C++Primer笔记

第一章 开始

一个简单的C++程序:

```
int main()
{
    return 0;
}
```

初识输入输出

一个使用IIO库的程序

```
#include <iostream>
int main()
{
    std::cout<<"Enter two numbers:"<<std::endl;
    int v1 = 0 ,v2 = 0;
    std::cin>>v1>>v2;
    std::cout<<"The sum of"<<v1<<"and"<<v2<<"is"<<v1+v2<<std::endl;
    return 0;
}
```

输入运算符：>>

输出运算符：<<

操纵符：endl

[^]操纵符的作用是结束当前行，并刷新缓冲区。

> 注释简介：
>
> 单行注释：
>
> ```
> //这是一个单行注释
> ```
>
> 多行注释：
>
> ```
> /*
> *这是一个多行注释
> */
> ```

控制流

```
#include <iostream>
int main()
{
    int sum =0,val = 1;
    while (val<10)
    {
        sum += val;
        ++val;
    }
    std::cout<<"Sum of 1 too 10 inclusive is "<<sum<<std::endl;
    return 0;
}
```

for循环：

```
for (type val_name;condiction;expression)
{
    /*
    *code
    */
}
```

if语句

```
if(tyoe val_name;condiction)
{
    /*dode*/
}
```

> C++程序的所进和格式
>
> C++会跟大程度上在格式上是自由的。任何的缩进，防止花括号都不会影响程序的语义，但是会影响代码的可读性。

类简介

第I部分 C++基础

第二章 变量和基本类型

2.1 基本内置类型

C++定义了一套基本的**算术类型**和**空类型**在内的基本数据类型。算是类型包括了字符、整型数、布尔值和浮点数。空类型不对应具体的数值，仅用于一些特殊的场合。

算术类型

算术类型主要分类两类：**整型**和**浮点型**

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
char_16_t	Unicode字符	16位
chat32_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	长整形	64位
float	单精度浮点类型	6位有效数字
double	双精度浮点类型	10位有效数字
long double	扩展精度浮点类型	10位有效数字

基本字符集的类型为char，其他扩展字符集一般用wchat_t、char16_t、char32_t类型。

> ## 带符号类型和无符号类型
>
> 除去布尔类型和扩展的字符型之外，其他整型还可以划分为带符号整型和无符号整型两种，同C。

类型转换

类型转换即把一种数据类型转换成另一种数据类型，除非有必要这么做，否则不建议乱转换，会容易造成数据丢失的。

- 当我们把一个非布尔类型的算术值赋给布尔类型时，初始值为 0 则结果为 false，否则结果为 true。
- 当我们把一个布尔值赋给非布尔类型时，初始值为 false 则结果为 0，初始值为 true 则结果为 1。
- 当我们把一个浮点数赋给整数类型时，进行了近似处理。结果值将仅保留浮点数中小数点之前的部分。
- 当我们把一个整数值赋给浮点类型时，小数部分记为 0。如果该整数所占的空间超过了浮点类型的容量，精度可能有损失。
- 当我们赋给无符号类型一个超出它表示范围的值时，结果是初始值对无符号类型表示数值总数取模后的余数。例如，8 比特大小的 unsigned char 可以表示 0 至 255 区间内的值，如果我们赋了一个区间以外的值，则实际的结果是该值对 256 取模后所得的余数。因此，把-1 赋给 8 比特大小的 unsigned char 所得的结果是 255。
- 当我们赋给带符号类型一个超出它表示范围的值时，结果是**未定义的**（undefined）。此时，程序可能继续工作、可能崩溃，也可能生成垃圾数据。

> ## 避免无法预知和依赖于实现环境的行为
>
> 无法预知编译器的行为源于编译器无须检测的错误，即使代码通过了，如果程序执行了一条未定义的表达式，仍可能产生错误。
>
> 所以程序员应该尽量避免依赖于实现环境的行为·。

> ## 切勿混用带符号i整形和无符号整型
>
> 如果在一个表达式中出现带符号整形和无符号整型，在运算过程中带符号整形可能会莫名其妙出现负值的情况。

字面值常量

转义字符用于在打印无法直接打印的字符时做替换使用

指定字面值的类型：

通过添加前后缀，可以改变**整型、浮点型和字符型**字面值的默认类型

```
L'a'     //宽字符型字面值，类型是wchar_t
u8“hi!” //utf-8字串符字面值
42ULL    //无符号i整型字面值，类型是unsigned long long
1E-3F    //单精度浮点类型字面值，类型是float
3.14159L     //扩展精度浮点类型字面值，类型是long double
```

表 2.2：指定字面值的类型

字符和字符串字面值		
前缀	含义	类型
u	Unicode 16 字符	char16_t
U	Unicode 32 字符	char32_t
L	宽字符	wchar_t
u8	UTF-8（仅用于字符串字面常量）	char

整型字面值		浮点型字面值	
后缀	最小匹配类型	后缀	类型
u or U	unsigned	f 或 F	float
l or L	long	l 或 L	long double
ll or LL	long long		

布尔字面值和指针字面值

true和false是布尔类型的字面值：

```
bool test = false;
```

`nullptr` 是指针的字面值。

变量

变量是一个提供具名的、可供程序操作的存储空间。

变量定义

变量定义的方法是先声名**类型说明符**，再说明变量名，最后在赋值。[记得以分号结束]

列表初始化

```
int units_sold=0;
int units_sold = {0}
int units_solds(0);
int units_solds{0};
```

默认初始化

如果在定义了一个变量之后没有给它赋值，那么它会直接被**默认初始化。**

> ### 未初始化变量引起的运行时的故障
>
> 未初始化的变量含有一个不确定的值，使用未初始化变量的值是一个错误的编程行为并且很难调试。尽管大多数编译器并不会报错，但是严格来说，是因为编译器并未被要求做此种检查。
>
> 未初始化的变量会导致无法预知的后果。真的别这么做！！（除非你想被炒鱿鱼）

变量声明和定义的关系

分离式编译：C++允许把程序拆分成多个逻辑部分进行编写，即支持分离式编译，该机制允许将程序分割成若干个文件，每个文件都可以被独立编译。

声名使得该变量被程序所知,而定义则负责创建与名字关联的实体。

> ### 关键概念：静态类型
>
> C++是一种静态类型语言，其含义是在编译阶段检查类型。其中，检查类型的过程称为*类型检查*。
>
> 程序越复杂，静态类型语言越容易发现错误。然而前提是你的编译器知道每一个实体对象的类型，这就要求我们在使用某个变量之前必须声明其类型。

标识符

C++的标识符是有字母、数字和下划线组成的，其中必须以字母或者下划线开头。

变量命名规范：：

- 标识符要能体现实际含义；
- 变量名一般用小写字母；
- 用户自定义的类名一般用大写字母开头；
- 如果标识符有很多个单词组成，则单词之间应有明显区分；

表 2.3：C++关键字

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

表 2.4：C++操作符替代名

and	bitand	compl	not_eq	or_eq	xor_eq
and eq	bitor	not	or	xor	

名字的作用域

作用域是程序的一部分，在其中名字有特定的含义。C++中大多数作用域以花括号分隔。

全局作用域:名字定义于main函数的所有花括号之外的作用域。

块作用域：即只存在于函数内的作用域，通常为块状。

嵌套函数的作用域

作用于能彼此嵌套，被包含的作用域被称为内层作用域，包含着别的作用于的作用域称为外层作用域。

> ### 引用
>
> C++11中新增了一种引用：所谓的“右引用”，（具体的将在13章涉及到）。这种引用主要用于内置类。严格来说，当我们使用数域“引用”时，指的一般是“左值引用”。

```
1 int ival = 1024;
2 int &refVival =ival;    //refVal指向ival
```

引用即别名

> 应用并非对象，相反的，它只是为一个已存在的对象所起的另外一个名字。

为了引用赋值，实际上是把值赋给了与引用绑定的对象。获取引用的值，实际上是获取了与引用绑定的对象的值1.

指针

指针是“指向”另外一种类型的复合类型。与引用类似。

指针本身就是一个对象，允许指针赋值或者拷贝。但是指针允许在定义是赋初始值，和其他内置类型一样，在块作用域内定义的指针如果没有初始化，也将拥有一个不确定的值。

> ### 获取对象的地址
>
> 若要获取某个对象的地址，则需要取地址符“&”。

指针值：

- 指向一个对象
- 指向紧邻对象所占空间的下一个位置
- 空指针，意味着没有指向任何对象
- 无效指针，也就是上述情况之外的值

利用指针访问对象

一般使用解引用符(*)来访问该对象

```
int ival = 42;
int *p = &ival;
cout <<*p;
```

空指针不知向任何对象，下面列出几种生成空指针的方法：

```
#include <cstdlib>
int *p1 =nullptr;
int *p2 =0;
int *p3 =NULL;
```

> 对于指针的初始化是一件很重要的事情，如果使用一个未经初始化的指针将会有非常严重的后果。
>
> 所以建议在指针刚出来的时候就给他变为空指针或者0，免得报错。

赋值和指针：

对于指针来说，它不同于引用，可以给指针赋上新的地址值，从而指向一个新的对象。

其他指针操作：

- 只要指针拥有一个合法的值，就能在条件表达式中。和采用算数值为条件遵循的规则类似，如果指针的值是0，条件取false。
- 对于两个类型相同的合法指针，

void* 指针

void*是一种特殊的类型指针，可以用相等操作符（==）或不等操作符（!=）来比较它们。

void* 指针能干的事很少，基本上只能比较地址值、传参，不能直接去操作只想的变量。

复合类型声明

即在使用多种修饰符时一定要注意好放置位置，否则极易产生误解！！

```
int* p;//这就是一个错误的放置位置，让人误解为存在int*变量类型
```

指向指针的指针（套娃开始）

通过*的个数可以区分指针的级别.。也就是说，**表示指向指针的指针，***是指向指针的指针的指针，依此类推。

```
int *ival =1024;
int *pi = &ival;     //指向指针的指针
int **ppi =&&pi;     //指向一个int 型的指针
```

指向指针的引用

引用本身不是一个对象，因此不能定义指向引用的指针，所以存在指向指针的引用。

```
int i = 42;
int *p; //p是一个int型指针
int *&r = p;     //r是一个对指针p的引用
r = &i;          //r引用了一个指针，因此给r赋值&i就是令p指向i
*r = 0;           //解引用r得到i，也就是p指向的对象，将i的值改为0
```

const 限定符

const限定符可以对变量的类型加以预定：

```
const int bufsize = 512;     //输入缓冲区大小
```

对const的引用可能引用一个非const的对象

必须认识到，常量引用仅对引用可参与操作符做出了限定，对于引用的对象本身不是一个常量未作限定，因为对象也有可能是一个非常量。所以允许通过过其他途径改变它的值。

指针和const

指向常量的指针不能用于修改所指对象的值。（类似于查询）要想存放常量对象的地址，只能使用只想常量的指针。

```
const double pi = 3.14; //pi是一个常量，它的值不能改变
double *ptr = &pi;  //错误，ptr是一个普通指针
const double *cptr = &pi;    //正确；cptr可以指向一个双精度常量
*ctpr =42        //错误，不能给*cptr赋值
```

const指针

常量指针必须初始化,而且一旦初始化完成，则它的值（就是存放在指针的那个地址）不会再改变了。

顶层const

顶层const表示指针本身就是一个常量，**底层const**表示指针所指的对象是一个常量。

更一般的，顶层const可以表示任意的对象是常量，这一点对所有的类型都适用。

底层const则与指针和引用等复合类型的基本部分有关。

```
int i = 0;
int *const p1 = &i;      //不改变p1的值，这是一个顶层的const
const int ci = 42;       //不能改变ci的值，这是一个顶层的const
const int *p2 =  &ci;    //允许改变p2的值，这是一个底层的const
const int *const p3 = p2;    //靠右的const是顶层const，靠左的是底层const
const int &r = ci;       //用于声明引用const都是底层const·
```

constexpr和常量表达式

常量表达式是指不会改变并且在编译过程就能得到计算结果的表达式。

```
const int max_files = 20;//max_files是常量表达式
const int limit = max_flies + 1;//limit是常量表达式
int staff_size = 27;    //staff_size不是常量表达式
const int sz get size();    //sz不是常量表达式
```

---

constexpr变量

C++11新标准允许吧变量声明为**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*。

```
constexpr int mf = 20;           //20是常量表达式
constexpr int limit = mf +1      //只有mf+1是常量表达式
const int sz =size();            //只有当size是一个constexpr函数时才是一条正确的声明语句
```

> 一般来说，如果你需要定义变量是一个常量表达式，那就把它声名成constexpr类型

指针和constexpr

与其他指针相比，constexpr指针**既可以指向常量又可以指向变量**·。

```
constexpr int *p = nunllptr;    //np是一个指向整数的常量指针，其值为空
int j = 0;
constexpr int i = 42;           //i的类型是整型常量
constexpr const int *p = &i     //p是常量指针，指向整数常量i
constexpr int *p1 = &j;         //p1是指针常量，指向数为
```

处理类型

类型别名

类型别名是一个名字，可以较简洁的使复杂的类型名字变得易于理解和使用。

有两种方法用于定义变量名：

1. 使用关键字**typedef**

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

另一种新方法，使用**别名声名**来定义类型的别名

```
using SI = Sales_item;      //等价于double hourly、weekly
SI tiem;                    //等价于Sales_item item
```

指针、常量和别名

如果某个类型别名指代的是复合类型或常量，那么把它用到声明类型语句里就会产生意想不到的后果。

auto类型说明符

auto类型说明符可以让编译器去分析表达式所属的类型。

```
//由val1和val2相加的结果来判断item的类型
auto item = val1 + val2;        //item初始化为val1和val2相加的结果
```

复合类型、常量和auto

编译器推断出来的auto类型有时候和你的初始值类型不一样，编译器会适当地改变结果类型使其更符合初始化原则。

其次，auto一般会忽略掉顶层的const，同时底层const则会完全保留下来。

```
const int ci = i, &cr = ci;
auto b = ci;    //b是一个整数(ci的顶层const特性被忽略掉了)
auto c = cr;    //c是一个整数(cr是ci的别名，ci本身十一的顶层const)
auto d = &i;    //d是一个整形指针(整数的地址就是指向整数的指针)
auto e = &ci;   //e是一个指向整数常量的指针(对常量对象取地址是一种底层const)
/*
*如果希望推断出auto类型是一个顶层const，需明确指出
*/
const auto f = ci;  //ci的推演类型是int，f是const int
```

decltype类型指示符

该说明符的作用是可以选择并返回操作数的数据类型。在此过程中，编译器会分析并得到它的类型，却不实际计算表达式的值：

```
decltype (f())sum = x;
```

编译器并未实际调用f，而是当调用发生时f的返回值类型作为sum的类型。

decltype和引用

如果decltype使用的表达式不是一个变量，则decltype返回表达式结果对应的类型。

```
//decltype的结果可以是引用类型
int i =42,*p =&i,&r =i;
decltype(r+0) b//正确，加法是int，因此b是一个(未初始化）的int
decltype(*p)c   //错误，c是int。必须初始化
```

decltype和auto的区别是

- decltype的结果类型与表达式形式密切相关。

```
decltype((i))  d;       //错误，d是一个int，必须初始化
decltype(i) e;          //正确，e是一个（未初始化的）int
```

自定义数据结构

C++用户允许用户以类的形式2自定义数据类型，而库类型string、istream、ostream等也都是以类的形式定义的。

初等例子：定义Sales_data类型

```
struct Sales_data
{
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
}
```

类数据成员

类体定义类的成员，其中有很多种类型，目前我们只有数据成员

C++11标准规定，可以为数据提供一个**类内初始值。**

对类内初始值的限制：切记不能使用圆括号，并且要放在等号右边。

使用Sales_data类

```
#include<iostream>
#include<string>
#include"Sales_data.h"
int main()
{
    Sales_data data1,data2;
    double price = 0;
    std::cin>>data1.bookNo>>data1.units_sold>>price;
    data1.revenue =data1.units_sold*price;
    std::cin>>data2.bookNo>>data2.units_sold>>price;
    data2.revenue = data2.units_sold*price;
    if (data1.bookNo == data2.bookNo)
    {
        unsigned totalCnt = data1.units.sold+data2units.sold;
        double totalRevenue = data1.revenue+data2.revenue;
        std::cout<<data1.bookNo<<" "<<totalCnt<<" "<<totalRevenue<<" ";
        if(totalCnt != 0)
        {
            std::cout <<                        <<std:endl;
        }
        else
        {
            std::cout<<"(no sales)"<<std::endl;
            return 0;
        }
        else
        {
            std::cout<<"Data must refer to the ISBN"<<std::endl;
            return -1;
        }
```

```
    }
    //读入data1和data2的代码
    //检查data1和data2的ISBN是否相同
    //如果相同，求data1和data2的和
}
```

编写自己的头文件

> 在编写头文件时，一旦使就不要乱改变头文件的内容，如果该等了，相关的代码也得改。

预处理器概述

确保头文件人能安全工作的常用技术是**预处理器**技术

```
#ifdef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data
{
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

> 预处理变量无视C++语言中关于作用域的规则。

第三章 字符串、向量和数组

命名空间的using声名

声名命名空间的形式：

```
using namespace::name;
```

一旦声明了上述语句，就可以直接访问命名空间中的名字。

```
#include <iostream>
using std::cin;
int main()
{
    int i;              //正确，cin和std::cin含义相同
    cin>>i;             //
    cout<<i;
    std::cout<<i;
    return 0;
}
```

每个名字都要使用using单独命名

按照规定，每个using声名引入命名空间中的一个成员。

```cpp
#include <iostream>
using std::cin;
using std::cout;using std::endl;
int main()
{
    cout<<"ENter two nmumbers"<<endl;
    intr v1,v2;
    cin>>v1>>v2;
    cout <<"The sum of " << v1+v2 << "and"<<
}
```

在上述程序中，一开始有对cin、cout、和endl的using声明，意味着2我们可以不用再添加std::的形式前缀就能直接使用它们。

> 头文件不应该包using声明，因为如果处理不好，容易造成命名空间重名的情况

标准库类型string

标准库string表示可变长的字符序列，使用string类型必须首先包含string头文件。

```cpp
#include <string>
using std::string;
```

定义和初始化string对象

```cpp
string s1;      //默认初始化，s1是一个空字符串
string s2 =s1;   //s1是s1的副本
string s3 ="hiya"   //s3是该字符串字面值的副本
string s4(10,'c')   //s4的内容是ccccccccccc
```

初始化string对象的方式

```cpp
string s1               //默认初始化，s1是一个空字符串
string s2(s1)           //s2是s1的副本
string s2 = s1          //等价于s2(s1),s2是s1的副本
string s3 ("value")     //s3是字面值“value”的副本，除了字面值最后的那个空字符串外
string s3 = "value"     //等价于s3("value"),s3是字面值"value"的副本
string s4(n,'c')        //把s4初始化为由n个连续的字符c组成的字符串
```

直接初始化和拷贝初始化

使用等号初始化一个变量，实际上执行的是**拷贝初始化**，如果不使用等号，则执行的是**直接初始化。**

当初值只有一个时，直接拷贝初始化比较方便

当初始值比较多的时候，不太建议用拷贝初始化，太麻烦，直接初始化反而比较有优势。

string对象上的操作

string的操作

```
os<<s                   //将s写到os当中，返回os
is>>s                   //从is中读取字符串赋值给s，字符串以空白字符分隔，返回is
getline(is,s)           //从is中读取一行赋值给s，返回is
s.empty()               //s为空返回ture,否则返回false
s.size()                //返回s中的字符的个数
s[n]                    //返回s中第n个字符的引用，位置n从0开始计起
s1+s2                   //返回s1+s2链接后的结果
s1=s2                   //用s2的副本代替s1中的原来的字符
s1==s2                  //如果s1和s2中所含的字符完全一样，则它们相等string对象的和等性判断对字母大小写敏感
s1!=s2
<.<=,>,>=               //利用字符串在字典中的顺序进行比较，且对字母的大小写敏感
```

读写string对象

还可以使用IO操作来读写string对象

```
//注意：要想编译下面的代码，还要适当的#include语句和using声名
int main()
{
    string s;           //空白字符
    cin>>s;             //将string对象带入，遇到空白字符停止
    cout<<s<<endl;      //输出s
    return 0;

}
```

读取未知数量的string对象

```
int main()
{
    string word;
    while(cin>>word)
        cout<<word<<endl;
    return 0;
}
```

使用getline()读取一整行

```
int main()
{
    string line;
    while (getline(cin,line))
        cout<<line<<endl;
    return 0;
}
```

stringf的empty操作

```
//输出非零行:
while(getline(cin,line))
    {
        if(!line.empty() )
            cout<<line<<endl;
    }
```

string::size_type操作

string::sizetype 是一个无符号类型的值，并且能放得下任何string的大小

现在(C++11)可以通过auto或者decltype来确定string::size_type的类型

比教string对象

一般用相等运算符(==)和不等运算符(!=)来检验string对象相等或者不等。

挂你运算符》，>,>=,<,<=分别用于检验两个字符串的对象长度的大小

上述这些运算符都依照(大小写敏感的)字典顺序:

> 1. 如果两个string对象长度不同，且较短的string对象的每个字符都与较长的·1string对象所对应位置上的字符相同，就说较短的string对象小于较长string对象;
> 2. 如果lianggestring对象在某些·1对应的位置上不一致，则string对象比较的结果其实是string对象中第一队相异字符比较的结果

示例:

```
string str ="Hello";
string phrease ="Heaal world";
string slang "Hiya";
```

给string对象赋值

```
string st1(10,'c'),stl2;     //st1的内容是ccccccccccc，st2是一个空字符串
st1 = st2;                   //赋值：用st2的副本
```

两个string对象相加

```
string s1= "Hello,",s2 = "world\n";
string s3 =s1+s2;
s1+=s2;
```

字面值和string对象相加

```
string s1 = "hello",s2 ="world\n";
string s3 =s1+","+s2+'\n';
```

处理string对象中的字符

> # cctype 头文件中的函数

```
isalnum(c)          //当c是字母时为真
isalpha(c)          //当c是字母时为真
isc吐rl(c)           //当c是控制字符时为真
is尘g江(c)            //当c是数字时为真
isgraph(c)          //当c不是空格但可打印时为真
islower(c)          //当c是小写字母时为真
isprint(c)          //当c是可打印字符时为真（即c是空格或c具有可视形式）
ispunct(c)          //当c是标点符号时为真（即c不是控制字符、数字、字母、可打印空白中的 一种）
isspace(c)          //当c是空白时为真（即c是空格、横向制表符、纵向制表符、回车符、换行 符、进纸符中的一种）
isupper(c)          //当e是大写字母时为真
isxdigit(c)         //当c是十六进制数字时为真
tolower(c)          //如果c是大写字母，输出对应的小写字母：否则原样输出c
toupper(c)          //如果c是小写字母，输出对应的大写字母：否则原样输出c
```

建议：使用C++版本的C标准头文件

C++标准库中除了定义C廿语言特有的功能外，也兼容了C语言的标准库。C语言 的头文件形如name上，C++ 则将这些文件命名为cname。也就是去掉了 .h后缀，而在 文件名name之前添加了字母c, 这里的c表示这是一个属于C语言标准库的头文件。

因此，cctype头文件和c七ype.h头文件的内容是一样的，只不过从命名规范上 来讲更符合C++语言的要求。特别的，在名为cname的头文件中定义的名字从属于命名 空间std, 而定义在名为.h的头文件中的则不然。

一般来说，C++程序应该使用名为cname的头文件而不使用name.h的形式，标准 库中的名字总能在命名空间std中找到。如果使用.h形式的头文件，程序员就不得不 时刻牢记哪些是从C语言那儿继承过来的，哪些又是C++语言所独有的。

如果相对每个字符都进行操作，最好使用for循环进行操作，其语法格式为：

```
for (declaration:expresion)
    statement
```

举个简单的例子

```
string str("some string");
for (auto c:str)
    cout<<c<<endl;
```

这个直接通过遍历的方法，通过遍历循环直接输出c中的字符串

```
string s("Hello world!!!");
    decltype(s.size()) punct_cut=0
    for (auto c:s)
        if (ispunct(c))
            ++punct_cut;
cout<<punct_cut<<"punctuation charateristic in"<<s<<endl;
```

只处理一部分字符?

通过下标来处理是一种比较简便的方法，这个参数表示字符串在里面所处的位置

下标运算符（"[]"）

标准库类型vector

标准库类型vector表示对象的集合，其中所有类型的 都相同。所以也常常称它为**容器。**

使用方法：·

```
#include<vector>
using std::vector;
```

C++中既有**类模板**，又有**函数模板**

编译器根据模板创建类或函数的过程称为**实例化。**

```
vector <int> ivec;  `                    //ivec保存int类型元素
vector<Sales_item> Sales_vec;            //保存Sales_vec
vector <vectorn<string>> file            //该向量的元素是vector对象
```

列表初始化vector对象

C++11新标准还提供了另外一种名为vector对象元素的赋初值的方法，即列表初始化

```
vector <string> articles = {"a","an","the"};
```

创建指定数量的元素

```
vector <int> ivec (10,-1);       //10个int类型的元素，每个都被初始化为-1
vector <string> svec(10,"Hi!")   //10个string类型的元素，每个都被初始化为“Hi！”
```

列表初始值还是元素初始值?

通过使用花括号和圆括号来区分是元素数量还是初始值：

```
vector<int> v1(10);      //v1有10个元素，每个值都是0
vector<int> v2{10};      //v2有1个元素，该元素的值为10
/*多元素的情况*/
vector<int> v3(10,1)     //v3有10个元素，每个元素的值都是1
vector<int> v4{10,1}     //v4有两个元素，分别是10和1
```

> ## 关键概念：vector对象高效增长
>
> 对于C++来说，vector能在运行时高效快速地添加元素，因此既然vector为此诞生的，所以在定义vector对象的时候设定其大小就没什么必要了。

向vector对象中添加元素必须确保循环准确无误，特别是在改变vector对象容量的时候.

其他vector操作

```
v.empty()         //如果v不含任何元素，则返回真，否则返回假
v.size()          //返回v中的元素个数
v.push_back(t)    //向v中添加一个值为t的元素
v{n}              //返回v中的第n个位置上的元素引用（注意返回的是引用！！）
v1 = v2           //用v2中的元素拷贝替换v1中的元素
v1 ={a,b,c,...}   //用列表中的元素拷贝替换v1中的元素
v1 == v2          //v1和v2相等当且仅当他们的元素数量相同且对应位置的元素值相同
v1 != v2          //反运算
<,<=,>,>=         //以字典的顺序进行比较
```

> 要使用size_type,需首先指定它是由哪种类型定义的。
>
> ```
> vector <int>::size_type      //正确
> vector::size_type            //错误
> ```

注：不能使用下标的形式来添加元素，不然汇报做，只能用下标来访问元素，毕竟计数器是单独出来的

迭代器：用于循环计数的一种遍历工具这个类类似于指针类型，也提供了间接访问的方式；

迭代器的使用

```
auto b= v.begin(),e=v.end();//b和e类型相同
```

尾后迭代器:end成员返回的迭代器称为尾后迭代器；

迭代器运算符：

标准容器迭代器的运算符

```
*iter             //返回迭代器iter所指的元素的引用
iter->men         //解引用iter并获取该元素的名men的成员，等价于(*iter).men
++tier            //令iter指示容器中的下一个元素
--iter            //令iter指示容器中的上一个元素
iter1 == iter2    //
iter1 != iter2    //
```

使用递增运算符(++)来将一个元素移动到下一个元素

> ## 泛型编程
>
> 原来使用C或者Java的程序员转而使用C++语言之后，会对for循环中使用!=而非<进行判断有点奇怪。之前讲过，大多数标准库类型只定义了==和!=的i情况，但是没有定义<运算符，所以要养成使用!=的习惯。

迭代器类型

```
vector <int>::iterator it;       //it能读写vector <int>的元素
string::iterator it2;            //  it2能够读写string中的内容
```

> ## 术语：迭代器和迭代器类型
>
> 迭代器可能表示三种意思：迭代器本身，容器定义的迭代器类型和得带起的对象
>
> 着重理解其中的概念，记得做好区分

箭头运算符(->):箭头运算符主要用于把**解引用**和**成员访问**两个操作并在一起。

以下两种写法等效：

```
(*it).men
it->men
```

有些vector对象的操作会使迭代器失效

已知一个限制是不能在范围for循环中向vector对象添加元素。另外一个限制是任何一种改变vector对象容量的操作都会使得vector对象得迭代器失效。

迭代器运算

迭代器的运算一般有移动跨过多个元素和进行关系运算。

```
iter+n          //迭代器加上一个整数值扔的一个迭代器，迭代器所指示的新位置与原来相比向前移动了n个元素
inter-n         //迭代器向后移动了n个元素
inter1 +=n      //迭代器加法的赋值语句，将iter+n的结果赋值给iter1
iter-=n         //与in=ter+=n互为逆运算
iter1-iter2     //两个迭代器相间的结果是它们的距离，注意，如果是此运算，则这两个迭代器所指向的元素必须是同一容器里的
>,<,>=,<=       //关系运算符，用于比较位置关系
```

二分法搜索：

```
//text必须是有序的
//beg和end表示我们搜索的范围
auto beg =text.begin(),end = text.end();
auto mid = text.begin() +mid.end()/2;//初始状态下的中间点
while (mid != end&&*mid != sought)
{
    if(sought < *mid)//前半部分
        end=mid;
    else
        beg =mid+1;//在mid之后寻找
}
```

数组

数组是一种*复合类型*。

声明方式：`a[d]`其中a是数组的名字，d是数组的维度(个数)

```
unsigned cnt =-42;  //不是常量表达式
constexpr unsigned sz =42       //常量表达式，关于constexpr,详见2.4.4
int arr[10];                    //含有10个整数的数组
int *parr[sz];                  //含有42个整型指针的数组
string bad[cnt];                //错误：cnt不是常量表达式
stringstrs[getchar_size()]      //当get_size是constexpr时正确；否则错误
```

显式初始化数组

可以直接显式初始化数组，此时可以不比生命名数组维度。

字符数组的特殊性：

字符数组有种额外的初始化方式：

```
char a[]={'C','+','+'};
char a1[] = {'C','\0','+'};
char a2[] = "c++";
const char[6] ="Daniel";
```

不允许拷贝和赋值

不能将数组的内容拷贝给其他数组作为初始值，也不能用数组为其他数组赋值

> 一些数组允许数组的赋值，这就是编译器扩展（类似于vscode），但一般来说，别这样做，因为你不能保证别人的也有这个扩展。即该程序非标准型程序。

理解复杂的数组声明

对于指向数组的指针，是从内到外进行指向的；

当然，队修饰符的数量没有特殊限制：

```
int *(&array)[10] =ptrs;
```

array是ptr数组的一个引用

访问数组的元素要用for语句会这下表进行访问，

数组除了大小固定之外，其他用法与vector类似

检查下标的值，这个交给程序员来干，记得就好了

指针和数组

大多数情况下，编译器会自动将数组替换成一个指向数组首元素的指针。

指针也是迭代器的一种！

获取尾后指针就要用到数组的另外一个特殊性质了。我们可以设法获取数组尾元素之后的那个并不存在的元素地址：

```
int *a =&arr[10];        //指向arr尾元素的下一位指针
```

标准函数库**begin和end**

这两个函数与容器中的两个同名成员函数类似，不过数组的不是类类型，因此这两个函数不是成员函数。

正确的使用形式是将数组作为他们的参数

```
int ia[] = {0,1,2,3,4,5};            //ia是一个含有10个整数的数组
int *beg = begin(ia);                //指向ia的首元素的指针
```

> 一个指针如果只想了某种内置函数类型的尾元素的"下一位置"，则其具备与vector的end函数返回的与迭代器类似的功能。特别注意，尾后环境不能执行解引用和递增操作。

指针运算

给(从)一个指针加上(或减去)某整数值，结果仍是指针。新所指向元素与原来的前进了（后退了）该整数值个位置。

```
constexpr size_t sz = 5;
int arr[sz] = {1,2,3,4,5};
int *ip =arr;           //等价于int *ip =&arr[0]
int *ip2 =ip+4;         //ip2指向arr的尾元素arr[4]
```

和迭代器一样，两指针相减得到的是他们之间的距离。参与运算的指针必须指向同一个数组中的元素。

```
auto n = end (arr) - begin(arr);            //n的值是5，也就是arr中元素的数量
```

只要两个指针指向同一个元素的数组，就可以进行关系运算。

解引用和指针运算的交互

如果一个指针指向了一个元素，则允许解引用该指针。

```
int ia[] ={0,2,4,6,8};      //含有5个整数的数组
int last = *(ia+4);         //正确，吧last初始化为8，也就是解引用ia[4]的地址
```

下标不同于指针，但是有和指针相似，不过最常用的还是指针，应为下标仅仅可以操作而数组，而指针可以操作很多元素和类。

> 内置下表运算符所用的索引值不是无符号整型，这一点与vector和string不一样！！！

C风格字符串

字符串字面值一种通用结构示例，这种结构即是从C语言继承来的，按此方法书写的字符串数组中并以空字符串结尾。

C标准库String函数

strlen(p)	返回p的长度，空字符串不计入在内
strcmp(p1,p2)	比教p1和p2的相等性。如果p1==p2则返回0；如果p1>p2，则返回一个正值，否则返回一个负值
strcat(p1,p2)	将p2附加到p1之后，返回p1
strcpy(p1,p2)	将p2拷贝给p1,返回p1

传入此类函数的指针必须指向以空字符串作为结束的数组!!

比较字符串

比较两个字符串时，比较方法和C完全不一样

用到数组的时候其实真正用到的是指向数组首元素的指针。

多维数组

严格来说，C++并没有多维数组，其实这些数组是嵌套数组，即数组里面有数组，谨记这一点，对以后的理解有好处。

声明方式：

```
int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

多维数组的下表使用

可以使用下标来访问数组中的元素，此时数组每个维度对应了一个下标运算符

```
int ia[rowCnt][cloCnt];          //12个未初始化的元素
for (size_t i = 0;i != rowCnt; ++i)
{
    for (size_t j =0;j != colCnt;++j)
    {
        ia[i][j]=i*cloCnt+j;
    }
}
```

> 要使用范围for语句处理多维数组，除了最内层的循环之外，其他所有循环的控制变量都是引用类型。

当程序使用多维数组的名字时，也会自动将其转换为指向数组首元素的指针

由多维数组名转换得来的指针实际上时指向第一个内层数组的指针。

> 在上述的声名中，圆括号必不可少：
>
> ```
> int *ip[4];          //整型指针的数组
> int (*ip)[4];        //指向含有4个整数的数组
> ```

由于到了C++11，所以尽量使用auto和decltype来避免在数组前加上一个指针类型了。

同样，多维数组也允许使用begin和end函数，并且同样能实现功能。

类型别名简化多维数组的指针

```
using int_array =int[4];     //新标准下的类型别名的声名
typedef int int_array[4];    //等价的typedef声名
for (int_array *p =ia; p!=m ia+3;++p)
{
    for (int *q =*p;q! =*p+4;++q)
    {
        cout<<*q<<' ';

    }
    cout<<endl;
}
```

第4章 表达式

基础

多元运算符和多元表达式

作用域一个运算对象的运算符就是一元运算符。

重载运算符：

当运算符作用于类类型的运算对象时，用户可以自定义其含义。因为这种定义事实上是为已有的运算符赋予了另外一层含义，所以称之为**重载运算符。**

l例子：>>,<<,string对象等

左值和右值

C++的表达式要么是左值，要么是右值。

- 赋值运算符需要一个左值作为其左侧的运算对象，得到的结果也仍然是左值
- 取地址符作用于一个左值运算对象，返回一个指向该运算对象的指针，这个指针是一个右值
- 内置解引用符，下标运算符、迭代器解引用运算符，string和vector运算符的结果都是左值
- 内置类型和迭代器的递增递减运算符作用于左值运算对象，其前置版本所得的结果也是左值；

优先级与结合律

复合表达式指的是含有两个或者多个运算符的表达式。

- 括号的优先级大于结合律
- 优先级与结合律的关系

  优先级会影响程序的正确性，因为如果不考虑优先级的话，很容易出现表达式的错误

求值顺序

网上有，这个可以找得到。

> 溢出和其他算数异常
>
> 算术表达式有可能产生未定义的结果.一部分原因时数学性质本身,另外一个部分是源于计算机的特点.

表 4.2：逻辑运算符和关系运算符

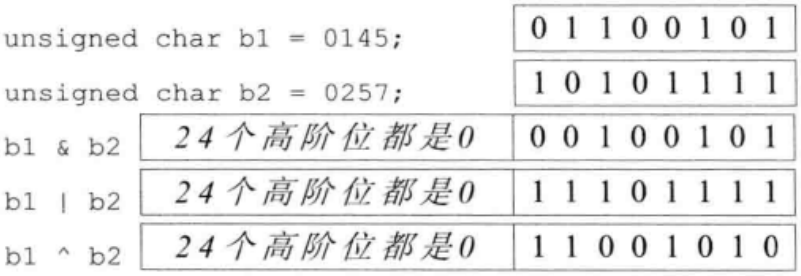
结合律	运算符	功能	用法
右	!	逻辑非	!expr
左	<	小于	expr < expr
左	<=	小于等于	expr <= expr
左	>	大于	expr > expr
左	>=	大于等于	expr >= expr
左	==	相等	expr == expr
左	!=	不相等	expr != expr
左	&&	逻辑与	expr && expr
左	||	逻辑或	expr || expr

表 4.3：位运算符（左结合律）

运算符	功能	用法
~	位求反	~ expr
<<	左移	expr1 << expr2
>>	右移	expr1 >> expr2
&	位与	expr & expr
^	位异或	expr ^ expr
|	位或	expr | expr

左移运算符和右移运算符是用来移动指针的;

与（&）、或（|）、异或（^）运算符在两个运算对象上逐位执行相应的逻辑操作：



移位运算符满足左结合律

即从右到左进行结合,并不是从左到右的

sizeof运算符

sizeof表达式返回一条表达式或者一个类型名字所占的字节数.该函数满足右结合律,所得的值是一个size_t类型

运算符的两种形式:

```
sizeof(type)
sizeof expr
```

第二种形式,sizeof函数返回的时表达式结果的大小,而不是原表达式的大小!!

与众不同的是,sizeof并不实际计算器的运算对象的值.

sizeof运算符的结果部分的依赖于其作用的类型:

- 对char或者类型为char的表达式执行sizeof运算,结果为1
- 对引用类型执行sizeof运算得到被引用对象所占空间大小.
- 对指针执行sizeof运算得到指针本身所占空间大小;
- 对解引用指针执行sizeof运算得到指向的对象所占空间的大小,指针不需要哦有效.
- 对数组执行sizeof函数,则会返回整个数组的大小.等价于对数组中的所有元素各执行一次sizeof运算并将得到的结果求和.注意!sizeof函数并不会将数组看作指针来处理
- 对string对象活vector对象执行sizeof运算只返回类型固定部分大小，不会计算对象中的元素占用了多少空间

因为sizeof的返回值是一个常量表达式，所以我们可以用sizeof的声名数组的维度。

逗号运算符

含有两个运算对象，按照从左到右的顺序依次求职。

类型转换

各个数据类型是可以进行相互转换的，有时候不需要程序员介入，称之为**隐式转换。**

发生隐式转换的条件：

- 在大多数表达式中，比int类型小的整型值首先提升为交大的整数类型
- 在条件中，非布尔值转换成布尔值类型
- 初始化过程中，初始值转换成变量类型：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种类型，需要转换成同一种类型
- 函数调用时也会自动发生转换

算术转换：把算术类型转换成另一种算术类型。

整型提升：负责把小整数类型提升成较大的。

其他隐式类型的转换

指针的转换：

```
int ia[10];      //含有10个整数的数组
int *ip =ia;     //ia转换成只想数组首元素的指针
```

转换成布尔类型：

```
chat *cp =get_string();
if (cp) /*...*/      //如果*cp不是空字符
while(*cp)/*...*/    //如果cp不是空字符，条件为真
```

转换成常量：

```
int i;
const int&j =i;          //非常量转换成const int de
const int *p = &i;       //非常量的地址可以转换成const的地址
int &r =j,*q = p;        //错误，不允许const转换换成非常量
```

命名的强制类型转换：

```
cast-name <type>(expression)
```

cast-name有四种：static_cast、dynamic_cast、const_cast和reinterpet_cast

- static_cast

  任何具有明确定义的类型转换，只要不包含底层const，都可以使用static_cast;

- const_cast

  const_cast只能改变运算对象的底层const。

```
const char *pc;
char *p = const_cast <char*>(pc);          //正确：但是通过p写值是未定义的行为
```

只有const_cast 能改变表达式中的常量属性，使用其他形式的命名强制类行转换改变表达式的常量属性都将引发编译器错误。同样的，也不能用const_cast改变表达式的类型：

```
const char *cp;          //错误：static不能转换掉const性质
char *q =static_cast <char*>(cp);       //正确：字面值转换成string类型e

```

- reinterpret_cast

reinterpret_cast通常为

reinterpret_cast运算对象的位模式提供较低层次上的重新解释。

```
int *p;
char *pc = reinterpret_cast<char*>(ip);
```

> 提示：避免强制类型转换

> 迁至类型转换或干扰正常的类型检查，因此物品们强烈建议必买内使用这类方法，不然会造成意想不到的后果！！就算实在是无法避免，也应该尽量限制其转换值得作用域，并且记录相关类型得所有假定。

表 4.4：运算符优先级

结合律	和运算符	功能	用法	参考页码
左	::	全局作用域	::name	256
左	::	类作用域	class::name	79
左	::	命名空间作用域	namespace::name	74
左	.	成员选择	object.member	20
左	->	成员选择	pointer->member	98
左	[]	下标	expr[expr]	104
左	()	函数调用	name(expr_list)	20
左	()	类型构造	type(expr_list)	145
右	++	后置递增运算	lvalue++	131
右	--	后置递减运算	lvalue--	131
右	typeid	类型 ID	typeid(type)	731
右	typeid	运行时类型 ID	typeid(expr)	731
右	explicit cast	类型转换	*cast_name*<type>(expr)	144
右	++	前置递增运算	++lvalue	131
右	--	前置递减运算	--lvalue	131
右	~	位求反	~expr	136
右	!	逻辑非	!expr	126
右	-	一元负号	-expr	124
右	+	一元正号	+expr	124
右	*	解引用	*expr	48
右	&	取地址	&lvalue	47
右	()	类型转换	(type) expr	145
右	sizeof	对象的大小	sizeof expr	139

结合律	和运算符	功能	用法	参考页码
右	sizeof	类型的大小	sizeof(type)	139
右	Sizeof...	参数包的大小	sizeof...(name)	619
右	new	创建对象	new type	407
右	new[]	创建数组	new type[size]	407
右	delete	释放对象	delete expr	409
右	delete[]	释放数组	delete[] expr	409
右	noexcept	能否抛出异常	noexcept (expr)	690
左	->*	指向成员选择的指针	ptr->*ptr_to_member	740
左	.*	指向成员选择的指针	obj.*ptr_to_member	740
左	*	乘法	expr * expr	124
左	/	除法	expr / expr	124
左	%	取模（取余）	expr % expr	124
左	+	加法	expr + expr	124
左	-	减法	expr - expr	124
左	<<	向左移位	expr << expr	136
左	>>	向右移位	expr >> expr	136
左	<	小于	expr < expr	126
左	<=	小于等于	expr <= expr	126
左	>	大于	expr > expr	126
左	>=	大于等于	expr >= expr	126
左	==	相等	expr == expr	126
左	!=	不相等	expr != expr	126
左	&	位与	expr & expr	136
左	^	位异或	expr ^ expr	136
左	|	位或	expr | expr	136
左	&&	逻辑与	expr && expr	126
左	||	逻辑或	expr || expr	126
右	? :	条件	expr ? expr : expr	134
右	=	赋值	lvalue = expr	129
右	*=, /=, %=	复合赋值	lvalue += expr 等	129
右	+=, -=			129
右	<<=, >>=			129
右	&=, |=, ^=			129
右	throw	抛出异常	throw expr	173
左	,	逗号	expr, expr	140