C++ Primer笔记

第一章 开始

1. 读取数量不定的输入数据

```
void test01() {
    int sum = 0, value = 0;
    while (cin >> value)
        sum += value;
    cout << "Sum is:" << sum << endl;
}
```

2. 错误输出

```
cerr << "No data?!" << endl;
clog << "No data?!" << endl;
```

第二章 变量和基本类型

1. `\a`在输出时可以发出报警声
2. `\r`和`\n`分别是回车符和换行符，在windows中使用`\r`使光标回到本行的初始位置，使用`\r\n`进行换行操作。
3. `-42`这个值中的`42`部分是字面值，而`-`是对字面值进行取反操作
4. 字符字面值`'a'`表示的是单独的字符，而字符串字面值`"A"`则代表了一个字符的数组，该数组包含两个字符：一个是字母`A`，另一个则是空字符`\0`用于表示字符串的结束
5. 初始化和赋值虽然都可以使用等号，但是其有很大的区别，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当前值擦除，而以一个新值来替代。

   未初始化的变量含有一个不确定的值，使用未初始化的变量是一种错误的行为
6. 如果定义变量时没有指定初值，则变量会被**默认初始化**。

   对于内置类型，定义于任何函数体之外的变量会被初始化为0；而定义在函数体内部的内置类型将**不被初始化**。（建议初始化每一个内置类型的变量）

   其中C++的内置类型包括算术类型和空类型（void）

   其中算术类型又分为两类：整型（包括`bool`&`char`）和浮点型
7. 如果想声明一个变量而非定义它，可以在变量名前添加`extern`关键字

```
extern int j; // 声明j而非定义j
```

8. 变量能且只能被定义一次，但是可以多次被声明。

   如果要在多个文件中使用同一个变量，就必须将声明和定义分离。此时，变量的定义必须出现且只能出现在一个文件（cpp文件）中，而其他用到该变量的文件必须对其进行声明（包含头文件），却绝对不能重复定义。

9. 声明空指针的三种方法：

```
int *p1 = nullptr;  // 新标准推荐使用

int *p2 = 0;

int *p3 = NULL;

int zero = 0; int *pi = zero; // 这是一种错误方式
```

   建议初始化所有指针，在对象未定义时将指针初始化为`nullptr`。

10. 指针的四种状态：
    1. 指向一个对象
    2. 指向紧邻对象所占空间的下一个位置
    3. 空指针
    4. 无效指针

11. 默认情况下，const对象**仅在文件内有效**（因为C++是分文件编译的，对于const类型，编译器在编译阶段直接使用其对应的常量值对这个const常量进行文本替换）。当多个文件中出现了同名的const变量时，其实等同于在不同的文件中分别定义了独立的变量。在一般情况下，如果const变量的初始值为一个常量表达式，我们可以在每个文件中都定义一个独立const变量即可。但如果某个文件定义了一个const变量，初始值不是一个常量表达式，且该const变量需要被其他文件共享时，需要在声明和定义中都添加关键字`extern`。例如：

```
// file_1.cc定义并初始化一个常量

extern const int bufSize = fcn();

// file_1.h头文件使用file_1.cc中的常量

extern const int bufSize;
```

12. **对const的引用**可能引用到一个非const对象。

```
int i = 42;

const int &r1 = i;     // 正确：允许将const int&绑定到一个普通的int对象上

const int &r2 = 42; //正确：常量引用

const int &r3 = r1 * 2;// 正确：常量引用

int &r4 = r1 * 2;    // 错误：r4是一个普通的非常量引用
```

正常情况下，引用的类型必须与其所引用的对象类型一致，例如：

```
const int ci = 1024;

const int &r1 = ci;
```

在下面这种情况下

```
double dval = 3.14;

const int &ri = deval;
```

其实编译器把上述代码变成了如下形式：

```
const int temp = dval;

const int &ri = temp;
```

注意这里的temp是编译器创建的临时量对象，并使用ri进行了绑定。ri并没有绑定到我们最初想让它引用的dval上，这和行为目的不符合，所以这种操作在C++中是非法的。而对于这种情况：

```
int i = 42;

const int &ri = i;
```

是一种合法的行为。只不过const限制了我们无法通过ri去修改i的值，但此时i的值还是可以通过别的方式修改，比如通过i自身修改。

13. 对于常量对象，我们只能使用**指向常量的指针**来存放它的地址。

```
const double pi = 3.14;

const double *cptr = &pi;
```

同引用一样，我们可以使用指向常量的指针来指向一个非常量的对象。但是我们无法通过指向常量的指针来修改这个非常量对象的值。

```
double dval = 3.14;

cptr = &dval;
```

其实所谓指向常量的指针和引用，不过是指针或者引用“自以为是”罢了，它们觉得自己指向了常量，所以自觉地不能通过自身去改变所指向对象的值。

指向常量的指针要和**const指针**区分开来，例如下面的curErr指针本身是一个不能修改的常量，但是其指向的对象errNum是一个可以修改的对象，我们可以通过`*curERR`来修改它的值。

```
int errNum = 0;

int *const curErr = &errNum;
```

14. 顶层const：指针本身是一个常量

    底层const：指针指向的对象是一个常量

```
int i = 0;

int *const p1 = &i; // 顶层const

const int ci = 42;    // 顶层const

const int *p2 = &ci;  //底层const

const &r = ci;       // 声明引用的const都是底层const
```

执行拷贝时，要求拷入和拷出的对象拥有一致的底层const限制（有或没有）或者底层cosnt限制可以自动转换，一般只有非常量可以自动转换为常量。

15. **常量表达式**是指值不会改变且在编译过程中就能得到计算结果的表达式。

```
const int max_files = 20;

const int limit = max_files + 1;
```

都是常量表达式。

```
int staff_size = 27;

const int sz = get_size();
```

都不是常量表达式。

在复杂系统中，我们可以将我们认定的常量表达式设置为一个**constexpr变量**。

```
constexpr int mf = 20;

constexpr int sz = size();  //只有当size是一个constexpr函数时，才是正确的声明语句
```

此外，函数体内定义的变量一般来说并非存放在固定地址当中，因此constexpr指针不能指向这样的变量。相反，定义于所有函数体之外的对象其地址固定不变，嗯那个用来初始化constexpr指针。

需要明确的是，如果在constexpr声明中定义了一个指针，constexpr仅仅对指针有效，和指针所指的对象无关。

```
const int *p = nullptr;   // 指向常量对象的指针

constexpr int *q = nullptr; // 指向对象的常量指针
```

16. 定义**类型别名**的传统方法是使用关键字typedef

    typedef double wages; // wages是double的同义词

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

    新标准规定了一种新的方法，使用**别名声明**来定义类型别名

    using SI = Sales_item;

    如果某个类型别名指代的是复合类型或者常量，例如

```
typedef char *pstring;

const pstring cstr = 0; // 注意cstr是指向char的常量指针

const pstring *ps;  // ps是一个指针，它指向的对象是指向char的常量指针
```

    虽然这里`pstring`是`char\*`的同义词，但是不可以直接替换为`char\*`，这将会导致

    `const char *cstr = 0;`

    也就是把cstr变成了指向`const char`的指针，是一种错误理解。

17. 在编程时常常需要把表达式的值赋给变量，这要求在声明变量时清楚的知道表达式的类型。C++11新标准引入了`auto`类型说明符，可以让编译器替我们去分析表达式所属的类型。显然，`auto`定义的变量必须有初始值。

    `auto i = 0, *p = &i;`是一条正确的语句。

    编译器有时推断出的类型和初始值的类型不完全一样，编译器会适当地改变结果类型使其更符合初始化规则。

    1. 使用引用其实是用引用的对象，特别是当引用被作为初始化值的时候，真正参与初始化的其实是引用对象的值。例如下面代码中的`auto`就是一个`int`类型。

       `int i = 0, &r = i;`

       `auto a= r;`

    2. auto一般会忽略顶层const，同时底层const被保留。

```
cosnt int ci = i, &cr = ci;

auto b = ci;  // b是一个int型变量，顶层cosnt特性被忽略。

auto c = cr;  // c是一个int型变量，cr是ci的别名，ci本身是顶层const，也被忽略

auto d = &i;  // auto是一个int型指针

auto e = &ci; // auto是一个指向int型常量的指针，对常量对象取地址是底层const
```

如果希望推出的auto类型是一个顶层const，则需要手动指定`const auto f = ci;`

18. 如果我们希望从表达式的类型推断出要定义的变量的类型，但是不想用该表达式的值初始化变量。C++11新标准中引入了第二种类型说明符`decltype`，它的作用是选择并返回操作数的数据类型。此过程中，编译器仅仅分析表达式并得到类型，而并不计算实际表达式的值。

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

    在上面的代码中，编译器并不实际调用函数f，而是使用当调用发生时f的返回值类型作为sum的类型。

    decltype处理顶层`const`和引用的方式与`auto`不同（返回类型包括顶层const和引用在内）。

    ```
    const int ci = 0, &cj = ci;

    decltype(ci) x = 0; // x的类型是const int

    decltype(cj) y = x; // y的类型是const int&

    decltype(cj) z; // 错误使用，z是一个引用，必须初始化。
    ```

    需要指出的是，引用从来都是作为其所指对象的同义词出现，只有在decltype中是一个例外。

    decltype和引用的关系：

    1. 如果`r`是一个`int`引用类型，则`decltype(r)`将得到引用类型。如果想让`r`是所指的类型，可以把r作为表达式的一部分，如`r+0`这个具体值，则`decltype(r + 0)`将得到int类型。
    2. 另一方面，对于`int i =42, *p = &i;`如果表达式的内容是解引用操作，将得到引用类型，即`decltype(*p)`将得到`int&`。
    3. decltype的表达式如果是加上了括号的变量，结果`decltype((variable))`将永远是引用。

    总而言之，对于变量，decltype将返回该变量的类型（包括顶层const和引用在内）；对于表达式，如果表达式的运算结果是一个可以作为赋值操作语句的左值，那么将返回引用类型（2和3就是很好的说明）。

19. 为了避免定义重复的类，我们将类定义在头文件中。为了避免重复引入头文件，我们使用**头文件保护符**（与处理功能的一种），头文件保护符依赖于预处理变量。

```
#ifndef SALES_DATA_H

#define SALES_DATA_H

// 头文件内容

#endif
```

预处理变量将无视C++中关于作用域的规则。

第三章 字符串、向量和数组

1. 命名空间的using声明，形式应当为`using namespace::name`;例如`using std::cin;`其中，头文件不应包含using声明，因为如果某个头文件有using声明，头文件的内容会被拷贝到所有引用它的文件中，可能会有名字冲突。
2. 使用cin接收字符串string对象时，会自动忽略开头的空白（空格符、换行符、制表符等）并从第一个真正的字符开始读起，直到遇见下一个空白为止。如果输入`"  Hello  world  "`，string接收到的值为`"Hello"`。

   同样也可以输入未知数量的string对象，使用`while(cin >> s){}`

   `getline()`函数可以保留输入的空白符，直到遇到换行符为止，可以读取一整行字符`getline(cin, s);`，但要注意string对象s中并不包含换行符。
3. string::size_type类型是s.size()的返回类型，具有与机器无关的特性。它是一个无符号类型，所以最好不要将其与带符号数混用。
4. C++语言中的字符串字面值并不是标准库类型string的对象，是不同的类型。
5. C++11新标准提供了范围for语句，可以遍历序列中的每一个元素。`for(auto c : str){}`

   如果想使用此for语句去改变序列中的元素，则需要把循环变量定义成引用类型。`for(auto &c : str){}`
6. &&运算符很重要的一点是，只有左侧运算对象为真时才会去检查右侧对象。
7. 在vector初始化时，`vcetor<T> v5{a,b,c,...}`和`vcetor<T> v5 = {a,b,c,...}`是等价的。
8. 在vector初始化时，可以只给定元素数量，如`vector<int> ivec(10);`。但是要求，元素类型支持默认初始化。
9. 在vector初始化时，一般来说{}花括号是列表初始化，()圆括号是构造vector对象。但是当vector使用{}花括号进行初始化时，给定的参数又不能用于列表初始化时，会考虑使用{}来构造vector对象

```
vector<string> v5("hi!"); //构造
vector<string> v5(10);  //构造
vector<string> v5(10, "hi!"); //构造
vector<string> v5{"hi!"}; //列表初始化
vector<string> v5{10};  //构造
vector<string> v5{10, "hi!"}; //构造
```

10. `unsigned`可以单独作为数据类型，它默认代表了`unsigned int`

11. 有迭代器的类型有名为begin和end的成员，其中`begin()`返回指向第一个元素的迭代器，而`end()`返回指向容器（或者string）尾元素的下一个位置的迭代器，即尾后迭代器。特别的，当容器为空时，begin和end返回的是同一个迭代器，都是尾后迭代器。所有类型的迭代器iter都可以执行`++iter`和`--iter`的操作，用于令iter指向容器中的下一个或者上一个元素。
12. 迭代器类型有`iterator`和`const_interator`，其中itertor指向的对象可读可写。而const_iterator指向的对象只能读取而不能修改它指向的元素，类似于指向常量指针。如果vertor对象或者string对象是常量，只能用后者，如果不是常量，则都可以使用。begin和end默认返回对应数据类型的迭代器（即对常量返回const_iterator，对非常量返回iterator）。C++11新标准引入了`cbegin()`和`cend()`来强制返回一个const_interator迭代器。
13. 不能在范围for循环或者使用迭代器的循环中向vector中添加元素；可能改变vector对象容量的操作都有可能会使迭代器失效，如push_back
14. vector和string还支持一些额外的迭代器操作

支持的操作	解释
iter + n	返回迭代器类型
iter - n	返回迭代器类型
iter += n	
iter -= n	
iter1 - iter2	必须指向同一个容器的某个位置，且返回类型为difference_type，是一种带符号型整数
>、>=、<、<=	必须指向同一个容器的某个位置，返回bool类型，来判断两个迭代器孰先孰后

15. 在定义数组a[d]时，必须保证d是一个constexpr常量表达式，而且不能使用auto自动推导数组中元素类型。默认情况下，数组元素默认初始化，但是如果在函数内部定义了某种内置类型数组，默认初始化会含有未定义的值。
16. 字符串字面值后面隐含了一个空字符`'\0'`，所以在使用字符串给数组初始化时，要注意设置数组的容量，例如`char a[6] = "Dannel";`就是错误的定义方式。其次，如果使用`char a[] = {'C', '+', '+'};`定义字符数组，将会得到一个没有以'\0'结尾的c风格字符，正确的做法应该是`char a[] = {'C', '+', '+', '\0'};`。
17. 数组不允许使用数组进行初始化或者赋值操作，因为编译器会把数组转换成一个指针。比如，使用auto自动推断类型时`auto b = a;`得到的将是指针类型；但使用decltype(a)将得到对应容量的数组类型，例如`decltype(a) b = {...}`。
18. 指向数组元素的指针基本上和vector或者string中的迭代器用法一致，为了获取数组的尾后指针，我们可以使用下标方式，如`int *end = &arr[size]`;其中size是arr的长度。

    但是这种方式容易出错，造成指针溢出，所以C++11新标准提供了begin和end方法，这两个方法来自iterator头文件，分别返回指向数组第一个元素和最后一个元素后一个位置的指针。

    指向数组元素的指针支持的运算同vector和string的迭代器基本一致（见14中的表格），不过指针相减时返回的是`pstrdiff_t`类型，同`difference_type`，属于带符号类型。

19. 指向数组元素的指针或者数组尾后指针都支持使用下标运算符[]访问数组元素，这是C++定义的。而string和vector的下标符号[]是他们自己的库文件定义的。其区别在于前者可以使用负值作为索引，而后者只能使用无符号类型进行索引。
20. c风格字符串提供了一系列函数。如`strlen`、`strcmp`、`strcat`、`strcopy`，但是当字符数组初始化为`char c[] = {'c', '+', '+'}`时，并没有使用`'\0'`结尾，上面的函数都会出现错误。所以我们尽可能的使用标准库string。
21. 在C和C++的衔接中，难免会混用string对象和C风格字符串，string对象在定义时为C风格字符串留下了一些接口。
    1. 允许使用以'\0'结尾的字符数组初始化string对象或者给string对象赋值，反之不行。

    ```
    char str[4] = {'C', '+', '+', '\0'};
    string str1 = str;
    string str2(str);
    ```

    2. string的加法允许其中一个运算元素为以'\0'结尾的字符数组。
    3. string提供了转换为C风格字符串的函数`c_str()`，它返回一个`const char*`。
    4. 在新老风格代码衔接中，vector可以使用数组进行初始化，如`vector<int> v(begin(arr), end(arr));`。
22. 因为编译器会把数组转换成一个指针，所以在使用范围for循环时需要将除了最内存以外的所有循环元素设置为引用。如

    ```
    int matrix[3][3] = {0};
    for(auto &row : matrix){
        for(auto num : row){
        //
        }
    }
    ```

23. c++的表达式要么是**左值**，要么是**右值**。对于对象来说，当一个对象被用作左值时，用的是对象的身份；反之被用作右值时，用的是对象的值。

    使用decltype求表达式（注意是表达式，不是对象）时，如果表达式求值结果是左值，那么就返回引用类型。
24. 指定了**求值顺序**的操作符有`&&`、`||`、`?:`、`,`。例如`&&`只有在确定左侧为真的情况下才会计算右侧的值，这种特性叫做**短路求值**。

    对于未指定求值顺序的操作符，如果参与运算的对象影响了同一对象，则将是未定义的表达式。如`cout << i << " " << i+1 <<endl;`。
25. 对于`/`除法运算，c++11新标准规定商一律向0取整，即无脑删除小数部分。而对于`%`取余运算`m%n`，结果的正负性同`m`永远一致。
26. 位运算符会自动进行整型提升（小整型->大整型，基本上是char、short->int）。
27. 左移运算符是在整数末尾补0，右移运算符则是在首位补上符号位或者0（视具体环境而定）。
28. sizeof的用法有`sizeof(type)`和`sizeof expr`，其中后者返回的是表达式结果类型的大小，但注意sizeof并不计算表达式的结果而仅仅是返回其类型。sizeof不会将数组转化为指针，所以返回的是整个数组所占的空间。如果sizeof作用在常量表达式上则可以用于声明数组的长度。

29. 隐式类型转换
    1. 整型提升：short和char会转换成int，无符号小整型则转换成unsigned int
    2. 数组转换为指针：除了decltype、&取地址、sizeof、typeid、使用引用初始化数组这几种情况
30. 显式类型转换

    旧式强制类型转换语法为type(expr)或(type)expr，这种强转在遇到错误时很难追踪其错，所以引入了新的强制类型转换`cast-name<type>(expression)`

    1. static-cast：只要不包含底层const都可以使用。一般用于将大的数据类型强转为小的数据类型而不会报错。还可以用于找回存于void*的指针。
    2. const-cast：只能改变底层const，将const转为非const
    3. reinterpret-cast：用小的数据类型指向大的数据类型，一般不要用。