iostream库包含两个基础类型istream(输入流)和ostream(输出流).

表达式由一个或多个运算对象和一个运算符组成.

std::cin 标准输入

std::cout 标准输出

std::cerr 标准错误

std::clog 输出一般信息, 关联到标准错误

流: 一个字符序列, 随着使时间推移, 字符是按顺序生成或消耗的.

输出运算符<<

<<运算符接受两个运算对象: 左侧必须是一个ostream对象, 右侧的运算对象是要打印的值.

<<运算符将给定值写到指定的ostream对象中, 计算结果就是其左侧运算对象.

endl: 是一个被称为操纵符的特殊值, 其效果是结束当前行, 并将与设备关联的缓冲区中的内容刷到设备中.

标准库定义的所有名字都在命名空间std中.

作用域运运算符(::): 指出想使用某个命名空间中的名字, 如std::cout.

输入运算符>>

>>运算符接受一个istream作为其左侧的运算对象, 接受一个对象作为其右侧运算对象.

>>运算符从给定的istream中读入数据, 并存入给定的对象中, 返回左侧运算对象作为结果.

使用调用运算符()来调用一个函数.

读取数量不确定的输入

while(std::cin >> value)

当使用一个istream对象作为条件时, 其效果时检测流的状态.

如果流是有效, 那么检测成功; 当遇到文件结束符(EOF)或遇到一个无效的输入是, istream对象的状态会变为无效. 处于无效状态的istream对象条件变为假.

# 变量和基本类型

## 基本内置类型

C++的基本数据类型分为算术类型和空类型.

算术类型包括字符, 整型数, 布尔值和浮点数, 也就是整型和浮点型.

C++规定一个int至少和一个short一样大, 一个long至少和一个int一样大, 一个long long至少和一个long一样大.

long long是C++11新定义的类型.

类型char在一些机器上是有符号的, 在一些机器上是无符号的.

不要在算术表达式中使用char或bool.

执行浮点运算选用double.

如果把一个布尔值用在算术表达式里, 则它的取值非0即1.

有符号数和无符号数在同一条表达式中, 有符号数会转变为无符号数

字面值常量的形式和值决定了它的数据类型

十进制字面值是带符号数.

八进制和十六进制字面值可能是带符号数也可能是无符号数.

十进制字面值是int, long, long long中尺寸最小的那个.

八进制和十六进制字面值类型是

int, unsigned int, long, unsigned long, long long, unsigned long long

中尺寸最小的那个.

类型short没有对应的字面值.

尽管整型字面值可以存储在带符号的数据类型中, 但严格来说, 十进制字面值不会是负数. 例如-42, 那个负号并不在字面值之内, 它的作用仅仅是对字面值取负值而已.

转义序列

转义序列均以反斜线作为开始, 反斜线(\)后跟特殊字符, 如单引号, 双引号, 问号, 反斜线.

**泛化的转义序列**: \x后跟1个或多个十六进制数字, 或\后跟1~3个八进制数字.

\7(响铃) \12(换行符) \40(空格)

\0(空字符) \115(字符M) \x4d(字符M)

可以通过前缀和后缀改变整型, 浮点型和字符型字面值的默认类型.

|  |  |  |
| --- | --- | --- |
| **字符和字符串的字面值** | | |
| **前缀** | **含义** | **类型** |
| u | Unicode16字符 | char16\_t |
| U | Unicode32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** | |
| **后缀** | **最小匹配类型** |
| u or U | unsigned |
| l or L | long |
| ll or LL | long long |
| **浮点型字面值** | |
| f or F | float |
| l or L | long double |

## 变量

C++11新标准: 可以用花括号来初始化变量. 称为列表初始化.

int uints\_sold = 0;

int uints\_sold = {0};

int uints\_sold{0};

int uints\_sold(0);

以上语句等价.

列表初始化不可以随便用于内置类型的变量初始化, 如果存在丢失信息的风险, 编译器将报错.

long double ld = 3.14159265836

int a{ld}, b = {ld}; // 错误, 存在丢失信息的风险.

int c(ld), d = ld; // 正确, 但会丢失部分值.

分离式编译机制: 允许将程序分割为若干个文件, 每个文件可独立编译.

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.

任何包含了显式初始化的声明即称为定义.

extern关键字用于表明是对变量的声明, 但extern语句如果包含了初始值就不再是声明.

extern int i; // 声明i

int j; // 定义j

extern double pi = 3.14; // 定义pi

在函数体内部试图初始化一个由extern关键字标记的变量会引发错误.

标识符由字母, 数字和下划线组成, 其中必须以下划线开头. 标识符的长度没有限制, 对大小写敏感.

用户自定义的标识符不能连续出现两个下划线, 也不能以下划线紧连大写字母开头.

作用域: C++中大多数作用域都以花括号分隔.

内层作用域如果定义了与外层作用域同名的变量, 内层作用域的变量将覆盖外层作用域的变量.

## 复合类型

C++最基本的两种符合类型: 引用和指针.

一条声明语句由一个**基本的数据类型**和紧随其后的一个**声明符**列表组成.

引用为**对象**起了另外一个名字.

定义引用时必须初始化, 在程序运行过程中不能改变引用绑定的对象.

引用并非对象. 所以不能定义引用的引用.

引用只能绑定到对象上, 而不能与字面值或某个表达式的计算结果绑定在一起;

引用类型都要和与之绑定的对象类型严格匹配.(有两个例外情况)

int &refVal1 = 10; // 错误: 字面值10不是一个对象

double dval = 3.14;

int &refval2 = dval; // 错误: 类型不匹配

指针和引用的不同点:

1. 指针是一个对象, 允许对指针赋值和拷贝, 而且在指针的声明周期内它可以先后指向几个不同的对象.

2. 指针无需在定义时赋初始值.

指针使用解引用符\*来访问对象.

空指针可以使用nullptr或NULL来表示.

nullptr是C++11新标准.

NULL在头文件cstdlib中定义.

不能将int变量直接赋值给指针.

在定义语句中\*和&是类型修饰符.

指向指针的引用

int \*p;

int \*&r = p; // r是一个对指针p的引用

## const限定符

以下语句可以解释清楚const限定符的作用.

const int \* p; // p is a pointer to int const.

int const \* p; // p is a pointer to const int.

int \* const p; // p is a const pointer to int.

const int \* const p; // p is a const pointer to int const.

int const \* const p; // p is a const pointer to const int.

可以使用非const的变量初始化一个const的变量.

注意， 以下写法是正确的

const int i = get\_size();

这表示在运行时初始化.

**默认状态下, const对象仅在文件内有效.**

当以编译时初始化的方式定义一个const对象时, 例如

const int bufSize = 512;

编译器将在编译过程中把用到该变量的地方都替换成对应的值. 也就是说, 编译器会找到代码中所有用到bufSize的地方, 然后用512替换. 为了执行上述替换, 编译器必须知道变量的初始值. 如果程序包含了多个文件, 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变量的文件中都有对它的定义. 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对象被设定为仅在文件内有效. 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.

某些时候, const变量的初始值不是一个常量表达式, 例如

const int i = get\_size();

但又确实必要在文件间共享. 解决的办法时, 对于const变量不管是定义还是声明都要加上extern关键字, 这样只需要定义一次就可以了.

// 在xxx.cpp文件中定义

extern const int bufSize = fcn();

// xxx.h头文件

extern const int bufSize;

引用类型必须与其所引用对象的类型一致, 但是有两个例外.

第一个例外, 在初始化常量时允许用任意表达式作为初始值, 只要该表达式的结果能够转换为引用的类型即可.

int i = 42;

const int &r1 = i; // 正确

const int &r2 = 42; // 正确

const int &r3 = r1 \* 2; // 正确

int &r4 = r1 \* 2; // 错误

以下语句也正确:

double dval = 3.14;

const int &r1 = dval;

事实上编译把上述代码改成了如下形式:

const int temp = dval;

const int &ri = temp;

在这种情况下, ri绑定了一个临时量.

常量表达式是指值不会改变并且在编译过程中就能得到计算结果的表达式.

一个对象(或表达式)是不是常量表达式由它的数据类型和初始值共同决定.

const int max\_files = 20; // max\_files是常量表达式

const int limit = max\_files + 1; // limit是常量表达式

int staff\_size = 27; // staff\_size不是常量表达式

constexpr变量

分辨初始值是否为常量表达式是很难的.

C++11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为常量表达式.

声明为constexpr的变量一定是一个常量, 而且必须使用常量表达式初始化.

字面值类型

可以被声明为constexpr的类型.

算术类型, 引用和指针都属于字面值类型.(?)

自定义类, IO库, string类不属于字面值类型.

在constexpr声明中如果定义了一个指针, 限定符仅对指针有效, 与指针所指对象无关

const int \*p = nullptr; // p是一个指向整型常量的指针

constexpr int \*q = nullptr; // p是一个指向整数的常量指针

constexpr仅对q有效.

## 处理类型

有两种方法可以定义类型别名.

(1) **typedef**方法

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

(2) 使用别名声明

**using** SI = Sales\_item; // SI是Sales\_item的同义词

注意以下例子:

typedef char \*pstring; // pstring是char\*的别名

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

const char \*cstr; // cstr是指向char常量的指针

const pstring \*ps; // ps是指针, 它的对象是指向char的常量指针

auto类型说明符

C++11新标准引入了auto类型说明符, 用它就能让编译器自动分析表达式所属的类型.

auto语句根据变量的初始值推断类型， 所以auto定义的变量必须有初始值.

使用auto也能在一条语句中声明多个变量.

因为一条声明语句只能有一个基本数据类型, 所以该语句中所有变量的初始基本数据类型必须一样:

auto i = 0, \*p = &i; // 正确: i是整数, p是整形指针

auto sz = 0, pi = 3.14; // 错误: sz和pi的类型不一致.

注意变量的**类型**和**基本数据类型**的概念区别.

符号&和\*只从属于某个声明符, 而非基本数据类型的一部分.

顶层const表示指针本身是个常量.

底层const表示指针所指的对象是一个常量.

const int \* const p; // p is a const pointer to int const.

const底层const

const顶层const

auto一般会忽略顶层const, 同时底层const则会保留下来.

decltype类型说明符

auto变量必须要初始化, 如果不想初始化时可使用decltype.

decltype的作用是选择并返回操作数的类型.

decltype(f()) sum = x; // sum的类型就是函数f的返回类型

decltype返回的变量类型包括顶层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&, y绑定到变量x

decltype(cj) z; // 错误: z是一个引用, 必须初始化

引用从来都作为其所指对象的同义词出现, 只用用在decltype处是一个例外.

decltype和引用

int i = 42, \*p = &i, &r = i;

decltype(r + 0) b; // 正确, b是一个int

decltype(\*p) c; // 错误, c是int&, 必须有初始值

如果将引用类型变量作为表达式的一部分, 则decltype得到的是该变量所指向的类型.

如果表达式的内容是解引用操作, 则的decltype的结果是引用类型.

decltype的结果类型与表达式形式密切相关.

如果decltype使用的是一个不加括号的变量, 则得到的结果就是该变量的类型;

如果给变量加上了一层或多层括号, 则decltype得到的是该变量类型的引用.

decltype((var))的结果是引用;

decltype(var)的结果只有当var本身是引用时才是引用.

# 字符串, 向量和数组

## 命名空间的using声明

域操作符::

编译器从操作符左侧名字所指示的作用域中寻早右侧那个名字.

using声明的形式

using namespace::name;

!!头文件不应该包含using声明.

## 标准库类型string

定义和初始化stirng对象

string s1; // s1是空字符串

string s2 = s1; // s2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"cccccccccc"

如果使用等号(=)初始化一个变量, 执行的是拷贝初始化, 否则是直接初始化.

建议使用直接初始化.

读取未知数量的string对象

string word;

while(cin >> word) {}

使用getline()读取一整行

sting line;

while(getline(cin, line)) {}

getline()会获取缓冲区中的换行符, 但是不会把换行符保存到字符串中.

使用索引遍历string对象中的字符最好使用string::size\_type类型.

string的加法运算

必须确保每个加法运算符(+)的两侧的运算对象至少有一个是string.

正确: string s4 = s1 + ", ";

错误: string s5 = "hello" + ", ";

正确: string s6 = s1 + ", " + "world";

错误: string s7 = "hello" + ", " + s2;

## 标准库类型vector

vector的初始化要注意以下方法:

vector<T> v2(v1);

vector<T> v2 = v1; // 以上两种方法等价

vector<T> v5{a, b, c};

vector<T> v5 = {a, b, c}; // 以上两种方法等价

vector<int> ivec(10); // 10个元素, 每个元素的初始值为0

vector<string> svec(10); // 10个元素, 每个都是空string对象

如果使用圆括号, 可以说提供的值是用来指导构造对象的;

如果使用花括号, 可以说提供的值是用来当作初始值的. 这叫列表初始化.

如果循环体内包含有向vector对象添加元素的语句, 则不能使用范围for循环.

vector一些值得注意的操作:

v1 == v2 当且仅当元素数量和对应位置元素的值都相同时返回true

< <= > >= 以字典序进行比较.

## 迭代器介绍

所有标准库容器都支持迭代器, 但只有其中少数几种才同时支持下标运算.

标准容器迭代器的运算不包括> >= < <=, 要比较两个迭代器, 只能使用==或!=.

迭代器的类型有iterator和const\_iterator两种, const\_iterator和常量指针差不多.

vector<int>::iterator it1; // it1能读写vector<int>的元素

vector<int>::const\_iterator it2; // it2只能读vector<int>的元素