址 0x12345678 然后进行后续操作

## 26.2. 闭包

### 26.2.1. 闭包的定义

1、闭包是带有上下文的函数。说白了，就是有状态的函数。
2、更直接一些，不就是个类吗？换了个名字而已(Java 中的非 static 内部类就是一个面向对象的闭包)
3、**一个函数，带上了一个状态，就变成了闭包了**
4、什么叫"带上状态"呢?意思是这个闭包有属于自己的变量，这些个变量的值是创建闭包的时候设置的，并在调用闭包的时候，可以访问这些变量
5、**函数是代码，状态是一组变量**，将代码和一组变量捆绑(bind)，就形成了闭包，内部包含 static 变量的函数，不是闭包，因为这个 static 变量不能捆绑。你不能捆绑不同的 static 变量，这个在编译的时候已经确定了
6、**闭包的状态捆绑必须发生在运行时**

### 26.2.2. C++实现闭包的方法

1、重载 operator()
- 因为闭包是一个函数+一个状态，这个状态通过隐含的 this 指针传入
- 所以闭包必然是一个函数对象。**因为成员变量就是极好的用于保存状态的工具**
- 因此实现 operator()运算符重载，该类的对象就能作为闭包使用
- **默认传入的 this 指针提供了访问成员变量的途径**

2、lambda
- c++11 里提供的 lambda 表达式就是很好的语法糖。
- 其本质和手写的函数对象没有区别。
- 当我们编写一个 lambda 后，编译器将该表达式翻译成一个未命名类的未命名对象
- lambda 表达式产生的类中含有一个重载的函数调用运算符，**默认情况下，由 lambda 产生的函数调用运算符是一个 const 成员函数**，如果 lambda 被声明为可变的，则函数调用运算符就不是 const 的了
- 捕获列表：

- 当一个 lambda 表达式通过引用捕获变量时，将由程序确保 lambda 执行时引用所引的对象确实存在，因此编译器可以直接使用该引用而无须在 lambda 产生的类中将其存储为数据成员
- 当一个 lambda 表达式通过值捕获变量时，将在 lambda 产生的类中为每个值捕获的变量建立对应的数据成员，同时创建构造函数，令其使用捕获的变量的值来初始化数据成员

3、boost::bind/std::bind

- 标准库提供的 bind 是更加强大的语法糖，将手写需要很多很多代码的闭包，浓缩到一行 bind 就可以搞定了

## 26. 3. 类相互包含的解决方法

1、在类的声明之后，定义之前，该类型是一个不完全的类型，无法定义对象，因为本根无法分配内存

- 此时该类的应用场景仅限以下情形
  - **可以定义指向这种类型的指针或引用**
  - **可以声明(但是不能定义)以不完全类型作为参数或返回类型的函数**
- 因此我们应该在每个类中定义一个指针指向另一种类型，而不能在类中定义另一个类型的对象(可以定义另一个类的对象，但是最多只能一个，此时另一个类必须用引用或者指针类型指向该类型)

2、头文件的一种结构如下

a.h：

```
#include"b.h"
class A{
    B* b;    //这里可以是 B 的对象，未必要是指针
    ...
}
```

b.h：

```
class A;//前置声明
class B{
    A *a;    //这里必须是指针或引用，因为 A 对象的定义尚未出现
    ...
}
```

- A 类的头文件 a.h 含有 B 类的头文件 b.h
- B 类的头文件，只有 A 类的声明
- 于是头文件不存在循环的包含

3、头文件的另一种结构如下

a.h：

```
class B
class A{
    B* b;    //这里可以是 B 的对象，未必要是指针
    ...
}
```

b.h：

```
class A;//前置声明
class B{
    A *a;    //这里必须是指针或引用，因为 A 对象的定义尚未出现
    …
}
```

- 在这种结构中，a.h 和 b.h 都不包含另一个头文件，仅仅前置声明了另一个类，因此在两个头文件中，另一个类型都是不完整的
- 因此，我们在头文件中不能存在定义或者使用另一个对象的代码
  - 以 a.h 为例
  - 不能定义 B 类的对象
  - **可以定义指向 B 类的指针，但是在该头文件中，不能使用该指针访问 B 类的域或者方法**
- 在 a.cpp 与 b.cpp 中，我们只需要包含两个头文件即可，这样是目前为止处理类相互包含最优雅的方式
- 应用场景：
  - 对于 LTEV2X 的设计，有一个系统类 System，与几个模块类 Model1、Model2...
  - 在模块类中有这样的需求：需要获取系统类的指针，从而获取操纵其他模块的能力
  - 在系统类中自然需要包含这几个模块类
  - 于是，System 包含指向各个模块的指针，各个模块包含指向 System 的指针

## 26.4. 声明和定义中的关键字

### 26.4.1. 声明和定义都必须注明

1、noexcept
2、&
3、&&
4、const

### 26.4.2. 只需要声明注明：

1、explicit
2、static
3、virtual
4、override

### 26.4.3. 只需要定义注明：

1、inline