学习笔记

函数的组成

> *函数由**返回值**、**函数名**、**参数列表**和**函数体**四部分组成。*

```
// main.cpp
int main(int argc, char *argv[])
{
    return 0;
}
```

指针和引用的主要区别

指针“指向”内存中某个对象，而引用“绑定到”内存中的某个对象，他们都实现了对其他对象的间接访问，二者的区别主要有两方面：

第一，指针本身就是一个**对象**，允许对指针**赋值**和**拷贝**，而且在指针的生命周期内她可以指向几个不同的对象；引用**不是**一个对象，无法令引用重新绑定到另外一个对象。

第二，指针**无须**在定义时赋初值，和其他内置类型一样，在块作用域定义内的指针如果没有被初始化，也将拥有一个不确定的值；引用则**必须**在定义时赋初值。

C++列表初始化

当用于内置类型的变量时，这种初始化形式有一个重要的特点：如果我们使用列表初始化且初始值存在**丢失信息**风险，则编译器将报错。

> *我的编译器(g++ (Ubuntu 4.8.5-4ubuntu8) 4.8.5)并没有报错，只是发出了警告；可以，使信息丢失是一个不好的习惯。*

名字的作用域

```
#include <iostream>
#include <string>

using namespace std;

int dog;

int main(int, char **)
```

```
{
    dog = 1234;
    string dog = "dog";
    {
        double dog = 1.234;
        cout << dog << endl;
        cout << ::dog << endl;
    }
    return 0;
}
```

运行结果

```
1.234
1234
```

指针的赋值

除了`const int *p = &i;`和`父类指针指向子类对象`之外,其他所有指针的类型都要和它所指向的对象**严格匹配**.

还有一种指针可以匹配任何类型--`void *`

指针的值

- 指向一个对象.
- 指向紧邻对对象所占空间的下一位置(++p)
- 空指针,意味着指针没有指向任何对象.
- 无效指针(野指针).

> *试图拷贝或以其他方式访问无效指针的值都将引发错误.编译器并不负责检查此类错误.这一点和试图始终未经初始化的变量是一样的.访问无效指针的后果无法预计,因此程序员必须清楚任意给定的指针是否有效.*

理解复合类型的声明

变量的定义包括一个基本数据类型(base type)和一组声明符.

对于复合类型到底是什么,最简单的办法是**从右向左**阅读,离变量名*最近*的符号对变量的类型有最直接的影响.

指向指针的引用

```
    // 引用绑定在指针上
    int i = 42;
    int *p = &i;
    int *&r = p;

    r = &i;
    *r = 0;
```

默认状态下,const对象仅在文件内有效

如果需要const常量在文件间共享:对于const变量不管是声明还是定义都添加extern关键字.

```
// test.h
#ifndef TEST_H
#define TEST_H

extern const int bufsize;

void fun();

int fun2();

#endif // TEST_H
```

```
// test.cpp
#include "test.h"
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

extern const int bufsize = fun2();

void fun()
{
    cout << "&bufsize = " << &bufsize << endl;
}

int fun2()
{
    srand(time(nullptr));
    return random() % 100;
}
```

```
// main.cpp
#include "test.h"
#include <iostream>

using namespace std;

int main(int, char **)
{
    fun();
    cout << bufsize << endl;
    cout << &bufsize << endl;
    return 0;
}
```

指针和const

> ***指向常量的指针***(pointer to const) *不能用于改变其所指对象的值.要想存放常量对象的地址,只能使用指向常量的指针.*

```
    const double pi = 3.14;
    const double *ptr = &pi;
    *ptr = 42; // error
    double dval = 3.13;
    ptr = &dval;
```

> *指针是对象而引用不是,因此就像其他对象类型一样,允许把指针本身定义为常量.**常量指针***(const pointer)***必须***初始化.*

```
    int errNumb = 0;
    int *const curErr = &errNumb; // curErr将一直指向
errNumb
    const double pi = 3.14159;
    const double *const pip = &pi; // pip是一个指向常量对象的
常量指针
```

> *关于const pointer 和pointer to const的写法和读法,记住从右向左的顺序阅读.*

顶层const

> *指针本身是一个对象,他又可以指向另外一个对象.因此,指针本身是不是常量以及指针所指的是不是一个常量就是两个相互独立的问题.*

- **顶层const**(top-level const) 表示指针本身是个常量.
- **底层const**(low-level const) 表示指针所指的对象是一个常量.
- **注:引用不是对象所以与常量绑定的引用都是底层const.**

```
int i = 0;
int *const p1 = &i; // top
const int ci = 42; // top
const int *p2 = &ci; // low
const int *const p3 = p2; // low top
const int &r = ci; // low 第三种情况
```

当执行对象的拷贝操作时,常量是顶层const还是底层const区别明显.其中,顶层const不受什么影响.

```
i = ci; // 没毛病 不受影响,看到了没?
p2 = p3; // p3 是顶层const
```

底层const的限制不能忽视.当执行对象的拷贝操作时,拷如和拷出的对象必须具有相同的底层const.

```
int *p = p3; // error
p2 = p3;
p2 = &i;
int &r = ci; // error
const int &r2 = i;
```

常量表达式和constexpr

常量表达式(const expression)是指值不会改变并且在编译过程就能得到计算结果的表达式.

constexpr变量

```
constexpr int mf = 20;
constexpr int limit = mf + 1;
constexpr int sz = size(); // error
```

指针和constexpr

如果constexpr声明定义了一个指针,限定符constexpr仅对指针有效,与指针所指的对象无效.

```
const int *p = nullptr;
constexpr int *q = nullptr; // 然而只能指向nullptr,然并卵
```

> *p是一个指向常量的指针,而q是一个常量指针,其中的关键在于constexpr把他所定义的对象置为了顶层const.*

```
    constexpr int *np = nullptr; // top-level const
    int j = 0;
    constexpr int i = 42;
    // 下面这两个都不行,因为他们不是编译极端有值的表达式
    constexpr const int *p = &i; // 我特么...
    constexpr int *p1 = &j; // 常量指针可以指向非常量对象
```

类型别名

传统方法

```
    typedef double wages;
    typedef wages base, *p; // base is double; p is double
pointer.
```

新标准

```
    using SI = Sales_item;
    using Integer = int;
    using Int_Pointer = int *;
```

指针,常量和类别名

```
    typedef char *pstring;
    const pstring cstr = nullptr; // char *const cstr =
nullptr; const修饰的是对象本身,对象本身是个指针类型啊!!!
    const pstring *ps; // char *const* ps;
```

auto类型说明符

auto让编译器通过初始值来推算变量的类型.显然auto定义的变量必须有初始值.(如果没有初始值,那编译器分析个鸡脖呀)

如果用一条的auto语句声明多个变量时,那他们的基本数据类型必须一致.

复合类型,常量和auto

- 当**引用**被当作auto的初始值时,真正参与初始化的其实是引用对象的值.此时编译器以引用的对象的类型作为auto的类型.

```
    int i = 0, &r = i;
    auto a = r; // a is int not int &.
```

- auto一般会忽略掉顶层const,同时底层const则会保留下来,比如初始值是一个指向常量的指针时.

```
    const int ci = i, &cr = ci; // ci is top-level cr is
low-level
    auto b = ci; // int b = ci;
    auto c = cr; // int c = cr; cr just is the ci;
    auto d = &i; // int *d = &i;
    auto e = &ci; // const int *e = &ci;
```

如果希望auto类型是一个顶层const,需要明确指出:

```
    const auto f = ci;
```

还可以将引用的类型设为auto:

```
    auto &g = ci;
    auto &h = 42; // error int &h = 42; you must be joke
me.
    const auto &j = 42;
```

> *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.如果我们给初始值绑定一个引用,此时常量就不是顶层const了.*

```
    const int number = 2345;
    const int &ref_number = number;
    auto &ref = ref_number; // 但是底层const还在,ref是一个
const int &;
```

> > *要在一条语句中定义多个变量时,切记,符号&和*只从属于某个声明符,而非基本数据类型的一部分,因此初始值必须是同一种类型.*

```
    auto k = ci, &l = i; // int
    auto &m = ci, *p = &ci; // const int &m = ci, *p =
&ci;
    auto &n = i, *p2 = &ci; // i is int *p2 is const int;
```

第二个例子:

```
    constexpr int ci = 2 << 9;
    const int &ref = ci, *poi = &ci;
    cout << ref << ", " << *poi << endl;
    cout << &ref << ", " << poi << endl;
```

经典练习题:

```
    const int i = 32;
    auto j = i; // int j = i; 可以忽略的顶层const;
    const auto &k =i; //引用不是对象,引用肯定是low-level;
const int &k =i;
    auto *p = &i; // i 是一个const int; &i 就可以是const int
*p = &i;
    const auto j2 = i, &k2 = i; // 这个很牛逼,很容易看出来是
const int; But,左边是顶层const, 右边是底层const;
    // const int j2 = i;
    // const int &k2 = i;
```

decltype类型指示符

> *有时候会遇到这种情况: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.*

decltype(expr) 编译器分析表达式并得到它的类型,却不实际计算表达式的值.

```
    decltype(fun()) sum = x;
    // 编译器在编译阶段分析函数fun的返回类型,而不实际调用.
```

decltype and const

decltype处理**顶层const和引用**的方式与auto不同,如果decltype使用的表达式是一个**变量**,则decltype返回**该变量的类型**(包括顶层const和引用在内).

```
    const int ci = 0, &cj = ci;
    decltype(ci) x = 0; // const int
    decltype(cj) y = x; // const int &;
    decltype(cj) z; // error
```

mycode

```
#include <iostream>

using namespace std;

int main(int, char **)
{
    const int ci = 0, &cj = ci;
    decltype(ci) x = 1234; // const int x = 1234;
    cout << "x = " << x << endl;

    decltype(cj) y = x; // const int &y = x;
    cout << "y = " << y << endl;
```

```
    cout << "\npointer : \n";
    int number = 1234;
    int *ptr_number = &number;
    decltype(ptr_number) _; // int *_ = &number;
    _ = &number;
    cout << "number = " << *_ << endl;

    int *const cprt_number = &number;
    decltype(cprt_number) __ = &number; // int *const __ =
&number;
    *__ = 2345;
    cout << "number = " << *__ << endl;
    // 对于变量,返回该变量的类型.
    return 0;
}
```

- **NOTE**: 引用从来都作为其所指对象的同义词出现,只有在decltype处是一个例外.

decltype和引用

一般来说表达式结果对象能作为一条赋值语句的左值时,decltype将返回一个引用类型.

```
    int i = 42, *p = &i, &r = i;
    decltype(r + 0) b; // int b; r + 0 is not l-value
    decltype(*p) c; // error c is int &; *p is a l-value;
```

- **暂时这么记**:如果一个表达式得到的对象我们可以获取她的地址,那么她是一个左值.

decltype和auto的另一处重要区别是,decltype的结果类型与表达式形式密切相关.有一种情况需要特别注意:对于decltype所用的表达式,如果变量名加上了一对**括号**,则得到的类型与不加括号时会有不同.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,编译器就会把她当成是一个表达式,即decltype就会得到**引用类型**.

```
    decltype((i)) d; // error d is a reference,without
init.
    decltype(i) e;
```

- **最后亿个**:赋值会产生引用的一类典型表达式，引用的类型就是左值的类型．也就是说，如果i是int，则表达式`i = x`的类型是一个`int &`.

  decltype(i = x) num = y; // int &num = y;

写个头文件的例子吧

```
// student.h
#ifndef STUDENT_H
#define STUDENT_H

struct student
{
    char name[40];
    int age;
};

// ....

#endif // STUDENT_H
/* EOF */
```