Lesson07---C++11

[本节目标]

- **1. C++11简介**
- **2. 统一的初始化**
- **3. 变量类型推导**
- **4. 范围for循环**
- **5. final与override**
- **6. 委派构造函数**
- **7. 右值引用**
- **8. 默认函数控制**
- **9. lambda表达式**
- **10. 线程库**

1. C++11简介

> 在2003年C++标准委员会曾经提交了一份技术勘误表(简称TC1)，使得C++03这个名字已经取代了C++98称为C++11之前的最新C++标准名称。不过由于TC1主要是对C++98标准中的漏洞进行修复，语言的核心部分则没有改动，因此人们习惯性的把两个标准合并称为C++98/03标准。从C++0x到C++11，C++标准10年磨一剑，第二个真正意义上的标准珊珊来迟。**相比于C++98/03，C++11则带来了数量可观的变化，其中包含了约140个新特性，以及对C++03标准中约600个缺陷的修正，这使得C++11更像是从C++98/03中孕育出的一种新语言。**相比较而言，**C++11能更好地用于系统开发和库开发、语法更加泛华和简单化、更加稳定和安全，不仅功能更强大，而且能提升程序员的开发效率。**

2. 列表初始化

2.1 C++98中{}的初始化问题

> 在C++98中，标准允许使用花括号{}对数组元素进行统一的列表初始值设定。比如：
>
> ```
> int array1[] = {1,2,3,4,5};
> int array2[5] = {0};
> ```
>
> 对于一些自定义的类型，却无法使用这样的初始化。比如：

```
vector<int> v{1,2,3,4,5};
```

就无法通过编译，导致每次定义vector时，都需要先把vector定义出来，然后使用循环对其赋初始值，非常不方便。**C++11扩大了用大括号括起的列表(初始化列表)的使用范围，使其可用于所有的内置类型和用户自定义的类型，使用初始化列表时，可添加等号(=)，也可不添加。**

2.2 内置类型的列表初始化

```
x int main()
{
    // 内置类型变量
    int x1 = {10};
    int x2{10};
    int x3 = 1+2;
    int x4 = {1+2};
    int x5{1+2};
    // 数组
    int arr1[5] {1,2,3,4,5};
    int arr2[]{1,2,3,4,5};

    // 动态数组，在C++98中不支持
    int* arr3 = new int[5]{1,2,3,4,5};

    // 标准容器
    vector<int> v{1,2,3,4,5};
    map<int, int> m{{1,1}, {2,2,},{3,3},{4,4}};
    return 0;
}
```

注意：列表初始化可以在{}之前使用等号，其效果与不使用=没有什么区别。

2.3 自定义类型的列表初始化

1. 标准库支持单个对象的列表初始化

```
class Point
{
public:
    Point(int x = 0, int y = 0)
        : _x(x)
        , _y(y)
    {}

private:
    int _x;
    int _y;
};

int main()
{

    Pointer p{ 1, 2 };
```

```
    return 0;
}
```

2. 多个对象的列表初始化

多个对象想要支持列表初始化，**需给该类(模板类)添加一个带有initializer_list类型参数的构造函数即可。**注意：initializer_list是系统自定义的类模板，该类模板中主要有三个方法：**begin()、end()迭代器以及获取区间中元素个数的方法size()。**

```
#include <initializer_list>
template<class T>
class Vector {
public:
    // ...
    Vector(initializer_list<T> l)
        : _capacity(l.size())
        , _size(0) {
            _array = new T[_capacity];
            for(auto e : l)
                _array[_size++] = e;
    }

    Vector<T>& operator=(initializer_list<T> l) {
        size_t i = 0;
        for (auto e : l)
                _array[i++] = e;
        return *this;
    }
    // ...
private:
    T* _array;
    size_t _capacity;
    size_t _size;
};
```


3. 变量类型推导

3.1 为什么需要类型推导

在定义变量时，必须先给出变量的实际类型，编译器才允许定义，但有些情况下可能不知道需要实际类型怎么给，或者类型写起来特别复杂，比如：

```
#include <map>
#include <string>
int main()
{
    short a = 32670;
    short b = 32670;

    // c如果给成short，会造成数据丢失，如果能够让编译器根据a+b的结果推导c的实际类型，就不会存在问题

    short c = a + b;
```

```
    std::map<std::string, std::string> m{{"apple", "苹果"}, {"banana","香蕉"}};

    // 使用迭代器遍历容器，迭代器类型太繁琐
    std::map<std::string, std::string>::iterator it = m.begin();
    while(it != m.end())
    {
        cout<<it->first<<" "<<it->second<<endl;
        ++it;
    }

    return 0;
}
```

C++11中，可以使用auto来根据变量初始化表达式类型推导变量的实际类型，可以给程序的书写提供许多方便。将程序中c与it的类型换成auto，程序可以通过编译，而且更加简洁。关于auto的详细介绍可以参考C++初阶课件。

3.2 decltype类型推导

3.2.1 为什么需要decltype

auto使用的前提是：必须要对auto声明的类型进行初始化，否则编译器无法推导出auto的实际类型。但有时候可能需要根据表达式运行完成之后结果的类型进行推导，因为编译期间，代码不会运行，此时auto也就无能为力。

```
template<class T1, class T2>
T1 Add(const T1& left, const T2& right)
{
    return left + right;
}
```

如果能用**加完之后结果的实际类型作为函数的返回值类型就不会出错**，但这需要程序运行完才能知道结果的实际类型，即**RTTI(Run-Time Type Identification 运行时类型识别)。**

C++98中确实已经支持RTTI：

- **typeid只能查看类型不能用其结果类定义类型**
- **dynamic_cast只能应用于含有虚函数的继承体系中**

运行时类型识别的缺陷是降低程序运行的效率。

3.2.2 decltype

decltype是根据表达式的实际类型推演出定义变量时所用的类型，比如：

1. 推演表达式类型作为变量的定义类型

```
int main()
{
    int a = 10;
    int b = 20;

    // 用decltype推演a+b的实际类型，作为定义c的类型
    decltype(a+b) c;
    cout<<typeid(c).name()<<endl;
    return 0;
}
```

2. 推演函数返回值的类型

```
void* GetMemory(size_t size)
{
    return malloc(size);
}

int main()
{
    // 如果没有带参数，推导函数的类型
    cout << typeid(decltype(GetMemory)).name() << endl;

    // 如果带参数列表，推导的是函数返回值的类型,注意：此处只是推演，不会执行函数
    cout << typeid(decltype(GetMemory(0))).name() <<endl;

    return 0;
}
```

4 基于范围for的循环

> 此处不进行讲解，请参考C++初阶课件。

5 final与override

> 此处不进行讲解，请参考C++初阶课件

6 委派构造函数

6.1 构造函数冗余造成重复

> 委派构造函数也是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。通过委派其他构造函数，多构造函数的类编写更加容易。**

```
class Info {
public:
    Info()
        : _type(0)
        , _name('a') {
        InitRSet();

    }
```

```
    Info(int type)
        : _type(type)
        , _name('a'){
        InitRSet();
    }

    Info(char a)
        : _type(0)
        , _name(a){
        InitRSet();
    }

private:
    void InitRSet() {//初始化其他变量}
private:
    int _type;
    char _name;
    //...
};
```

上述**构造函数除了初始化列表不同之外，其他部分都是类似的，代码重复。**

初始化列表可以通过：类内部成员初始化进行优化，但是构造函数体的重复在C++98中无法解决。

能否将：构造函数体中重复的代码提出来，作为一个基础版本，在其他构造函数中调用呢？

6.2 委派构造函数

所谓委派构造函数：就是指委派函数将构造的任务委派给目标构造函数来完成的一种类构造的方式。

```
class Info{
public:
    // 目标构造函数
    Info()
        : _type(0)
        , _a('a') {
        InitRSet();
    }

    // 委派构造函数
    Info(int type)
        : Info() {
        _type = type;
    }

    // 委派构造函数
    Info(char a)
        : Info() {
        _a = a;
    }


private:
```

```
    void InitRSet(){
        //初始化其他变量
    }
private:
    int _type =  0;
    char _a = 'a';
    //...
};
```

在初始化列表中调用"基准版本"的构造函数称为委派构造函数，而被调用的"基准版本"则称为目标构造函数。

注意：构造函数不能同时"委派"和使用初始化列表。

7. 默认函数控制

在C++中对于**空类编译器会生成一些默认的成员函数**，比如：**构造函数、拷贝构造函数、运算符重载、析构函数和&和const&的重载、移动构造、移动拷贝构造等函数。如果在类中显式定义了，编译器将不会重新生成默认版本。**有时候这样的规则可能被忘记，最常见的是声明了带参数的构造函数，必要时则需要定义不带参数的版本以实例化无参的对象。而且**有时编译器会生成，有时又不生成，容易造成混乱，于是C++11让程序员可以控制是否需要编译器生成。**

7.1 显式缺省函数

在C++11中，可以在默认函数定义或者声明时加上=default，从而显式的指示编译器生成该函数的默认版本，用=default修饰的函数称为显式缺省函数。

```
class A
{
public:
    A(int a)
        : _a(a)
    {}

    // 显式缺省构造函数，由编译器生成
    A() = default;

    // 在类中声明，在类外定义时让编译器生成默认赋值运算符重载
    A& operator=(const A& a);
private:
    int _a;
};

A& A::operator=(const A& a) = default;

int main()
{
    A a1(10);
    A a2;
    a2 = a1;
    return 0;
}
```

7.2 删除默认函数

> 如果能想要限制某些默认函数的生成，**在C++98中，是该函数设置成private，并且不给定义**，这样只要其他人想要调用就会报错。**在C++11中更简单，只需在该函数声明加上=delete即可，该语法指示编译器不生成对应函数的默认版本，称=delete修饰的函数为删除函数。**

```
class A
{
public:
    A(int a)
        : _a(a)
    {}

    // 禁止编译器生成默认的拷贝构造函数以及赋值运算符重载
    A(const A&) = delete;
    A& operator(const A&) = delete;

private:
    int _a;
};

int main()
{
    A a1(10);

    // 编译失败，因为该类没有拷贝构造函数
    //A a2(a1);

    // 编译失败，因为该类没有赋值运算符重载
    A a3(20);
    a3 = a2;
    return 0;
}
```

> **注意：避免删除函数和explicit一起使**

8 右值引用

8.1 移动语义

> 如果一个类中涉及到资源管理，用户必须显式提供拷贝构造、赋值运算符重载以及析构函数，否则编译器将会自动生成一个默认的，如果遇到拷贝对象或者对象之间相互赋值，就会出错，比如：

```
class String
{
public:
    String(char* str = "")
    {
        if (nullptr == str)
            str = "";
        _str = new char[strlen(str) + 1];

        strcpy(_str, str);
```

```
    }

    String(const String& s)
        : _str(new char[strlen(s._str) + 1])
    {
            strcpy(_str, s._str);
    }

    String& operator=(const String& s)
    {
        if (this != &s)
        {
            char* pTemp = new char[strlen(s._str) +1];
            strcpy(pTemp, s._str);
            delete[] _str;
            _str = pTemp;
        }

        return *this;
    }

    ~String()
    {
        if (_str)
            delete[] _str;
    }

private:
    char* _str;
};
```

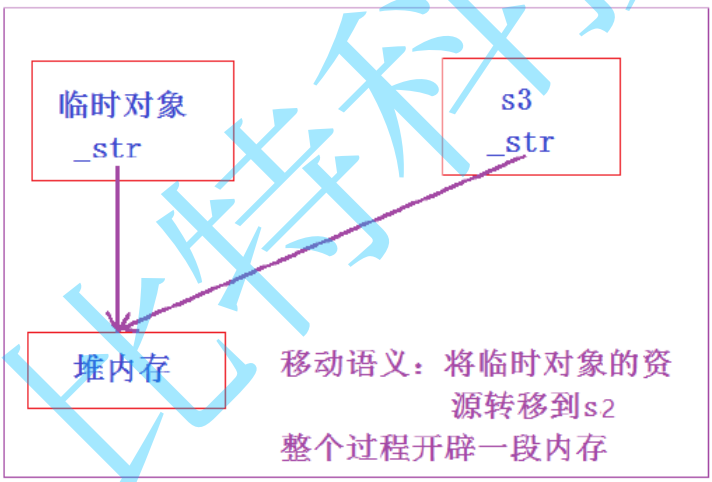
假设现在有一个函数，返回值为一个String类型的对象：

```
String GetString(char* pStr)
{
    String strTemp(pStr);
    return strTemp;
}

int main()
{
    String s1("hello");
    String s2(GetString("world"));
    return 0;
}
```

大家分析下以上代码会出现什么问题?

上述代码看起来没有什么问题，但是有一个不太尽人意的地方：**GetString函数返回的临时对象，将s2拷贝构造成功之后，立马被销毁了(临时对象的空间被释放)，再没有其他作用；而s2在拷贝构造时，又需要分配空间，一个刚释放一个又申请，有点多此一举。**那**能否将GetString返回的临时对象的空间直接交给s2呢？这样s2也不需要重新开辟空间了**，代码的效率会明显提高。



将一个对象中资源移动到另一个对象中的方式，称之为移动语义。在C++11中如果需要实现移动语义，必须使用右值引用。

```
String(String&& s)
    : _str(s._str)
{
    s._str = nullptr;
}
```

8.2 C++11中的右值

右值引用，顾名思义就是对右值的引用。C++11中，右值由两个概念组成：**纯右值和将亡值。**

- **纯右值**

纯右值是C++98中右值的概念，用于识别临时变量和一些不跟对象关联的值。比如：**常量、一些运算表达式(1+3)**等

- **将亡值**

  声明周期将要结束的对象。比如：在值返回时的临时对象。

8.3 右值引用

右值引用书写格式：

类型&& 引用变量名字 = 实体;

右值引用最长常见的一个使用地方就是：与移动语义结合，减少无必要资源的开辟来提高代码的运行效率。

```
String&& GetString(char* pStr)
{
    String strTemp(pStr);
    return strTemp;
}

int main()
{
    String s1("hello");
    String s2(GetString("world"));
    return 0;
}
```

右值引用另一个比较常见的地方是：给一个匿名对象取别名，延长匿名对象的声明周期。

```
String GetString(char* pStr)
{
    return String(pStr);
}

int main()
{
    String&& s = GetString("hello");
    return 0;
}
```

注意：

1. 与引用一样，**右值引用在定义时必须初始化。**
2. 通常情况下，**右值引用不能引用左值。**

```
int main()
{
    int a = 10;
    //int&& ra;   // 编译失败，没有进行初始化
    //int&& ra = a;  // 编译失败，a是一个左值

    // ra是匿名常量10的别名
    const int&& ra = 10;
    return 0;
}
```

8.4 std::move()

C++11中，**std::move()函数**位于 头文件中，这个函数名字具有迷惑性，它**并不搬移任何东西，唯一的功能就是将一个左值强制转化为右值引用，通过右值引用使用该值，实现移动语义。** 注意：被转化的左值，其声明周期并没有随着左右值的转化而改变，即std::move转化的左值变量lvalue不会被销毁。

```
// 移动构造函数
class String
{
    //....
    String(String&& s)
        : _str(s._str)
    {
        s._str = nullptr;
    }

    // ....
};

int main()
{
    String s1("hello world");
    String s2(move(s1));
    String s3(s2);
    return 0;
}
```


以上代码中，s2的构造方式与s3的构造方式相同吗？ 上述代码有什么问题吗？

注意：**上述代码是move()误用的一个非常典型的例子，move更多的是用在声明周期即将结束的对象上。**

```
class Person
{
public:
    Person(char* name, char* sex, int age)
        : _name(name)
        , _sex(sex)
        , _age(age)
    {}
```

```
    Person(const Person& p)
        : _name(p._name)
        , _sex(p._sex)
        , _age(p._age)
    {}

#if 0

    Person(Person&& p)
        : _name(p._name)
        , _sex(p._sex)
        , _age(p._age)
    {}

#else

    Person(Person&& p)
        : _name(move(p._name))
        , _sex(move(p._sex))
        , _age(p._age)
    {}

#endif

private:
    String _name;
    String _sex;
    int _age;
};

Person GetTempPerson()
{
    Person p("prety", "male", 18);
    return p;
}

int main()
{
    Person p(GetTempPerson());
    return 0;
}
```


上述代码中的条件编译打开和不打开有什么区别?

注意：**为了保证移动语义的传递，程序员在编写移动构造函数时，最好使用std::move转移拥有资源的 成员为右值**

8.5 移动语义中的一些问题

注意:

1. 如果将移动构造函数声明为常右值引用或者返回右值的函数声明为常量，都会导致移动语义无法实现。

```
String(const String&&);
const Person GetTempPerson();
```

2. 在C++11中，无参构造函数/拷贝构造函数/移动构造函数实际上有3个版本

```
Object()
Object(const T&)
Object(T &&)
```

3. C++11中默认成员函数

默认情况下，编译器会为程序员隐式生成一个(如果没有用到则不会生成)移动构造函数。如果程序员声明了自定义的构造函数、移动构造、拷贝构造函数、赋值运算符重载、移动赋值、析构函数，编译器都不会再为程序员生成默认版本。**编译器生成的默认移动构造函数实际和默认的拷贝构造函数类似，都是按照位拷贝(即浅拷贝)来进行的。**因此，**在类中涉及到资源管理时，程序员最好自己定义移动构造函数。**其他类有无移动构造都无关紧要。注意：**在C++11中，拷贝构造/移动构造/赋值/移动赋值函数必须同时提供，或者同时不提供，程序才能保证类同时具有拷贝和移动语义。**

8.6 完美转发

完美转发是指在函数模板中，完全依照模板的参数的类型，将参数传递给函数模板中调用的另外一个函数。

```
void Func(int x)
{
    // ......
}

template<typename T>
void PerfectForward(T t)
{
    Fun(t);
}
```

PerfectForward为转发的模板函数，Func为实际目标函数，但是上述转发还不算完美，**完美转发是目标函数总希望将参数按照传递给转发函数的实际类型转给目标函数，而不产生额外的开销**，就好像转发者不存在一样。

所谓完美：**函数模板在向其他函数传递自身形参时，如果相应实参是左值，它就应该被转发为左值；如果相应实参是右值，它就应该被转发为右值。**这样做是为了保留在其他函数针对转发而来的参数的左右值属性进行不同处理（比如参数为左值时实施拷贝语义；参数为右值时实施移动语义）。

C++11通过forward函数来实现完美转发，比如：

```
void Fun(int &x){cout << "lvalue ref" << endl;}
void Fun(int &&x){cout << "rvalue ref" << endl;}
void Fun(const int &x){cout << "const lvalue ref" << endl;}
void Fun(const int &&x){cout << "const rvalue ref" << endl;}

template<typename T>

void PerfectForward(T &&t){Fun(std::forward<T>(t));}
```

```
int main()
{
    PerfectForward(10); // rvalue ref

    int a;
    PerfectForward(a); // lvalue ref
    PerfectForward(std::move(a)); // rvalue ref

    const int b = 8;
    PerfectForward(b); // const lvalue ref
    PerfectForward(std::move(b)); // const rvalue ref

    return 0;
}
```

9 lambda表达式

9.1 C++98中的一个例子

在C++98中，如果想要对一个数据集合中的元素进行排序，可以使用std::sort方法。

```
#include <algorithm>
#include <functional>

int main()
{
    int array[] = {4,1,8,5,3,7,0,9,2,6};

    // 默认按照小于比较，排出来结果是升序
    std::sort(array, array+sizeof(array)/sizeof(array[0]));

    // 如果需要降序，需要改变元素的比较规则
    std::sort(array, array + sizeof(array) / sizeof(array[0]), greater<int>());
    return 0;
}
```

如果待排序元素为自定义类型，需要用户定义排序时的比较规则：

```
struct Goods
{
    string _name;
    double _price;
};

struct Compare
{
    bool operator()(const Goods& gl, const Goods& gr)
    {
        return gl._price <= gr._price;
    }
};
```

```
int main()
{
    Goods gds[] = { { "苹果", 2.1 }, { "相交", 3 }, { "橙子", 2.2 }, {"菠萝", 1.5} };
    sort(gds, gds+sizeof(gds) / sizeof(gds[0]), Compare());
    return 0;
}
```

随着C++语法的发展，**人们开始觉得上面的写法太复杂了，每次为了实现一个algorithm算法， 都要重新去写一个类，如果每次比较的逻辑不一样，还要去实现多个类，特别是相同类的命名，这些都给编程者带来了极大的不便**。因此，在C11语法中出现了Lambda表达式。

9.2 lambda表达式

```
int main()
{
    Goods gds[] = { { "苹果", 2.1 }, { "相交", 3 }, { "橙子", 2.2 }, {"菠萝", 1.5} };
    sort(gds, gds + sizeof(gds) / sizeof(gds[0]), [](const Goods& l, const Goods& r)
                                                  {
                                                      return l._price < r._price;
                                                  });
    return 0;
}
```

上述代码就是使用C++11中的lambda表达式来解决，可以看出lamb表达式实际是一个匿名函数。

9.3 lambda表达式语法

lambda表达式书写格式：**[capture-list] (parameters) mutable -> return-type { statement }**

1. lambda表达式各部分说明
   - [capture-list]：**捕捉列表**，该列表总是出现在lambda函数的开始位置，**编译器根据[]来判断接下来的代码是否为lambda函数，捕捉列表能够捕捉上下文中的变量供lambda函数使用。**
   - (parameters)：参数列表。与**普通函数的参数列表一致**，如果不需要参数传递，则可以连同()一起省略
   - mutable：默认情况下，lambda函数总是一个const函数，mutable可以取消其常量性。使用该修饰符时，参数列表不可省略(即使参数为空)。
   - **->return-type：返回值类型。用追踪返回类型形式声明函数的返回值类型**，没有返回值时此部分可省略。**返回值类型明确情况下，也可省略，由编译器对返回类型进行推导。**
   - **{statement}：函数体。**在该函数体内，除了可以使用其参数外，还可以使用所有捕获到的变量。

   注意： 在lambda函数定义中，**参数列表和返回值类型都是可选部分，而捕捉列表和函数体可以为空。**因此C++11中**最简单的lambda函数为：[]{};** 该lambda函数不能做任何事情。

```
int main()
{
    // 最简单的lambda表达式，该lambda表达式没有任何意义
    []{};

    // 省略参数列表和返回值类型，返回值类型由编译器推导为int

    int a = 3, b = 4;
```

```
    [=]{return a + 3; };

    // 省略了返回值类型，无返回值类型
    auto fun1 = [&](int c){b = a + c; };
    fun1(10)
    cout<<a<<" "<<b<<endl;

    // 各部分都很完善的lambda函数
    auto fun2 = [=, &b](int c)->int{return b += a+ c; };
    cout<<fun2(10)<<endl;

    // 复制捕捉x
    int x = 10;
    auto add_x = [x](int a) mutable { x *= 2; return a + x; };
    cout << add_x(10) << endl;

    return 0;
}
```

通过上述例子可以看出，lambda表达式实际上可以理解为无名函数，该函数无法直接调用，如果想要直接调用，可借助auto将其赋值给一个变量。

2. 捕获列表说明

捕捉列表描述了上下文中那些数据可以被lambda使用，以及使用的方式传值还是传引用。

- [var]：表示值传递方式捕捉变量var
- [=]：表示值传递方式捕获所有父作用域中的变量(包括this)
- [&var]：表示引用传递捕捉变量var
- [&]：表示引用传递捕捉所有父作用域中的变量(包括this)
- [this]：表示值传递方式捕捉当前的this指针

注意：

a. **父作用域指包含lambda函数的语句块**

b. **语法上捕捉列表可由多个捕捉项组成，并以逗号分割。**

比如：[=, &a, &b]：以引用传递的方式捕捉变量a和b，值传递方式捕捉其他所有变量 [&，a, this]：值传递方式捕捉变量a和this，引用方式捕捉其他变量 c. **捕捉列表不允许变量重复传递，否则就会导致编译错误。** 比如：[=, a]：=已经以值传递方式捕捉了所有变量，捕捉a重复

d. **在块作用域以外的lambda函数捕捉列表必须为空。**

e. 在块作用域中的lambda函数仅能捕捉父作用域中局部变量，捕捉任何非此作用域或者非局部变量都会导致编译报错。

f. **lambda表达式之间不能相互赋值**，即使看起来类型相同

```
void (*PF)();
int main()
{

    auto f1 = []{cout << "hello world" << endl; };
    auto f2 = []{cout << "hello world" << endl; };
```

```
    // 此处先不解释原因，等lambda表达式底层实现原理看完后，大家就清楚了
    //f1 = f2;    // 编译失败--->提示找不到operator=()
    // 允许使用一个lambda表达式拷贝构造一个新的副本
    auto f3(f2);
    f3();

    // 可以将lambda表达式赋值给相同类型的函数指针
    PF = f2;
    PF();
    return 0;
}
```

9.4 函数对象与lambda表达式

函数对象，又称为仿函数，即可以想函数一样使用的对象，就是在类中重载了operator()运算符的类对象。

```
class Rate
{
public:
    Rate(double rate)
        : _rate(rate)
    {}

    double operator()(double money, int year)
    {
        return money * _rate * year;
    }

private:
    double _rate;
};

int main()
{
    // 函数对象
    double rate = 0.49;
    Rate r1(rate);
    r1(10000, 2);

    // 仿函数
    auto r2 = [=](double monty, int year)->double{return monty*rate*year; };
    r2(10000, 2);
    return 0;
}
```


从使用方式上来看，函数对象与lambda表达式完全一样。

函数对象将rate作为其成员变量，在定义对象时给出初始值即可，lambda表达式通过捕获列表可以直接将该变量捕获到。

```
Rate r1(rate);
sub   esp,8
movsd xmm0,mmword ptr [rate]
movsd mmword ptr [esp],xmm0
lea   ecx,[r1]
call  Rate::Rate (0D414F1h)

r1(10000, 2);
push  2
sub   esp,8
movsd xmm0,mmword ptr ds:[0D4EDC8h]
movsd mmword ptr [esp],xmm0
lea   ecx,[r1]
call  Rate::operator() (0D414F6h)
fstp  st(0)
```

函数对象底层代码

```
auto r2 = [=](double monty, int year)->double{return mont
lea   eax,[rate]
push  eax
lea   ecx,[r2]
call  <lambda_beb564a084f75c98b17a3763eb2a66ed>::
      <lambda_beb564a084f75c98b17a3763eb2a66ed>(0D433B0h)
r2(10000, 2);
push  2
sub   esp,8
movsd xmm0,mmword ptr ds:[0D4EDC8h]
movsd mmword ptr [esp],xmm0
lea   ecx,[r2]
call  <lambda_beb564a084f75c98b17a3763eb2a66ed>::
fstp  st(0)     operator() (0D43D00h)
```

lambda表达式底层代码

实际在底层编译器对于lambda表达式的处理方式，完全就是按照函数对象的方式处理的，即：如果定义了一个lambda表达式，编译器会自动生成一个类，在该类中重载了operator()。

10 线程库

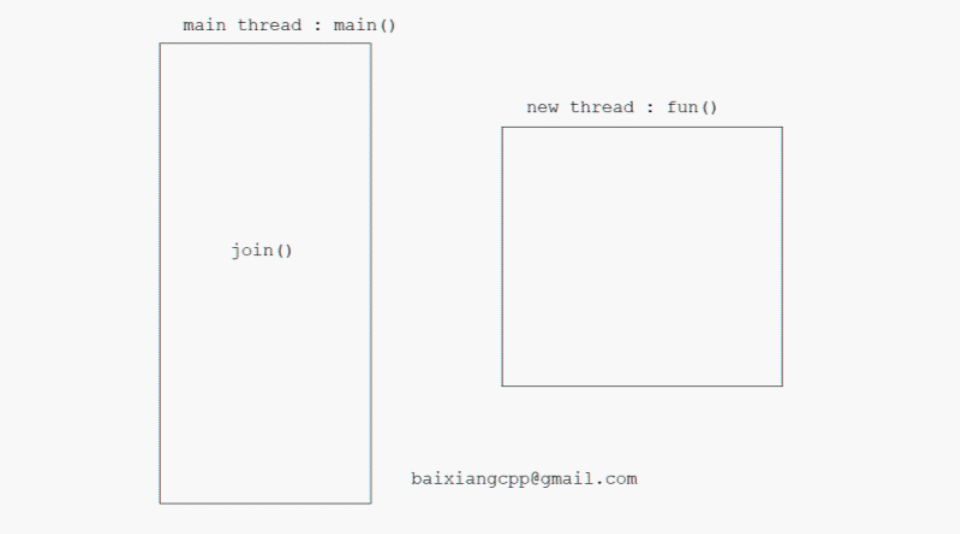
10.1 thread类的简单介绍

C++11中最重要的特性就是对线程进行支持了，使得C++在并行编程时不需要依赖第三方库，而且在原子操作中还引入了原子类的概念。要使用标准库中的线程，必须包含< thread >头文件，该头文件声明了std::thread 线程类。C++11中线程类

```
#include <iostream>
using namespace std;
#include <thread>

void fun()
{
    cout << "A new thread!" << endl;
}

int main()
{
    thread t(fun);
    t.join();
    cout << "Main thread!" << endl;
    return 0;
}
```



10.2 线程的启动

C++线程库通过**构造一个线程对象来启动一个线程，该线程对象中就包含了线程运行时的上下文环境**，比如：**线程函数、线程栈、线程起始状态等以及线程ID等，所有操作全部封装在一起，最后在底层统一传递给_beginthreadex() 创建线程函数来实现** (注意：_beginthreadex是windows中创建线程的底层c函数)。std::thread()创建一个新的线程可以接受任意的可调用对象类型（带参数或者不带参数），包括lambda表达式（带变量捕获或者不带），函数，函数对象，以及函数指针。

```
// 使用lambda表达式作为线程函数创建线程
int main()
{
    int n1 = 500;
    int n2 = 600;
    thread t([&](int addNum){
                            n1 += addNum;
                            n2 += addNum;
                            }, 500);

    t.join();
    std::cout << n1 << ' ' << n2 << std::endl;
    return 0;
}
```

10.3 线程的结束

启动了一个线程后，当这个线程结束的时候，如何去回收线程所使用的资源呢？thread库给我们两种选择：

- **加入式：join()**

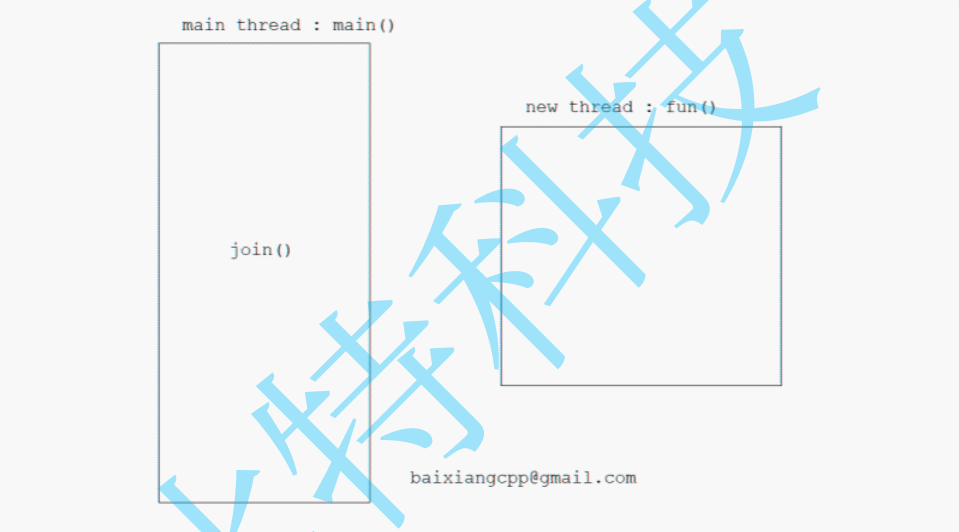
  join()：会主动地等待线程的终止。在调用进程中join()，**当新的线程终止时，join()会清理相关的资源，然后返回，调用线程再继续向下执行。**由于join()清理了线程的相关资源，thread对象与已销毁的线程就没有关系了，因此**一个线程的对象每次你只能使用一次join()**，当你调用的join()之后joinable()就将返回false了。

  ```
  #include <iostream>
  using namespace std;
  #include <thread>
  ```

```
void foo()
{
    this_thread::sleep_for(std::chrono::seconds(1));
}

int main()
{
    thread t(foo);
    cout << "before join, joinable=" << t.joinable() << std::endl;

    t.join();
    cout << "after join, joinable=" << t.joinable()<< endl;
    return 0;
}
```



- **分离式：detach()**

  detach：会从调用线程中分理出新的线程，之后不能再与新线程交互。就像是你和你女朋友分手，那之后你们就不会再有联系（交互）了，而她的之后消费的各种资源也就不需要你去埋单了（清理资源）。此时调用joinable()必然是返回false。**分离的线程会在后台运行，其所有权和控制权将会交给c++运行库。同时，C++运行库保证，当线程退出时，其相关资源的能够正确的回收。**

注意：必须在thread对象销毁之前做出选择，这是因为线程可能在你加入或分离线程之前，就已经结束了，之后如果再去分离它，线程可能会在thread对象销毁之后继续运行下去。

10.4 原子性操作库(atomic)

多线程最主要的问题是共享数据带来的问题(即线程安全)。如果共享数据都是只读的，那么没问题，因为只读操作不会影响到数据，更不会涉及对数据的修改，所以所有线程都会获得同样的数据。但是，**当一个或多个线程要修改共享数据时，就会产生很多潜在的麻烦。**比如：

```
#include <iostream>
using namespace std;
#include <thread>

unsigned long sum = 0L;

void fun(size_t num)
{
    for (size_t i = 0; i < num; ++i)
        sum++;
}

int main()
{
    cout << "Before joining,sum = " << sum << std::endl;

    thread t1(fun, 10000000);
    thread t2(fun, 10000000);
    t1.join();
    t2.join();

    cout << "After joining,sum = " << sum << std::endl;
    return 0;
}
```

C++98中传统的解决方式：可以对共享修改的数据可以加锁保护。

```
#include <iostream>
```

```cpp
using namespace std;
#include <thread>
#include <mutex>

std::mutex m;
unsigned long sum = 0L;

void fun(size_t num)
{
    for (size_t i = 0; i < num; ++i)
    {
        m.lock();
        sum++;
        m.unlock();
    }
}

int main()
{
    cout << "Before joining,sum = " << sum << std::endl;

    thread t1(fun, 10000000);
    thread t2(fun, 10000000);
    t1.join();
    t2.join();

    cout << "After joining,sum = " << sum << std::endl;
    return 0;
}
```

虽然加锁可以解决，但是加锁有一个缺陷就是：只要一个线程在对sum++时，其他线程就会被阻塞，会影响程序运行的效率，而且锁如果控制不好，还容易造成死锁。因此C++11中引入了原子操作。

原子类型名称	对应的内置类型名称
atomic_bool	bool
atomic_char	char
atomic_schar	signed char
atomic_uchar	unsigned_char
atomic_int	int
atomic_uint	unsigned int
atomic_short	short
atomic_ushort	unsigned short
atomic_long	long
atomic_ulong	unsigned long
atomic_llong	long long
atomic_ullong	unsigned long long
atomic_char16_t	char16_t
atomic_char32_t	char32_t
atomic_wchar_t	wchar_t

注意：需要使用以上原子操作变量时，必须添加头文件

```cpp
#include <iostream>
using namespace std;
#include <thread>
#include <atomic>

atomic_long sum{ 0 };

void fun(size_t num)
{
    for (size_t i = 0; i < num; ++i)
        sum ++;   // 原子操作
}

int main()
{
    cout << "Before joining, sum = " << sum << std::endl;

    thread t1(fun, 1000000);
    thread t2(fun, 1000000);
    t1.join();
    t2.join();

    cout << "After joining, sum = " << sum << std::endl;
    return 0;
}
```

http://www.cnblogs.com/ittinybird/p/4830834.html https://blog.csdn.net/column/details/ccia.html?&page=1

https://blog.csdn.net/z_ml118/article/details/78253313#三互斥-mutex