# 1. 什么是C+11

C++11标准为C++编程语言的第三个官方标准，正式名叫ISO/IEC 14882:2011 - Information technology -- Programming languages -- C++。在正式标准发布前，原名C++0x。它将取代C++标准第二版ISO/IEC 14882:2003 - Programming languages -- C++成为C++语言新标准。

C++11是对目前C++语言的扩展和修正， C++11不仅包含核心语言的新机能，而且扩展了C++的标准程序库（STL） ，并入了大部分的C++ Technical Report 1（TR1） 程序库(数学的特殊函数除外)。

C++11包括大量的新特性：包括lambda表达式，类型推导关键字auto、 decltype，和模板的大量改进。

# 2. 类型推导

## 2.1 auto

auto的自动类型推导，用于从初始化表达式中推断出变量的数据类型。从这个意义上讲，auto并非一种“类型”声明，而是一个类型声明时的“占位符”，编译器在编译时期会将auto替换为变量实际的类型。

通过auto的自动类型推导，可以大大简化我们的编程工作：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

double foo() {}

void func(vector<string> & tmp)

{

for (auto i = tmp.begin(); i < tmp.end(); i++)

{

// 一些代码

}

}

int main()

{

auto x = 1; // x的类型为int

auto y = foo(); // y的类型为double

struct m { int i; }str;

auto str1 = str; // str1的类型是struct m

auto z; // err, 无法推导，无法通过编译

z = x;

return 0;

}

注意点：

void fun(auto x =1) {} // 1: auto函数参数，有些编译器无法通过编译

struct str

{

auto var = 10; // 2: auto非静态成员变量，无法通过编译

};

int main()

{

char x[3];

auto y = x;

auto z[3] = x; // 3: auto数组，无法通过编译

// 4: auto模板参数（实例化时），无法通过编译

vector<auto> x = {1};

return 0;

}

## 2.2 decltype

decltype实际上有点像auto的反函数， auto可以让你声明一个变量，而decltype则可以从一个变量或表达式中得到其类型，如下：

#include <typeinfo>

#include <iostream>

#include <vector>

using namespace std;

int main()

{

int i;

decltype(i) j = 0;

cout << typeid(j).name() << endl; // 打印出"i", g++表示integer

float a;

double b;

decltype(a + b) c;

cout << typeid(c).name() << endl; // 打印出"d", g++表示double

vector<int> vec;

typedef decltype(vec.begin()) vectype; // decltype(vec.begin()) 改名为 vectype

vectype k;

//decltype(vec.begin()) k;

for (k = vec.begin(); k < vec.end(); k++)

{

// 做一些事情

}

enum {Ok, Error, Warning}flag; // 匿名的枚举变量

decltype(flag) tmp = Ok;

return 0;

}

## 2.3 追踪返回类型

返回类型后置：在函数名和参数列表后面指定返回类型。

int func(int, int);

auto func2(int, int) -> int;

template<typename T1, typename T2>

auto sum(const T1 & t1, const T2 & t2) -> decltype(t1 + t2)

{

return t1 + t2;

}

template <typename T1, typename T2>

auto mul(const T1 & t1, const T2 & t2) -> decltype(t1 \* t2)

{

return t1 \* t2;

}

int main()

{

auto a = 3;

auto b = 4L;

auto pi = 3.14;

auto c = mul( sum(a, b), pi );

cout << c << endl; // 21.98

return 0;

}

# 3. 易用性的改进

## 3.1 初始化

### 3.1.1 类内成员初始化

class Mem

{

public:

Mem(int i): m(i){} //初始化列表给m初始化

int m;

};

class Group

{

public:

Group(){}

private:

int data = 1; // 使用"="初始化非静态普通成员，也可以 int data{1};

Mem mem{2}; // 对象成员，创建对象时，可以使用{}来调用构造函数

string name{"mike"};

};

### 3.1.2 列表初始化

C++11引入了一个新的初始化方式，称为初始化列表(List Initialize)，具体的初始化方式如下：

int a[]{1, 3, 5};

int i = {1};

int j{3};

初始化列表可以用于初始化结构体类型，例如：

struct Person

{

std::string name;

int age;

};

int main()

{

Person p = {"Frank", 25};

std::cout << p.name << " : " << p.age << std::endl;

}

其他一些不方便初始化的地方使用，比如std<vector>的初始化，如果不使用这种方式，只能用构造函数来初始化，难以达到效果：

std::vector<int> ivec1(3, 5);

std::vector<int> ivec2 = {5, 5, 5};

std::vector<int> ivec3 = {1,2,3,4,5}; //不使用列表初始化用构造函数难以实现

### 3.1.3 防止类型收窄

类型收窄指的是导致数据内容发生变化或者精度丢失的隐式类型转换。使用列表初始化可以防止类型收窄。

int main(void)

{

const int x = 1024;

const int y = 10;

char a = x; // 收窄，但可以通过编译

char\* b = new char(1024); // 收窄，但可以通过编译

char c = { x }; // err, 收窄，无法通过编译

char d = { y }; // 可以通过编译

unsigned char e{ -1 }; // err,收窄，无法通过编译

float f{ 7 }; // 可以通过编译

int g{ 2.0f }; // err,收窄，无法通过编译

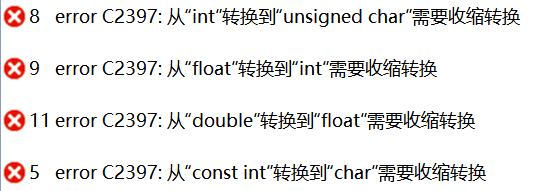
float \* h = new float{ 1e48 }; // err,收窄，无法通过编译

float i = 1.2l; // 可以通过编译

return 0;

}

vs错误提示：



## 3.2 基于范围的for循环

在C++中for循环可以使用基于范围的for循环，示例代码如下：

int main()

{

int a[5] = { 1, 2, 3, 4, 5 };

for (int & e: a)

{

e \*= 2;

}

for (int & e: a)

{

cout << e << ", ";

}

cout << endl;

return 0;

}

使用基于范围的for循环，其for循环迭代的范围必须是可确定的：

int func(int a[])//形参中数组是指针变量，无法确定元素个数

{

for(auto e: a) // err, 编译失败

{

cout << e;

}

}

int main()

{

int a[] = {1, 2, 3, 4, 5};

func(a);

return 0;

}

## 3.3 静态断言

C/C++提供了调试工具assert，这是一个宏，用于在**运行阶段**对断言进行检查，如果条件为真，执行程序，否则调用abort()。

int main()

{

bool flag = false;

//如果条件为真，程序正常执行，如果为假，终止程序，提示错误

assert(flag == true); //#include <cassert>或#include <assert.h>

cout << "Hello World!" << endl;

return 0;

}

C++ 11新增了关键字static\_assert，可用于在**编译阶段**对断言进行测试。

静态断言的好处：

* 更早的报告错误，我们知道构建是早于运行的，更早的错误报告意味着开发成本的降低
* 减少运行时开销，静态断言是编译期检测的，减少了运行时开销

语法如下：

static\_assert(常量表达式，提示字符串)

注意：只能是常量表达式，不能是变量

int main()

{

//该static\_assert用来确保编译仅在32位的平台上进行，不支持64位的平台

static\_assert( sizeof(void \*)== 4, "64-bit code generation is not supported.");

cout << "Hello World!" << endl;

return 0;

}

## 3.3 noexcept修饰符(vs2013不支持)

void func3() throw(int, char) //只能够抛出 int 和char类型的异常

{//C++11已经弃用这个声明

throw 0;

}

void BlockThrow() throw() //代表此函数不能抛出异常，如果抛出，就会异常

{

throw 1;

}

//代表此函数不能抛出异常，如果抛出，就会异常

//C++11 使用noexcept替代throw()

void BlockThrowPro() noexcept

{

throw 2;

}

## 3.4 nullptr

nullptr是为了解决原来C++中NULL的二义性问题而引进的一种新的类型，因为NULL实际上代表的是0。

void func(int a)

{

cout << \_\_LINE\_\_ << " a = " << a <<endl;

}

void func(int \*p)

{

cout << \_\_LINE\_\_ << " p = " << p <<endl;

}

int main()

{

int \*p1 = nullptr;

int \*p2 = NULL;

if(p1 == p2)

{

cout << "equal\n";

}

//int a = nullptr; //err, 编译失败，nullptr不能转型为int

func(0); //调用func(int)， 就算写NULL，也是调用这个

func(nullptr);

return 0;

}

## 3.5 强类型枚举

C++ 11引入了一种新的枚举类型，即“枚举类”，又称“强类型枚举”。声明请类型枚举非常简单，只需要在enum后加上使用class或struct。如：

enum Old{Yes, No}; // old style

enum class New{Yes, No}; // new style

enum struct New2{Yes, No}; // new style

“传统”的C++枚举类型有一些缺点：它会在一个代码区间中抛出枚举类型成员（如果在相同的代码域中的两个枚举类型具有相同名字的枚举成员，这会导致命名冲突），它们会被隐式转换为整型，并且不可以指定枚举的底层数据类型。

int main()

{

enum Status{Ok, Error};

//enum Status2{Ok, Error};//err, 导致命名冲突, Status已经有成员叫Ok, Error

return 0;

}

在C++11中，强类型枚举解决了这些问题：

int main()

{

enum class Status {Ok, Error};

enum struct Status2{Ok, Error};

//Status flag2 = Ok; // err，必须使用强类型名称

Status flag3 = Status::Ok;

enum class C : char { C1 = 1, C2 = 2};//指定枚举的底层数据类型

enum class D : unsigned int { D1 = 1, D2 = 2, Dbig = 0xFFFFFFF0U };

cout << sizeof(C::C1) << endl; // 1

cout << sizeof(D::D1) << endl; // 4

cout << sizeof(D::Dbig) << endl; // 4

return 0;

}

## 3.6 常量表达式(vs2013 不支持)

常量表达式主要是允许一些计算发生在编译时，即发生在代码编译而不是运行的时候。

这是很大的优化：假如有些事情可以在编译时做，它将只做一次，而不是每次程序运行时都计算。

使用constexpr，你可以创建一个编译时的函数：

constexpr int GetConst()

{

return 3;

}

int main()

{

int arr[ GetConst() ] = {0};

enum { e1 = GetConst(), e2 };

constexpr int num = GetConst();

return 0;

}

constexpr函数的限制：

* 函数中只能有一个return语句（有极少特例）
* 函数必须返回值（不能是void函数）
* 在使用前必须已有定义
* return返回语句表达式中不能使用非常量表达式的函数、全局数据，且必须是一个常量表达式

//err,函数中只能有一个return语句

constexpr int data()

{

constexpr int i = 1;

return i;

}

constexpr int data2()

{

//一个constexpr函数，只允许包含一行可执行代码

//但允许包含typedef、 using 指令、静态断言等。

static\_assert(1, "fail");

return 100;

}

int a = 3;

constexpr int data3()

{

return a;//err, return返回语句表达式中不能使用非常量表达式的函数、全局数据

}

int main()

{

constexpr int func(); //函数声明，定义放在main函数后面

constexpr int c = func(); //err, 无法通过编译, 在使用前必须已有定义

return 0;

}

constexpr int func()

{

return 1;

}

常量表达式的构造函数有以下限制：

* 函数体必须为空
* 初始化列表只能由常量表达式来赋值

struct Date

{

constexpr Date(int y, int m, int d): year(y), month(m), day(d) {}

constexpr int GetYear() { return year; }

constexpr int GetMonth() { return month; }

constexpr int GetDay() { return day; }

private:

int year;

int month;

int day;

};

int main()

{

constexpr Date PRCfound {1949, 10, 1};

constexpr int foundmonth = PRCfound.GetMonth();

cout << foundmonth << endl; // 10

return 0;

}

## 3.7 用户定义字面量(vs2013 不支持)

用户自定义字面值，或者叫“自定义后缀”更直观些，主要作用是简化代码的读写。

long double operator"" \_mm(long double x) { return x / 1000; }

long double operator"" \_m(long double x) { return x; }

long double operator"" \_km(long double x) { return x \* 1000; }

int main()

{

cout << 1.0\_mm << endl; //0.001

cout << 1.0\_m << endl; //1

cout << 1.0\_km << endl; //1000

return 0;

}

根据 C++ 11 标准，只有下面参数列表才是合法的：

char const \*

unsigned long long

long double

char const \*, size\_t

wchar\_t const \*, size\_t

char16\_t const \*, size\_t

char32\_t const \*, size\_t

最后四个对于字符串相当有用，因为第二个参数会自动推断为字符串的长度。例如：

size\_t operator"" \_len(char const \* str, size\_t size)

{

return size;

}

int main()

{

cout << "mike"\_len <<endl; //结果为4

return 0;

}

对于参数char const \*，应该被称为原始字面量 raw literal 操作符。例如：

char const \* operator"" \_r(char const\* str)

{

return str;

}

int main()

{

cout << 250\_r <<endl; //结果为250

return 0;

}

## 3.8 原生字符串字面值

原生字符串字面值(raw string literal)使用户书写的字符串“所见即所得”。C++11中原生字符串的声明相当简单，只需在字符串前加入前缀，即字母R，并在引号中使用括号左右标识，就可以声明该字符串字面量为原生字符串了。

int main()

{

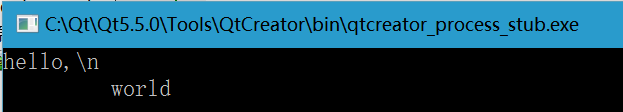
cout << R"(hello,\n

world)" << endl;

return 0;

}

运行结果：



# 4. 类的改进

## 4.1 继承构造(vs2013 不支持)

C++ 11允许派生类继承基类的构造函数（默认构造函数、复制构造函数、移动构造函数除外）。

class A

{

public:

A(int i) { cout << "i = " << i << endl; }

A(double d, int i) {}

A(float f, int i, const char\* c) {}

// ...

};

class B : public A

{

public:

using A::A; // 继承构造函数

// ...

virtual void *ExtraInterface*(){}

};

注意：

* 继承的构造函数只能初始化基类中的成员变量，不能初始化派生类的成员变量
* 如果基类的构造函数被声明为私有，或者派生类是从基类中虚继承，那么不能继承构造函数
* 一旦使用继承构造函数，编译器不会再为派生类生成默认构造函数

## 4.2 委托构造

和继承构造函数类似，委托构造函数也是C++11中对C++的构造函数的一项改进，其目的也是为了减少程序员书写构造函数的时间。

如果一个类包含多个构造函数，C++ 11允许在一个构造函数中的定义中使用另一个构造函数，但这必须通过初始化列表进行操作，如下：

class Info

{

public:

Info() : Info(1) { } // 委托构造函数

Info(int i) : Info(i, 'a') { } // 既是目标构造函数，也是委托构造函数

Info(char e): Info(1, e) { }

private:

Info(int i, char e): type(i), name(e) { /\* 其它初始化 \*/ } // 目标构造函数

int type;

char name;

// ...

};

## 4.3 继承控制：final和override

C++11之前，一直没有继承控制关键字，禁用一个类的进一步衍生比较麻烦。

C++ 11添加了两个继承控制关键字：final和override。

* final阻止类的进一步派生和虚函数的进一步重写
* override确保在派生类中声明的函数跟基类的虚函数有相同的签名

class B1 final {}; // 此类不能被继承

//class D1: public B1 {}; // error!

class B

{

public:

// virtual void func() override // error! 指定了重写但实际并没重写,没有基类

// {

// cout << \_\_func\_\_ << std::endl;

// }

virtual void *f*() const

{

cout << \_\_func\_\_ << std::endl;

}

virtual void *fun*()

{

cout << \_\_func\_\_ << std::endl;

}

};

class D : public B

{

public:

virtual void *f*(int) // ok! 隐藏,由于没有重写同名函数B::f,在D中变为不可见

{

cout << "hiding: " <<\_\_func\_\_ << std::endl;

}

// virtual void f() override // error! 指定了重写但实际并没重写,类型声明不完全相同

// {

// cout << \_\_func\_\_ << std::endl;

// }

virtual void *fun*() override final // ok! 指定了重写实际上也重写了，同时，指定为最终，后代类中不能再重写此虚函数

{

cout << \_\_func\_\_ << std::endl;

}

};

class D2 : public D

{

public:

virtual void *f*() const // ok! 重写B::f(),同时,由于没有重写D::f(int),在D2中变不可见

{

cout << \_\_func\_\_ << std::endl;

}

// virtual void fun() // error! 基类的此虚函数被指定为最终,不能被重写,虽然没有显示指定"override"

// {

// cout << \_\_func\_\_ << std::endl;

// }

// virtual void fun() override // error! 基类的此虚函数被指定为最终，不能被重写

// {

// cout << \_\_func\_\_ << std::endl;

// }

};

## 4.4 类默认函数的控制："=default" 和 "=delete"函数

### 4.4.1 "=default"函数

C++ 的类有四类特殊成员函数，它们分别是：默认构造函数、析构函数、拷贝构造函数以及拷贝赋值运算符。这些类的特殊成员函数负责创建、初始化、销毁，或者拷贝类的对象。如果程序员没有显式地为一个类定义某个特殊成员函数，而又需要用到该特殊成员函数时，则编译器会隐式的为这个类生成一个默认的特殊成员函数。

但是，如果程序员为类显式的自定义了非默认构造函数，编译器将不再会为它隐式地生成默认无参构造函数。

class X

{

public:

X(){} // 手动定义默认构造函数

X(int i)

{

a = i;

}

private:

int a;

};

X obj; //必须手动定义默认构造函数X(){} 才能编译通过

原本期望编译器自动生成的默认构造函数却需要程序员手动编写，即程序员的工作量加大了。此外，手动编写的默认构造函数的代码执行效率比编译器自动生成的默认构造函数低。

类的其它几类特殊成员函数也和默认构造函数一样，当存在用户自定义的特殊成员函数时，编译器将不会隐式的自动生成默认特殊成员函数，而需要程序员手动编写，加大了程序员的工作量。类似的，手动编写的特殊成员函数的代码执行效率比编译器自动生成的特殊成员函数低。

C++11 标准引入了一个新特性："=default"函数。程序员只需在函数声明后加上“=default;”，就可将该函数声明为 "=default"函数，编译器将为显式声明的 "=default"函数自动生成函数体。

class X

{

public:

X()= default; //该函数比用户自己定义的默认构造函数获得更高的代码效率

X(int i)

{

a = i;

}

private:

int a;

};

X obj;

"=default"函数特性仅适用于类的特殊成员函数，且该特殊成员函数没有默认参数。例如：

class X

{

public:

int f() = default; // err , 函数 f() 非类 X 的特殊成员函数

X(int, int) = default; // err , 构造函数 X(int, int) 非 X 的特殊成员函数

X(int = 1) = default; // err , 默认构造函数 X(int=1) 含有默认参数

};

"=default"函数既可以在类体里（inline）定义，也可以在类体外（out-of-line）定义。例如：

class X

{

public:

X() = default; //Inline defaulted 默认构造函数

X(const X&);

X& operator = (const X&);

~X() = default; //Inline defaulted 析构函数

};

X::X(const X&) = default; //Out-of-line defaulted 拷贝构造函数

X& X::operator= (const X&) = default; //Out-of-line defaulted 拷贝赋值操作符

### 4.4.2 "=delete"函数

为了能够让程序员显式的禁用某个函数，C++11 标准引入了一个新特性："=delete"函数。程序员只需在函数声明后上“=delete;”，就可将该函数禁用。

class X

{

public:

X();

X(const X&) = delete; // 声明拷贝构造函数为 deleted 函数

X& operator = (const X &) = delete; // 声明拷贝赋值操作符为 deleted 函数

};

int main()

{

X obj1;

X obj2=obj1; // 错误，拷贝构造函数被禁用

X obj3;

obj3=obj1; // 错误，拷贝赋值操作符被禁用

return 0;

}

"=delete"函数特性还可用于禁用类的某些转换构造函数，从而避免不期望的类型转换：

class X

{

public:

X(double)

{

}

X(int) = delete;

};

int main()

{

X obj1(1.2);

X obj2(2); // 错误，参数为整数 int 类型的转换构造函数被禁用

return 0;

}

"=delete"函数特性还可以用来禁用某些用户自定义的类的 new 操作符，从而避免在自由存储区创建类的对象：

class X

{

public:

void \*operator new(size\_t) = delete;

void \*operator new[](size\_t) = delete;

};

int main()

{

X \*pa = new X; // 错误，new 操作符被禁用

X \*pb = new X[10]; // 错误，new[] 操作符被禁用

return 0;

}

# 5. 模板的改进

## 5.1 右尖括号>改进

在C++98/03的泛型编程中，模板实例化有一个很繁琐的地方，就是连续两个右尖括号(>>)会被编译解释成右移操作符，而不是模板参数表的形式，需要一个空格进行分割，以避免发生编译时的错误。

template <int i> class X{};

template <class T> class Y{};

int main()

{

Y<X<1> > x1; // ok, 编译成功

Y<X<2>> x2; // err, 编译失败

return 0;

};

在实例化模板时会出现连续两个右尖括号，同样static\_cast、dynamic\_cast、reinterpret\_cast、const\_cast表达式转换时也会遇到相同的情况。C++98标准是让程序员在>>之间填上一个空格，在C++11中，这种限制被取消了。在C++11标准中，要求编译器对模板的右尖括号做单独处理，使编译器能够正确判断出">>"是一个右移操作符还是模板参数表的结束标记。

## 5.2 模板的别名

#include <iostream>

#include <type\_traits> //std::is\_same

using namespace std;

using uint = unsigned int;

typedef unsigned int UINT;

using sint = int;

int main()

{

//std::is\_same 判断类型是否一致

//这个结构体作用很简单，就是两个一样的类型会返回true

cout << is\_same<uint, UINT>::value << endl; // 1

return 0;

}

## 5.3 函数模板的默认模板参数

C++11之前，类模板是支持默认的模板参数，却不支持函数模板的默认模板参数：

//1、普通函数带默认参数，c++98 编译通过，c++11 编译通过

void DefParm(int m = 3) {}

//2、类模板是支持默认的模板参数，c++98 编译通过，c++11 编译通过

template <typename T = int>

class DefClass {};

//3、函数模板的默认模板参数， c++98 - 编译失败，c++11 - 编译通过

template <typename T = int> void DefTempParm() {}

类模板的默认模板参数必须从右往左定义，数模板的默认模板参数则没这个限定：

template<class T1, class T2 = int> class DefClass1;

template<class T1 = int, class T2> class DefClass2; // 无法通过编译

template<class T, int i = 0> class DefClass3;

template<int i = 0, class T> class DefClass4; // 无法通过编译

template<class T1 = int, class T2> void DefFunc1(T1 a, T2 b);

template<int i = 0, class T> void DefFunc2(T a);

# 6. 可变参数的模板

在C++11之前，类模板和函数模板只能含有固定数量的模板参数。C++11增强了模板功能，允许模板定义中包含0到任意个模板参数，这就是可变参数模板。

可变参数模板和普通模板的语义是一样的，只是写法上稍有区别，声明可变参数模板时需要在typename或class后面带上省略号“...”：

template<class ... T> void func(T ... args)//T叫模板参数包，args叫函数参数包

{//可变参数模板函数

}

func(); // OK：args不含有任何实参

func(1); // OK：args含有一个实参：int

func(2, 1.0); // OK：args含有两个实参int和double

省略号“...”的作用有两个：

1. 声明一个参数包，这个参数包中可以包含0到任意个模板参数
2. 在模板定义的右边，可以将参数包展开成一个一个独立的参数

## 6.1 可变参数模板函数

### 6.1.1 可变参数模板函数的定义

一个可变参数模板函数的定义如下：

template<class ... T> void func(T ... args)

{//可变参数模板函数

//sizeof...（sizeof后面有3个小点）计算变参个数

cout << "num = " << sizeof...(args) << endl;

}

int main()

{

func(); // num = 0

func(1); // num = 1

func(2, 1.0); // num = 2

return 0;

}

### 6.1.2 参数包的展开

6.1.2.1 递归方式展开

通过递归函数展开参数包，需要提供一个参数包展开的函数和一个递归终止函数。

//递归终止函数

void debug()

{

cout << "empty\n";

}

//展开函数

template <class T, class ... Args>

void debug(T first, Args ... last)

{

cout << "parameter " << first << endl;

debug(last...);

}

int main()

{

debug(1, 2, 3, 4);

/\*

运行结果：

parameter 1

parameter 2

parameter 3

parameter 4

empty

\*/

return 0;

}

递归调用过程如下：

debug(1, 2, 3, 4);

debug(2, 3, 4);

debug(3, 4);

debug(4);

debug();

6.1.2.2 非递归方式展开

template <class T>

void print(T arg)

{

cout << arg << endl;

}

template <class ... Args>

void expand(Args ... args)

{

int a[] = { (print(args), 0)... };

}

int main()

{

expand(1, 2, 3, 4);

return 0;

}

expand函数的逗号表达式：(print(args), 0)， 也是按照这个执行顺序，先执行print(args)，再得到逗号表达式的结果0。

同时，通过初始化列表来初始化一个变长数组，{ (print(args), 0)... }将会展开成( (print(args1), 0), (print(args2), 0), (print(args3), 0), etc...), 最终会创建一个元素只都为0的数组int a[sizeof...(args)]。

## 6.2 可变参数模板类

### 6.2.1 继承方式展开参数包

可变参数模板类的展开一般需要定义2 ~ 3个类，包含类声明和特化的模板类：

template<typename... A> class BMW{}; // 变长模板的声明

template<typename Head, typename... Tail> // 递归的偏特化定义

class BMW<Head, Tail...> : public BMW<Tail...>

{//当实例化对象时，则会引起基类的递归构造

public:

BMW()

{

printf("type: %s\n", typeid(Head).name());

}

Head head;

};

template<> class BMW<>{}; // 边界条件

int main()

{

BMW<int, char, float> car;

/\*

运行结果：

type: f

type: c

type: i

\*/

return 0;

}

### 6.2.2 模板递归和特化方式展开参数包

template <long... nums> struct Multiply;// 变长模板的声明

template <long first, long... last>

struct Multiply<first, last...> // 变长模板类

{

static const long val = first \* Multiply<last...>::val;

};

template<>

struct Multiply<> // 边界条件

{

static const long val = 1;

};

int main()

{

cout << Multiply<2, 3, 4, 5>::val << endl; // 120

return 0;

}

# 7. 右值引用

## 7.1 左值引用、右值引用

### 7.1.1 左值、右值

在C语言中，我们常常会提起左值（lvalue）、右值（rvalue）这样的称呼。一个最为典型的判别方法就是，在赋值表达式中，出现在等号左边的就是“左值”，而在等号右边的，则称为“右值”。如：

int b = 1;

int c = 2;

int a = a + b;

在这个赋值表达式中，a就是一个左值，而b + c则是一个右值。

不过C++中还有一个被广泛认同的说法，那就是可以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值。那么这个加法赋值表达式中，&a是允许的操作，但&(b + c)这样的操作则不会通过编译。因此a是一个左值，(b + c)是一个右值。

相对于左值，右值表示字面常量、表达式、函数的非引用返回值等。

### 7.1.2 左值引用、右值引用

左值引用是对一个左值进行引用的类型，右值引用则是对一个右值进行引用的类型。

左值引用和右值引用都是属于引用类型。无论是声明一个左值引用还是右值引用，都必须立即进行初始化。而其原因可以理解为是引用类型本身自己并不拥有所绑定对象的内存，只是该对象的一个别名。

左值引用是具名变量值的别名，而右值引用则是不具名（匿名）变量的别名。

左值引用：

int &a = 2; // 左值引用绑定到右值，编译失败, err

int b = 2; // 非常量左值

const int &c = b; // 常量左值引用绑定到非常量左值，编译通过, ok

const int d = 2; // 常量左值

const int &e = c; // 常量左值引用绑定到常量左值，编译通过, ok

const int &b = 2; // 常量左值引用绑定到右值，编程通过, ok

“const 类型 &”为 “万能”的引用类型，它可以接受非常量左值、常量左值、右值对其进行初始化；

右值引用，使用&&表示：

int && r1 = 22;

int x = 5;

int y = 8;

int && r2 = x + y;

T && a = ReturnRvalue();

通常情况下，右值引用是不能够绑定到任何的左值的。

int c;

int && d = c; //err

测试示例：

void process\_value(int & i) //参数为左值引用

{

cout << "LValue processed: " << i << endl;

}

void process\_value(int && i) //参数为右值引用

{

cout << "RValue processed: " << i << endl;

}

int main()

{

int a = 0;

process\_value(a); //LValue processed: 0

process\_value(1); //RValue processed: 1

return 0;

}

## 7.2 移动语义

### 7.2.1 为什么需要移动语义

右值引用是用来支持转移语义的。转移语义可以将资源 ( 堆，系统对象等 ) 从一个对象转移到另一个对象，这样能够减少不必要的临时对象的创建、拷贝以及销毁，能够大幅度提高 C++ 应用程序的性能。临时对象的维护 ( 创建和销毁 ) 对性能有严重影响。

转移语义是和拷贝语义相对的，可以类比文件的剪切与拷贝，当我们将文件从一个目录拷贝到另一个目录时，速度比剪切慢很多。

通过转移语义，临时对象中的资源能够转移其它的对象里。

### 7.2.2 移动语义定义

在现有的 C++ 机制中，我们可以定义拷贝构造函数和赋值函数。要实现转移语义，需要定义转移构造函数，还可以定义转移赋值操作符。对于右值的拷贝和赋值会调用转移构造函数和转移赋值操作符。

如果转移构造函数和转移拷贝操作符没有定义，那么就遵循现有的机制，拷贝构造函数和赋值操作符会被调用。

普通的函数和操作符也可以利用右值引用操作符实现转移语义。

### 7.2.3 转移构造函数

class MyString

{

public:

MyString(const char \*tmp = "abc")

{//普通构造函数

len = strlen(tmp); //长度

str = new char[len+1]; //堆区申请空间

strcpy(str, tmp); //拷贝内容

cout << "普通构造函数 str = " << str << endl;

}

MyString(const MyString &tmp)

{//拷贝构造函数

len = tmp.len;

str = new char[len + 1];

strcpy(str, tmp.str);

cout << "拷贝构造函数 tmp.str = " << tmp.str << endl;

}

//移动构造函数

//参数是非const的右值引用

MyString(MyString && t)

{

str = t.str; //拷贝地址，没有重新申请内存

len = t.len;

//原来指针置空

t.str = NULL;

cout << "移动构造函数" << endl;

}

MyString &operator= (const MyString &tmp)

{//赋值运算符重载函数

if(&tmp == this)

{

return \*this;

}

//先释放原来的内存

len = 0;

delete []str;

//重新申请内容

len = tmp.len;

str = new char[len + 1];

strcpy(str, tmp.str);

cout << "赋值运算符重载函数 tmp.str = " << tmp.str << endl;

return \*this;

}

~MyString()

{//析构函数

cout << "析构函数: ";

if(str != NULL)

{

cout << "已操作delete, str = " << str;

delete []str;

str = NULL;

len = 0;

}

cout << endl;

}

private:

char \*str = NULL;

int len = 0;

};

MyString func() //返回普通对象，不是引用

{

MyString obj("mike");

return obj;

}

int main()

{

MyString &&tmp = func(); //右值引用接收

return 0;

}

和拷贝构造函数类似，有几点需要注意：

* 参数（右值）的符号必须是右值引用符号，即“&&”。
* 参数（右值）不可以是常量，因为我们需要修改右值。
* 参数（右值）的资源链接和标记必须修改，否则，右值的析构函数就会释放资源，转移到新对象的资源也就无效了。

有了右值引用和转移语义，我们在设计和实现类时，对于需要动态申请大量资源的类，应该设计转移构造函数和转移赋值函数，以提高应用程序的效率。

### 7.2.4 转移赋值函数

class MyString

{

public:

MyString(const char \*tmp = "abc")

{//普通构造函数

len = strlen(tmp); //长度

str = new char[len+1]; //堆区申请空间

strcpy(str, tmp); //拷贝内容

cout << "普通构造函数 str = " << str << endl;

}

MyString(const MyString &tmp)

{//拷贝构造函数

len = tmp.len;

str = new char[len + 1];

strcpy(str, tmp.str);

cout << "拷贝构造函数 tmp.str = " << tmp.str << endl;

}

//移动构造函数

//参数是非const的右值引用

MyString(MyString && t)

{

str = t.str; //拷贝地址，没有重新申请内存

len = t.len;

//原来指针置空

t.str = NULL;

cout << "移动构造函数" << endl;

}

MyString &operator= (const MyString &tmp)

{//赋值运算符重载函数

if(&tmp == this)

{

return \*this;

}

//先释放原来的内存

len = 0;

delete []str;

//重新申请内容

len = tmp.len;

str = new char[len + 1];

strcpy(str, tmp.str);

cout << "赋值运算符重载函数 tmp.str = " << tmp.str << endl;

return \*this;

}

//移动赋值函数

//参数为非const的右值引用

MyString &operator=(MyString &&tmp)

{

if(&tmp == this)

{

return \*this;

}

//先释放原来的内存

len = 0;

delete []str;

//无需重新申请堆区空间

len = tmp.len;

str = tmp.str; //地址赋值

tmp.str = NULL;

cout << "移动赋值函数\n";

return \*this;

}

~MyString()

{//析构函数

cout << "析构函数: ";

if(str != NULL)

{

cout << "已操作delete, str = " << str;

delete []str;

str = NULL;

len = 0;

}

cout << endl;

}

private:

char \*str = NULL;

int len = 0;

};

MyString func() //返回普通对象，不是引用

{

MyString obj("mike");

return obj;

}

int main()

{

MyString tmp("abc"); //实例化一个对象

tmp = func();

return 0;

}

## 7.3 标准库函数 std::move

既然编译器只对右值引用才能调用转移构造函数和转移赋值函数，而所有命名对象都只能是左值引用，如果已知一个命名对象不再被使用而想对它调用转移构造函数和转移赋值函数，也就是把一个左值引用当做右值引用来使用，怎么做呢？标准库提供了函数 std::move，这个函数以非常简单的方式将左值引用转换为右值引用。

int a;

int &&r1 = a; // 编译失败

int &&r2 = std::move(a); // 编译通过

## 7.4 完美转发 std::forward

完美转发适用于这样的场景：需要将一组参数原封不动的传递给另一个函数。

“原封不动”不仅仅是参数的值不变，在 C++ 中，除了参数值之外，还有一下两组属性：左值／右值和 const/non-const。完美转发就是在参数传递过程中，所有这些属性和参数值都不能改变，同时，而不产生额外的开销，就好像转发者不存在一样。在泛型函数中，这样的需求非常普遍。

下面举例说明：

#include <iostream>

using namespace std;

template <typename T> void process\_value(T & val)

{

cout << "T &" << endl;

}

template <typename T> void process\_value(const T & val)

{

cout << "const T &" << endl;

}

//函数 forward\_value 是一个泛型函数，它将一个参数传递给另一个函数 process\_value

template <typename T> void forward\_value(const T& val)

{

process\_value(val);

}

template <typename T> void forward\_value(T& val)

{

process\_value(val);

}

int main()

{

int a = 0;

const int &b = 1;

//函数 forward\_value 为每一个参数必须重载两种类型，T& 和 const T&

forward\_value(a); // T&

forward\_value(b); // const T &

forward\_value(2); // const T&

return 0;

}

对于一个参数就要重载两次，也就是函数重载的次数和参数的个数是一个正比的关系。这个函数的定义次数对于程序员来说，是非常低效的。

那C++11是如何解决完美转发的问题的呢？实际上，C++11是通过引入一条所谓“引用折叠”（reference collapsing）的新语言规则，并结合新的模板推导规则来完成完美转发。

typedef const int T;

typedef T & TR;

TR &v = 1; //在C++11中，一旦出现了这样的表达式，就会发生引用折叠，即将复杂的未知表达式折叠为已知的简单表达式

C++11中的引用折叠规则：

| **TR的类型定义** | **声明v的类型** | **v的实际类型** |
| --- | --- | --- |
| T & | TR | T & |
| T & | TR & | T & |
| T & | TR && | T & |
| T && | TR | T && |
| T && | TR & | T & |
| T && | TR && | T && |

**一旦定义中出现了左值引用，引用折叠总是优先将其折叠为左值引用**

C++11中，std::forward可以保存参数的左值或右值特性：

#include <iostream>

using namespace std;

template <typename T> void process\_value(T & val)

{

cout << "T &" << endl;

}

template <typename T> void process\_value(T && val)

{

cout << "T &&" << endl;

}

template <typename T> void process\_value(const T & val)

{

cout << "const T &" << endl;

}

template <typename T> void process\_value(const T && val)

{

cout << "const T &&" << endl;

}

//函数 forward\_value 是一个泛型函数，它将一个参数传递给另一个函数 process\_value

template <typename T> void forward\_value(T && val) //参数为右值引用

{

process\_value( std::forward<T>(val) );//C++11中，std::forward可以保存参数的左值或右值特性

}

int main()

{

int a = 0;

const int &b = 1;

forward\_value(a); // T &

forward\_value(b); // const T &

forward\_value(2); // T &&

forward\_value( std::move(b) ); // const T &&

return 0;

}

# 8. 智能指针

C++11中有unique\_ptr、shared\_ptr与weak\_ptr等智能指针(smart pointer)，定义在<memory>中。可以对动态资源进行管理，保证任何情况下，已构造的对象最终会销毁，即它的析构函数最终会被调用。

## 8.1 unique\_ptr

unique\_ptr持有对对象的独有权，同一时刻只能有一个unique\_ptr指向给定对象（通过禁止拷贝语义、只有移动语义来实现）。

unique\_ptr指针本身的生命周期：从unique\_ptr指针创建时开始，直到离开作用域。

离开作用域时，若其指向对象，则将其所指对象销毁(默认使用delete操作符，用户可指定其他操作)。

#include <iostream>

#include <memory>

using namespace std;

int main()

{

unique\_ptr<int> up1(new int(11)); // 无法复制的unique\_ptr

//unique\_ptr<int> up2 = up1; // err, 不能通过编译

cout << \*up1 << endl; // 11

unique\_ptr<int> up3 = move(up1); // 现在p3是数据的唯一的unique\_ptr

cout << \*up3 << endl; // 11

//cout << \*up1 << endl; // err, 运行时错误

up3.reset(); // 显式释放内存

up1.reset(); // 不会导致运行时错误

//cout << \*up3 << endl; // err, 运行时错误

unique\_ptr<int> up4(new int(22)); // 无法复制的unique\_ptr

up4.reset(new int(44)); //"绑定"动态对象

cout << \*up4 << endl;

up4 = nullptr;//显式销毁所指对象，同时智能指针变为空指针。与up4.reset()等价

unique\_ptr<int> up5(new int(55));

int \*p = up5.release(); //只是释放控制权，不会释放内存

cout << \*p << endl;

//cout << \*up5 << endl; // err, 运行时错误

delete p; //释放堆区资源

return 0;

}

## 8.2 shared\_ptr

shared\_ptr允许多个该智能指针共享第“拥有”同一堆分配对象的内存，这通过引用计数（reference counting）实现，会记录有多少个shared\_ptr共同指向一个对象，一旦最后一个这样的指针被销毁，也就是一旦某个对象的引用计数变为0，这个对象会被自动删除。

int main()

{

shared\_ptr<int> sp1(new int(22));

shared\_ptr<int> sp2 = sp1;

cout << "count: " << sp2.use\_count() << endl; //打印引用计数

cout << \*sp1 << endl; // 22

cout << \*sp2 << endl; // 22

sp1.reset(); //显式让引用计数减1

cout << "count: " << sp2.use\_count() << endl; //打印引用计数

cout << \*sp2 << endl; // 22

return 0;

}

## 8.3 weak\_ptr

weak\_ptr是为配合shared\_ptr而引入的一种智能指针来协助shared\_ptr工作，它可以从一个shared\_ptr或另一个weak\_ptr对象构造，它的构造和析构不会引起引用计数的增加或减少。没有重载 \* 和 -> 但可以使用lock获得一个可用的shared\_ptr对象

weak\_ptr的使用更为复杂一点，它可以指向shared\_ptr指针指向的对象内存，却并不拥有该内存，而使用weak\_ptr成员lock，则可返回其指向内存的一个share\_ptr对象，且在所指对象内存已经无效时，返回指针空值nullptr。

void check(weak\_ptr<int> &wp)

{

shared\_ptr<int> sp = wp.lock(); // 转换为shared\_ptr<int>

if (sp != nullptr)

{

cout << "still " << \*sp << endl;

}

else

{

cout << "pointer is invalid" << endl;

}

}

int main()

{

shared\_ptr<int> sp1(new int(22));

shared\_ptr<int> sp2 = sp1;

weak\_ptr<int> wp = sp1; // 指向shared\_ptr<int>所指对象

cout << "count: " << wp.use\_count() << endl; //打印计数器

cout << \*sp1 << endl; // 22

cout << \*sp2 << endl; // 22

check(wp); // still 22

sp1.reset();

cout << "count: " << wp.use\_count() << endl;

cout << \*sp2 << endl; // 22

check(wp); // still 22

sp2.reset();

cout << "count: " << wp.use\_count() << endl;

check(wp); // pointer is invalid

return 0;

}

# 9. 闭包的实现

## 9.1 什么是闭包

闭包有很多种定义，一种说法是，闭包是带有上下文的函数。说白了，就是有状态的函数。更直接一些，不就是个类吗？换了个名字而已。

一个函数，带上了一个状态，就变成了闭包了。那什么叫 “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。

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。

闭包的状态捆绑，必须发生在运行时。

## 9.2 闭包的实现

### 9.2.1 仿函数：重载 operator()

class MyFunctor

{

public:

MyFunctor(int tmp) : round(tmp) {}

int operator()(int tmp) { return tmp + round; }

private:

int round;

};

int main()

{

int round = 2;

MyFunctor f(round);//调用构造函数

cout << "result = " << f(1) << endl; //operator()(int tmp)

return 0;

}

### 9.2.2 std::bind绑定器

9.2.2.1 std::function

在C++中，可调用实体主要包括：函数、函数指针、函数引用、可以隐式转换为函数指定的对象，或者实现了opetator()的对象。

C++11中，新增加了一个std::function类模板，它是对C++中现有的可调用实体的一种类型安全的包裹。通过指定它的模板参数，它可以用统一的方式处理函数、函数对象、函数指针，并允许保存和延迟执行它们。

#include <iostream>

#include <functional> //std::cout

using namespace std;

void func(void)

{//普通全局函数

cout << \_\_func\_\_ << endl;

}

class Foo

{

public:

static int foo\_func(int a)

{//类中静态函数

cout << \_\_func\_\_ << "(" << a << ") ->: ";

return a;

}

};

class Bar

{

public:

int operator()(int a)

{//仿函数

cout << \_\_func\_\_ << "(" << a << ") ->: ";

return a;

}

};

int main()

{

//绑定一个普通函数

function< void(void) > f1 = func;

f1();

//绑定类中的静态函数

function< int(int) > f2 = Foo::foo\_func;

cout << f2(111) << endl;

//绑定一个仿函数

Bar obj;

f2 = obj;

cout << f2(222) << endl;

/\*

运行结果：

func

foo\_func(111) ->: 111

operator()(222) ->: 222

\*/

return 0;

}

std::function对象最大的用处就是在实现函数回调，使用者需要注意，它不能被用来检查相等或者不相等，但是可以与NULL或者nullptr进行比较。

9.2.2.2 std::bind

std::bind是这样一种机制，它可以预先把指定可调用实体的某些参数绑定到已有的变量，产生一个新的可调用实体，这种机制在回调函数的使用过程中也颇为有用。

C++98中，有两个函数bind1st和bind2nd，它们分别可以用来绑定functor的第一个和第二个参数，它们都是只可以绑定一个参数，各种限制，使得bind1st和bind2nd的可用性大大降低。

在C++11中，提供了std::bind，它绑定的参数的个数不受限制，绑定的具体哪些参数也不受限制，由用户指定，这个bind才是真正意义上的绑定。

std::bind的基本语法：

#include <iostream>

#include <functional> //std::bind

using namespace std;

void func(int x, int y)

{

cout << x << " " << y << endl;

}

int main()

{

bind(func, 1, 2)(); //输出：1 2

bind(func, std::placeholders::\_1, 2)(1);//输出：1 2

using namespace std::placeholders; // adds visibility of \_1, \_2, \_3,...

bind(func, 2, \_1)(1); //输出：2 1

bind(func, 2, \_2)(1, 2); //输出：2 2

bind(func, \_1, \_2)(1, 2); //输出：1 2

bind(func,\_2, \_1)(1, 2); //输出：2 1

//err, 调用时没有第二个参数

//bind(func, 2, \_2)(1);

return 0;

}

std::placeholders::\_1是一个占位符，代表这个位置将在函数调用时，被传入的第一个参数所替代。

9.2.2.3 std::bind和std::function配合使用

#include <iostream>

#include <functional> //std::cout

using namespace std;

using namespace std::placeholders; // adds visibility of \_1, \_2, \_3,...

class Test

{

public:

int i = 0;

void func(int x, int y)

{

cout << x << " " << y << endl;

}

};

int main()

{

Test obj; //创建对象

function<void(int, int)> f1 = bind(&Test::func, &obj, \_1, \_2);

f1(1, 2); //输出：1 2

function< int &()> f2 = bind(&Test::i, &obj);

f2() = 123;

cout << obj.i << endl;//结果为 123

return 0;

}

通过std::bind和std::function配合使用，所有的可调用对象均有了统一的操作方法。

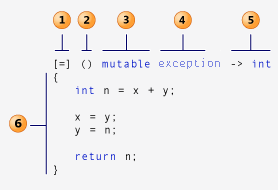
### 9.2.3 lambda表达式

9.2.3.1 lambda基础使用

lambda 表达式(lambda expression)是一个匿名函数，lambda表达式基于数学中的 λ 演算得名。

C++11中的lambda表达式用于定义并创建匿名的函数对象，以简化编程工作。

lambda表达式的基本构成：



**① 函数对象参数**

[]，标识一个**lambda的开始**，这部分必须存在，**不能省略**。函数对象参数是传递给编译器自动生成的函数对象类的构造函数的。函数对象参数只能使用那些到定义lambda为止时lambda所在作用范围内可见的局部变量（包括lambda所在类的this）。函数对象参数有以下形式：

* + 空。没有使用任何函数对象参数。
  + =。函数体内可以使用lambda所在作用范围内所有可见的局部变量（包括lambda所在类的this），并且是**值传递方式**（相当于编译器自动为我们按值传递了所有局部变量）。
  + &。函数体内可以使用lambda所在作用范围内所有可见的局部变量（包括lambda所在类的this），并且是**引用传递方式**（相当于编译器自动为我们按引用传递了所有局部变量）。
  + this。函数体内可以使用lambda所在类中的成员变量。
  + a。将a按值进行传递。按值进行传递时，函数体内不能修改传递进来的a的拷贝，因为默认情况下函数是const的。要修改传递进来的a的拷贝，可以添加mutable修饰符。
  + &a。将a按引用进行传递。
  + a, &b。将a按值进行传递，b按引用进行传递。
  + =，&a, &b。除a和b按引用进行传递外，其他参数都按值进行传递。
  + &, a, b。除a和b按值进行传递外，其他参数都按引用进行传递。

**② 操作符重载函数参数**

标识重载的()操作符的参数，没有参数时，这部分可以省略。参数可以通过按值（如：(a,b)）和按引用（如：(&a,&b)）两种方式进行传递。

**③ 可修改标示符**

mutable声明，这部分可以省略。按值传递函数对象参数时，加上mutable修饰符后，可以修改按值传递进来的拷贝（注意是能修改拷贝，而不是值本身）。

**④ 错误抛出标示符**

exception声明，这部分也可以省略。exception声明用于指定函数抛出的异常，如抛出整数类型的异常，可以使用throw(int)

**⑤ 函数返回值**

->返回值类型，标识函数返回值的类型，当返回值为void，或者函数体中只有一处return的地方（此时编译器可以自动推断出返回值类型）时，这部分可以省略。

**⑥ 是函数体**

{}，标识函数的实现，这部分不能省略，但函数体可以为空。

class Test

{

public:

int i = 0;

void func(int x, int y)

{

auto x1 = []{ return i; }; //err, 没有捕获外部变量

auto x2 = [=]{ return i+x+y; }; //ok, 值传递方式捕获所有外部变量

auto x3 = [=]{ return i+x+y; }; //ok, 引用传递方式捕获所有外部变量

auto x4 = [this]{ return i; }; //ok, 捕获this指针

auto x5 = [this]{ return i+x+y; }; //err, 没有捕获x, y

auto x6 = [this, x, y]{ return i+x+y; };//ok, 捕获this指针, x, y

auto x9 = [this]{ return i++; }; //ok, 捕获this指针, 并修改成员的值

}

};

int main()

{

int a = 0, b = 1;

auto f1 = []{ return a; }; //err, 没有捕获外部变量

auto f2 = [=]{ return a; }; //ok, 值传递方式捕获所有外部变量

auto f3 = [=]{ return a++; }; //err, a是以赋值方式捕获的，无法修改

auto f4 = [=]() mutable { return a++; }; //ok, 加上mutable修饰符后，可以修改按值传递进来的拷贝

auto f5 = [&]{ return a++; }; //ok, 引用传递方式捕获所有外部变量, 并对a执行自加运算

auto f6 = [a]{ return a+b; }; //err, 没有捕获变量b

auto f9 = [a,&b]{ return a+(b++); }; //ok, 捕获a, &b

auto f8 = [=,&b]{ return a+(b++); }; //ok, 捕获所有外部变量，&b

return 0;

}

值传递和引用传递区别：

int main()

{

int j = 12;

auto by\_val\_lambda = [=] { return j + 1;};

auto by\_ref\_lambda = [&] { return j + 1;};

cout << "by\_val\_lambda: " << by\_val\_lambda() << endl;

cout << "by\_ref\_lambda: " << by\_ref\_lambda() << endl;

j++;

cout << "by\_val\_lambda: " << by\_val\_lambda() << endl;

cout << "by\_ref\_lambda: " << by\_ref\_lambda() << endl;

/\*

运行结果：

by\_val\_lambda: 13

by\_ref\_lambda: 13

by\_val\_lambda: 13

by\_ref\_lambda: 14

\*/

return 0;

}

第3次调用结果还是13，原因是由于by\_val\_lambda中，j被视为了一个常量，一旦初始化后不会再改变。

9.2.3.2 lambda与仿函数

class MyFunctor

{

public:

MyFunctor(int tmp) : round(tmp) {}

int operator()(int tmp) { return tmp + round; }

private:

int round;

};

int main()

{

//仿函数

int round = 2;

MyFunctor f1(round);//调用构造函数

cout << "result1 = " << f1(1) << endl; //operator()(int tmp)

//lambda表达式

auto f2 = [=](int tmp) -> int { return tmp + round; } ;

cout << "result2 = " << f2(1) << endl;

return 0;

}

通过上面的例子，我们看到，仿函数以round初始化类，而lambda函数也捕获了round变量，其它的，如果在参数传递上，两者保持一致。

除去在语法层面上的不同，lambda和仿函数有着相同的内涵——都可以捕获一些变量作为初始化状态，并接受参数进行运行。

而事实上，仿函数是编译器实现lambda的一种方式，通过编译器都是把lambda表达式转化为一个仿函数对象。因此，在C++11中，lambda可以视为仿函数的一种等价形式。

9.2.3.3 lambda类型

lambda表达式的类型在C++11中被称为“闭包类型”，每一个lambda表达式则会产生一个临时对象(右值)。因此，严格地将，lambda函数并非函数指针。

不过C++11标准却允许lambda表达式向函数指针的转换，但提前是lambda函数没有捕获任何变量，且函数指针所示的函数原型，必须跟lambda函数函数有着相同的调用方式。

int main()

{

//使用std::function和std::bind来存储和操作lambda表达式

function<int(int)> f1 = [](int a) { return a; };

function<int()> f2 = bind([](int a){ return a; }, 123);

cout << "f1 = " << f1(123) << endl;

cout << "f2 = " << f2() << endl;

auto f3 = [](int x, int y)->int{ return x + y; }; //lambda表达式，没有捕获任何外部变量

typedef int (\*PF1)(int x, int y); //函数指针类型

typedef int (\*PF2)(int x);

PF1 p1; //函数指针变量

p1 = f3; //ok, lambda表达式向函数指针的转换

cout << "p1 = " << p1(3, 4) << endl;

PF2 p2;

p2 = f3; //err, 编译失败，参数必须一致

decltype(f3) p3 = f3; // 需通过decltype获得lambda的类型

decltype(f3) p4 = p1; // err 编译失败，函数指针无法转换为lambda

return 0;

}

9.2.3.4 lambda优势

#include <vector>

#include <algorithm> //std::for\_each

#include <iostream>

using namespace std;

vector<int> nums;

vector<int> largeNums;

class LNums

{

public:

LNums(int u): ubound(u){} //构造函数

void operator () (int i) const

{//仿函数

if (i > ubound)

{

largeNums.push\_back(i);

}

}

private:

int ubound;

};

int main()

{

//初始化数据

for(auto i = 0; i < 10; ++i)

{

nums.push\_back(i);

}

int ubound = 5;

//1、传统的for循环

for (auto itr = nums.begin(); itr != nums.end(); ++itr)

{

if (\*itr > ubound)

{

largeNums.push\_back(\*itr);

}

}

//2、使用仿函数

for\_each(nums.begin(), nums.end(), LNums(ubound));

//3、使用lambda函数和算法for\_each

for\_each(nums.begin(), nums.end(), [=](int i)

{

if (i > ubound)

{

largeNums.push\_back(i);

}

}

);

//4、遍历元素

for\_each(largeNums.begin(), largeNums.end(), [=](int i)

{

cout << i << ", ";

}

);

cout << endl;

return 0;

}

lambda表达式的价值在于，就地封装短小的功能闭包，可以及其方便地表达出我们希望执行的具体操作，并让上下文结合更加紧密。

# 10. 线程

在C++11之前，C/C++一直是一种顺序的编程语言。顺序是指所有指令都是串行执行的，即在相同的时刻，有且仅有单个CPU的程序计数器执行代码的代码段，并运行代码段中的指令。而C/C++代码也总是对应地拥有一份操作系统赋予进程的包括堆、栈、可执行的（代码）及不可执行的（数据）在内的各种内存区域。

而在C++11中，一个相当大的变化就是引入了多线程的支持。这使得C/C++语言在进行线程编程时，不比依赖第三方库。

## 10.1 线程的使用

### 10.1.1 线程的创建

用std::thread创建线程非常简单，只需要提供线程函数或函数对象即可，并且可以同时指定线程函数的参数。

#include <thread>

#include <iostream>

using namespace std;

void func1()

{

while(1)

{

cout << \_\_func\_\_ << endl;

}

}

void func2()

{

while(1)

{

cout << \_\_func\_\_ << endl;

}

}

int main()

{

thread t1(func1); //子线程1

thread t2(func2); //子线程2

while(1)//主线程

{

cout << \_\_func\_\_ << endl;

}

return 0;

}

线程还可以接收任意个数的参数：

void func(int a, char ch, const char \*str)

{

std::cout << "a = " << a << "\n";

std::cout << "ch = " << ch << "\n";

std::cout << "str = " << str << "\n";

}

int main()

{

std::thread t(func, 1, 'a', "mike"); //子线程, 需要头文件#include <thread>

while(1); //特地写一个死循环，让程序不结束

return 0;

}

### 10.1.2 回收线程资源

std::thread::join等待线程结束（此函数会阻塞），并回收线程资源，如果线程函数有返回值，返回值将被忽略。

#include <iostream> // std::cout

#include <thread> // std::thread, std::this\_thread::sleep\_for

#include <chrono> // std::chrono::seconds

using namespace std;

void pause\_thread(int n)

{

//指定当前线程休眠一定的时间

this\_thread::sleep\_for(chrono::seconds(n));

cout << "pause of " << n << " seconds ended\n";

}

int main()

{

cout << "Spawning 3 threads...\n";

thread t1(pause\_thread, 1);

thread t2(pause\_thread, 2);

thread t3(pause\_thread, 3);

cout << "Done spawning threads. Now waiting for them to join:\n";

t1.join();//等待线程结束（此函数会阻塞）

t2.join();

t3.join();

cout << "All threads joined!\n";

return 0;

}

如果不希望线程被阻塞执行，可以调用线程的std::thread::detach，将线程和线程对象分离，让线程作为后台线程去执行。但需要注意的是，detach之后就无法在和线程发生联系了，比如detach之后就不能再通过join来等待执行完，线程何时执行完我们也无法控制。

void pause\_thread(int n)

{

this\_thread::sleep\_for (chrono::seconds(n));

cout << "pause of " << n << " seconds ended\n";

}

int main()

{

cout << "Spawning and detaching 3 threads...\n";

thread(pause\_thread,1).detach();

thread(pause\_thread,2).detach();

thread(pause\_thread,3).detach();

cout << "Done spawning threads.\n";

cout << "(the main thread will now pause for 5 seconds)\n";

// give the detached threads time to finish (but not guaranteed!):

pause\_thread(5);

return 0;

}

### 10.1.3 获取线程ID和CPU核心数

void func()

{

this\_thread::sleep\_for (chrono::seconds(1));//休眠1秒

//获取当前线程id

cout << "func id = " << this\_thread::get\_id() << endl;

}

int main()

{

thread t(func);

cout << "t.get\_id() = " << t.get\_id() << endl; //获取线程t的id

cout << "main id = "<<this\_thread::get\_id() << endl; //主线程id

cout << "cup num = " << thread::hardware\_concurrency() << endl;//获取cpu核心数，失败返回0

t.join(); //线程阻塞

/\*

运行结果：

t.get\_id() = 2

main id = 1

cup num = 4

func id = 2

\*/

return 0;

}

## 10.2 互斥量

### 10.2.1 为什么需要互斥量

在多任务操作系统中，同时运行的多个任务可能都需要使用同一种资源。这个过程有点类似于，公司部门里，我在使用着打印机打印东西的同时（还没有打印完），别人刚好也在此刻使用打印机打印东西，如果不做任何处理的话，打印出来的东西肯定是错乱的。

// 打印机

void printer(const char \*str)

{

while(\*str != '\0')

{

cout << \*str;

str++;

this\_thread::sleep\_for (chrono::seconds(1));

}

cout << endl;

}

// 线程一

void func1()

{

const char \*str = "hello";

printer(str);

}

// 线程二

void func2()

{

const char \*str = "world";

printer(str);

}

int main(void)

{

thread t1(func1);

thread t2(func2);

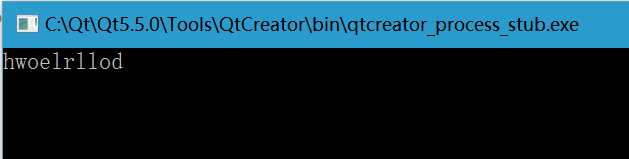
t1.join();

t2.join();

return 0;

}

运行结果如下：



### 10.2.2 独占互斥量std::mutex

互斥量的基本接口很相似，一般用法是通过lock()方法来阻塞线程，直到获得互斥量的所有权为止。在线程获得互斥量并完成任务之后，就必须使用unlock()来解除对互斥量的占用，lock()和unlock()必须成对出现。try\_lock()尝试锁定互斥量，如果成功则返回true, 如果失败则返回false，它是非阻塞的。

mutex g\_lock; //全局互斥锁对象，#include <mutex>

// 打印机

void printer(const char \*str)

{

g\_lock.lock(); //上锁

while(\*str != '\0')

{

cout << \*str;

str++;

this\_thread::sleep\_for (chrono::seconds(1));

}

cout << endl;

g\_lock.unlock(); //解锁

}

// 线程一

void func1()

{

const char \*str = "hello";

printer(str);

}

// 线程二

void func2()

{

const char \*str = "world";

printer(str);

}

int main(void)

{

thread t1(func1);

thread t2(func2);

t1.join();

t2.join();

return 0;

}

使用std::lock\_guard可以简化lock/unlock的写法，同时也更安全，因为lock\_guard在构造时会自动锁定互斥量，而在退出作用域后进行析构时就会自动解锁，从而避免忘了unlock操作。

mutex g\_lock; //全局互斥锁对象，#include <mutex>

// 打印机

void printer(const char \*str)

{

lock\_guard<std::mutex> locker(g\_lock);

while(\*str != '\0')

{

cout << \*str;

str++;

this\_thread::sleep\_for (chrono::seconds(1));

}

cout << endl;

}

### 10.2.3 原子操作

所谓的原子操作，取的就是“原子是最小的、不可分割的最小个体”的意义，它表示在多个线程访问同一个全局资源的时候，能够确保所有其他的线程都不在同一时间内访问相同的资源。也就是他确保了在同一时刻只有唯一的线程对这个资源进行访问。这有点类似互斥对象对共享资源的访问的保护，但是原子操作更加接近底层，因而效率更高。

//全局的结果数据

long total = 0;

//点击函数

void func()

{

for(int i = 0; i < 1000000; ++i)

{

// 对全局数据进行无锁访问

total += 1;

}

}

int main()

{

clock\_t start = clock(); // 计时开始

//线程

thread t1(func);

thread t2(func);

t1.join();

t2.join();

clock\_t end = clock(); // 计时结束

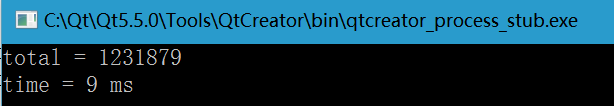
cout << "total = " << total << endl;

cout << "time = " << end-start << " ms\n";

return 0;

}

运行结果如下：



由于线程间对数据的竞争而导致每次运行的结果都不一样。因此，为了防止数据竞争问题，我们需要对total进行原子操作。

通过互斥锁进行原子操作：

//全局的结果数据

long total = 0;

mutex g\_lock;

//点击函数

void func()

{

for(int i = 0; i < 1000000; ++i)

{

g\_lock.lock(); //加锁

total += 1;

g\_lock.unlock(); //解锁

}

}

int main()

{

clock\_t start = clock(); // 计时开始

//线程

thread t1(func);

thread t2(func);

t1.join();

t2.join();

clock\_t end = clock(); // 计时结束

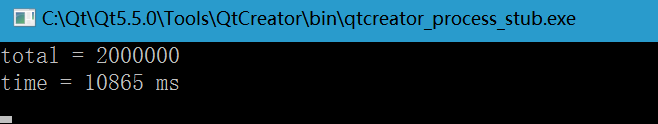
cout << "total = " << total << endl;

cout << "time = " << end-start << " ms\n";

return 0;

}

每次运行的结果都一样，只是耗时长点。



在新标准C++11，引入了原子操作的概念。如果我们在多个线程中对这些类型的共享资源进行操作，编译器将保证这些操作都是原子性的，也就是说，确保任意时刻只有一个线程对这个资源进行访问，编译器将保证多个线程访问这个共享资源的正确性。从而避免了锁的使用，提高了效率。

//原子数据类型

atomic<long> total = {0}; //需要头文件 #include <atomic>

//点击函数

void func()

{

for(int i = 0; i < 1000000; ++i)

{

total += 1;

}

}

int main()

{

clock\_t start = clock(); // 计时开始

//线程

thread t1(func);

thread t2(func);

t1.join();

t2.join();

clock\_t end = clock(); // 计时结束

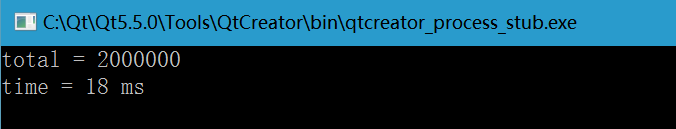
cout << "total = " << total << endl;

cout << "time = " << end-start << " ms\n";

return 0;

}

运行结果如下，耗时也很短：



原子操作的实现跟普通数据类型类似，但是它能够在保证结果正确的前提下，提供比mutex等锁机制更好的性能。