Lesson06---C++11

[本节目标]

- **1. C++11简介**
- **2. 列表初始化**
- **3. 变量类型推导**
- **4. 范围for循环**
- **5. final与override**
- **6. 智能指针**
- **7. 新增加容器---静态数组array、forward_list以及unordered系列**
- **8. 默认成员函数控制**
- **9. 右值引用**
- **10. lambda表达式**
- **11. 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：
>
> ```
> 1  int array1[] = {1,2,3,4,5};
> 2  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

```
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) {
        delete[] _array;
        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

```cpp
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

```cpp
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 范围for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 智能指针

> 此处不进行讲解，请参考C++初阶课件

7. 新增加容器---静态数组array、forward_list以及unordered系列

> 此处不进行讲解，请参考C++初阶课件

8. 默认成员函数控制

> 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。**

8.1 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;
}
```

8.2 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);
```

```
19
20      // 编译失败，因为该类没有赋值运算符重载
21      A a3(20);
22      a3 = a2;
23      return 0;
24  }
```

注意：避免删除函数和explicit一起使

9 右值引用

9.1 右值引用概念

C++98中提出了引用的概念，引用即别名，引用变量与其引用实体公共同一块内存空间，而引用的底层是通过指针来实现的，因此使用引用，可以提高程序的可读性。

```
1   void Swap(int& left, int& right)
2   {
3       int temp = left;
4       left = right;
5       right = temp;
6   }
7
8   int main()
9   {
10      int a = 10;
11      int b = 20;
12      Swap(a, b);
13  }
```

为了提高程序运行效率，C++11中引入了右值引用，右值引用也是别名，但其只能对右值引用。

```
1   int Add(int a, int b)
2   {
3       return a + b;
4   }
5
6   int main()
7   {
8       const int&& ra = 10;
9
10      // 引用函数返回值，返回值是一个临时变量，为右值
11      int&& rRet = Add(10, 20);
12      return 0;
13  }
```

为了与C++98中的引用进行区分，C++11将该种方式称之为右值引用。

9.2 左值与右值

左值与右值是C语言中的概念，但C标准并没有给出严格的区分方式，一般认为：**可以放在=左边的，或者能够取地址的称为左值，只能放在=右边的，或者不能取地址的称为右值**，但是也不一定完全正确。

```cpp
int g_a = 10;
// 函数的返回值结果为引用
int& GetG_A()
{
    return g_a;
}

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

    // a和b都是左值，b既可以在=的左侧，也可在右侧，
    // 说明：左值既可放在=的左侧，也可放在=的右侧
    a = b;
    b = a;

    const int c = 30;
    // 编译失败，c为const常量，只读不允许被修改
    //c = a;
    // 因为可以对c取地址，因此c严格来说不算是左值
    cout << &c << endl;

    // 编译失败：因为b+1的结果是一个临时变量，没有具体名称，也不能取地址，因此为右值
    //b + 1 = 20;

    GetG_A() = 100;
    return 0;
}
```

因此关于左值与右值的区分不是很好区分，一般认为：

1. 普通类型的变量，因为有名字，可以取地址，都认为是左值。
2. const修饰的常量，不可修改，只读类型的，理论应该按照右值对待，但因为其可以取地址(如果只是const类型常量的定义，编译器不给其开辟空间，如果对该常量取地址时，编译器才为其开辟空间)，C++11认为其是左值。
3. 如果表达式的运行结果是一个临时变量或者对象，认为是右值。
4. 如果表达式运行结果或单个变量是一个引用则认为是左值。

总结：

1. 不能简单地通过能否放在=左侧右侧或者取地址来判断左值或者右值，要根据表达式结果或变量的性质判断，比如上述：c常量
2. 能得到引用的表达式一定能够作为引用，否则就用常引用。

C++11对右值进行了严格的区分：

- C语言中的纯右值，比如：a+b, 100
- 将亡值。比如：表达式的中间结果、函数按照值的方式进行返回。

9.3 引用与右值引用比较

在C++98中的普通引用与const引用在引用实体上的区别：

```cpp
int main()
{
    // 普通类型引用只能引用左值，不能引用右值
    int a = 10;
    int& ra1 = a;   // ra为a的别名
    //int& ra2 = 10;   // 编译失败，因为10是右值

    const int& ra3 = 10;
    const int& ra4 = a;
    return 0;
}
```

注意：**普通引用只能引用左值，不能引用右值，const引用既可引用左值，也可引用右值。**

C++11中右值引用：**只能引用右值，一般情况不能直接引用左值。**

```cpp
int main()
{
    // 10纯右值，本来只是一个符号，没有具体的空间，
    // 右值引用变量r1在定义过程中，编译器产生了一个临时变量，r1实际引用的是临时变量
    int&& r1 = 10;
    r1 = 100;

    int a = 10;
    int&& r2 = a;  // 编译失败：右值引用不能引用左值
    return 0;
}
```

问题：既然C++98中的const类型引用左值和右值都可以引用，那为什么C++11还要复杂的提出右值引用呢？

9.4 值的形式返回对象的缺陷

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：

```cpp
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);

    }
```

```
17
18      String& operator=(const String& s)
19      {
20          if (this != &s)
21          {
22              char* pTemp = new char[strlen(s._str) +1];
23              strcpy(pTemp, s._str);
24              delete[] _str;
25              _str = pTemp;
26          }
27          return *this;
28      }
29
30      String operator+(const String& s)
31      {
32          char* pTemp = new char[strlen(_str) + strlen(s._str) + 1];
33          strcpy(pTemp, _str);
34          strcpy(pTemp + strlen(_str), s._str);
35          String strRet(pTemp);
36          return strRet;
37      }
38
39      ~String()
40      { if (_str) delete[] _str;}
41  private:
42      char* _str;
43  };
44
45  int main()
46  {
47      String s1("hello");
48      String s2("world");
49      String s3(s1+s2);
50      return 0;
51  }
```


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：

在operator+中：**strRet在按照值返回时，必须创建一个临时对象，临时对象创建好之后，strRet就被销毁了，最后使用返回的临时对象构造s3，s3构造好之后，临时对象就被销毁了。**仔细观察会发现：**strRet、临时对象、s3每个对象创建后，都有自己独立的空间，而空间中存放内容也都相同，相当于创建了三个内容完全相同的对象，对于空间是一种浪费，程序的效率也会降低，而且临时对象确实作用不是很大**，那能否对该种情况进行优化呢?

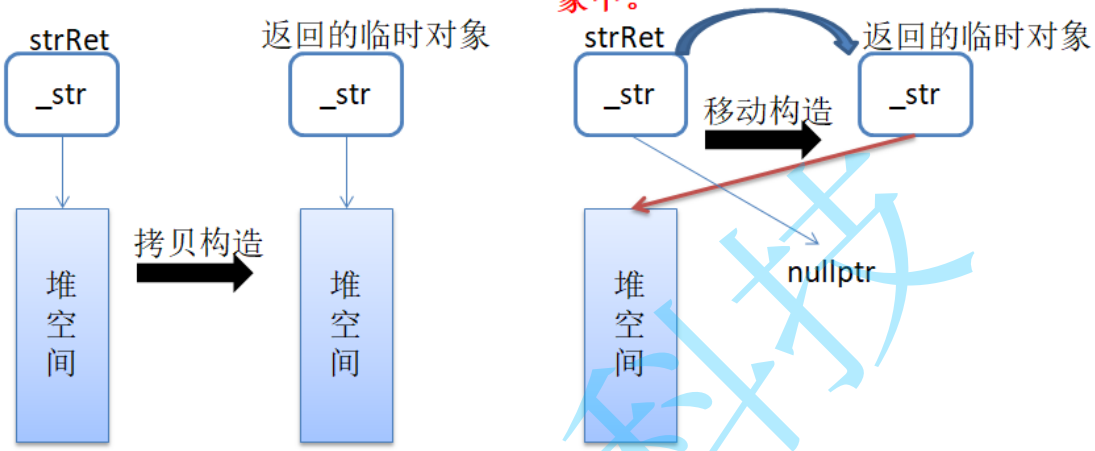
9.5 移动语义

C++11提出了**移动语义**概念，即：**将一个对象中资源移动到另一个对象中的方式**，可以有效缓解该问题。

```
String operator+(const String& s)
{
    char* pTemp = new char[strlen(_str) + strlen(s._str) + 1];
    strcpy(pTemp, _str);
    strcpy(pTemp + strlen(_str), s._str);
    String strRet(pTemp);
    return strRet;
}
```

函数正常返回时，需要用strRet构造一个临时的对象，临时对象和strRet中内容完全相同，出了函数作用域，strRet将被销毁掉。

传统返回缺陷：
strRet拷贝构造临时对象成功后，strRet就被销毁了，该过程中经历了：刚申请一段空间，又释放相同大小一段空间。引入C++11中移动构造进行改进：将strRet中资源转移到临时对象中。



在C++11中如果需要实现移动语义，必须使用右值引用。上述String类增加移动构造：

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

因为strRet对象的生命周期在创建好临时对象后就结束了，即将亡值，C++11认为其为右值，在用strRet构造临时对象时，就会采用移动构造，即将strRet中资源转移到临时对象中。而临时对象也是右值，因此在用临时对象构造s3时，也采用移动构造，将临时对象中资源转移到s3中，整个过程，只需要创建一块堆内存即可，既省了空间，又大大提高程序运行的效率。

注意：

1. 移动构造函数的参数千万不能设置成const类型的右值引用，因为资源无法转移而导致移动语义失效。
2. 在C++11中，编译器会为类默认生成一个移动构造，该移动构造为浅拷贝，因此当类中涉及到资源管理时，用户必须显式定义自己的移动构造。

9.6 右值引用引用左值

按照语法，右值引用只能引用右值，但右值引用一定不能引用左值吗？因为：有些场景下，可能真的需要用右值去引用左值实现移动语义。**当需要用右值引用引用一个左值时，可以通过move函数将左值转化为右值。**C++11中，**std::move()函数**位于 头文件中，该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，然后实现移动语义。**

```
template<class _Ty>
inline typename remove_reference<_Ty>::type&& move(_Ty&& _Arg) _NOEXCEPT
{
    // forward _Arg as movable
    return ((typename remove_reference<_Ty>::type&&)_Arg);
}
```

注意：

1. 被转化的左值，其生命周期并没有随着左值的转化而改变，即std::move转化的左值变量lvalue不会被销毁。
2. STL中也有另一个move函数，就是将一个范围中的元素搬移到另一个位置。

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

注意：以上代码是move函数的经典的误用，因为move将s1转化为右值后，在实现s2的拷贝时就会使用移动构造，此时s1的资源就被转移到s2中，s1就成为了无效的字符串。

使用move的一个例子：

```
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;
}
```

9.7 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;
}
```

9.8 右值引用作用

C++98中引用作用：因为引用是一个别名，需要用指针操作的地方，可以使用指针来代替，可以提高代码的可读性以及安全性。

C++11中右值引用主要有以下作用：

1. 实现移动语义(移动构造与移动赋值)
2. 给中间临时变量取别名：

```
int main()
{
    string s1("hello");
    string s2(" world");
    string s3 = s1 + s2;   // s3是用s1和s2拼接完成之后的结果拷贝构造的新对象
    stirng&& s4 = s1 + s2; // s4就是s1和s2拼接完成之后结果的别名
    return 0;
}
```

3. 实现完美转发

10 lambda表达式

10.1 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>

```

```cpp
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：

```cpp
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。

10.2 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)
                                                  ->bool
                                                   {
                                                       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。

10.3 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)。
   - **->returntype：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;
}

```

10.4 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);

    // lamber
    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。

函数对象底层代码

```
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)
```

lambda表达式底层代码

```
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)
```

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()。

11 线程库

11.1 thread类的简单介绍

在C++11之前，涉及到多线程问题，都是和平台相关的，比如**windows和linux下各有自己的接口，这使得代码的可移植性比较差。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。C++11中线程类

函数名	功能
thread()	构造一个线程对象，没有关联任何线程函数，即没有启动任何线程
thread(fn, args1, args2, ...)	构造一个线程对象，并关联线程函数fn，args1，args2，...为线程函数的参数
get_id()	获取线程id
jionable()	线程是否还在执行，joinable代表的是一个正在执行中的线程。
jion()	该函数调用后会阻塞住线程，当该线程结束后，主线程继续执行
detach()	在创建线程对象后马上调用，用于把被创建线程与线程对象分离开，分离的线程变为后台线程，创建的线程的"死活"就与主线程无关

注意：

1. 线程是操作系统中的一个概念，**线程对象可以关联一个线程，用来控制线程以及获取线程的状态。**
2. 当创建一个线程对象后，没有提供线程函数，该对象实际没有对应任何线程。

```
#include <thread>
int main()
{
    std::thread t1;
    cout << t1.get_id() << endl;
    return 0;
}
```

get_id()的返回值类型为id类型，id类型实际为std::thread命名空间下封装的一个类，该类中包含了一个结构体：

```
// vs下查看
typedef struct
{   /* thread identifier for Win32 */
    void *_Hnd; /* Win32 HANDLE */
    unsigned int _Id;
} _Thrd_imp_t;
```

3. 当创建一个线程对象后，并且给线程关联线程函数，该线程就被启动，与主线程一起运行。线程函数一般情况下可按照以下三种方式提供：
   - 函数指针
   - lambda表达式

- 函数对象

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

void ThreadFunc(int a)
{
    cout << "Thread1" << a << endl;
}


class TF
{
public:
    void operator()()
    {
        cout << "Thread3" << endl;
    }
};

int main()
{
    // 线程函数为函数指针
    thread t1(ThreadFunc, 10);

    // 线程函数为lambda表达式
    thread t2([]{cout << "Thread2" << endl; });

    // 线程函数为函数对象
    TF tf;
    thread t3(tf);

    t1.join();
    t2.join();
    t3.join();
    cout << "Main thread!" << endl;
    return 0;
}
```

4. thread类是防拷贝的，不允许拷贝构造以及赋值，但是可以移动构造和移动赋值，即将一个线程对象关联线程的状态转移给其他线程对象，转移期间不意向线程的执行。
5. 可以通过jionable()函数判断线程是否是有效的，如果是以下任意情况，则线程无效
   - 采用无参构造函数构造的线程对象
   - 线程对象的状态已经转移给其他线程对象
   - 线程已经调用jion或者detach结束

面试题：并发与并行的区别?

11.2 线程函数参数

线程函数的参数是以值拷贝的方式拷贝到线程栈空间中的，因此：即使线程参数为引用类型，在线程中修改后也不能修改外部实参，因为**其实际引用的是线程栈中的拷贝，而不是外部实参。**

```
#include <thread>
void ThreadFunc1(int& x)
{
    x += 10;
}

void ThreadFunc2(int* x)
{
    *x += 10;
}

int main()
{
    int a = 10;

    // 在线程函数中对a修改，不会影响外部实参，因为：线程函数参数虽然是引用方式，但其实际引用的是线程栈中的拷贝
    thread t1(ThreadFunc1, a);
    t1.join();
    cout << a << endl;

    // 如果想要通过形参改变外部实参时，必须借助std::ref()函数
    thread t2(ThreadFunc1, std::ref(a));
    t2.join();
    cout << a << endl;

    // 地址的拷贝
    thread t3(ThreadFunc2, &a);
    t3.join();
    cout << a << endl;
    return 0;
}
```

注意：如果是类成员函数作为线程参数时，必须将this作为线程函数参数。

11.3 join与detach

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：

- **join()方式**

  join()：主线程被阻塞，当新线程终止时，join()会清理相关的线程资源，然后返回，主线程再继续向下执行，然后销毁线程对象。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**一个线程对象只能使用一次join()，否则程序会崩溃。**


```cpp
// jion()的误用一
void ThreadFunc() { cout<<"ThreadFunc()"<<endl; }
bool DoSomething() { return false; }
int main()
{
    std::thread t(ThreadFunc);
    if(!DoSomething())
        return -1;

    t.join();
    return 0;
}
/*
说明：如果DoSomething()函数返回false,主线程将会结束，jion()没有调用，线程资源没有回收，造成资源泄漏。
*/

// jion()的误用二
void ThreadFunc() { cout<<"ThreadFunc()"<<endl; }
void Test1() { throw 1; }
void Test2()
{
    int* p = new int[10];
    std::thread t(ThreadFunc);
    try
    {
        Test1();
    }
    catch(...)
    {
        delete[] p;
        throw;
    }

    t.jion();
}

```

```
// 说明：与上述原因相似
```

因此：采用jion()方式结束线程时，jion()的调用位置非常关键。为了避免该问题，可以**采用RAII的方式对线程对象进行封装**，比如：

```
#include <thread>

class mythread
{
public:
    explicit mythread(std::thread &t) :m_t(t){}
    ~mythread()
    {
        if (m_t.joinable())
            m_t.join();
    }

    mythread(mythread const&)=delete;
    mythread& operator=(const mythread &)=delete;

private:
    std::thread &m_t;
};

void ThreadFunc() { cout << "ThreadFunc()" << endl; }
bool DoSomething() { return false; }

int main()
{
    thread t(ThreadFunc);
    mythread q(t);

    if (DoSomething())
        return -1;

    return 0;
}
```


- **detach()方式**

  detach()：该函数被调用后，新线程与线程对象分离，不再被线程对象所表达，就不能通过线程对象控制线程了，新线程会在后台运行，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。

  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）。

  detach()函数一般在线程对象创建好之后就调用，因为如果不是jion()等待方式结束，那么**线程对象可能会在新线程结束之前被销毁掉而导致程序崩溃。**因为**std::thread的析构函数中，如果线程的状态是jionable，std::terminate将会被调用，而terminate()函数直接会终止程序。**

因此：**线程对象销毁前，要么以jion()的方式等待线程结束，要么以detach()的方式将线程与线程对象分离。**

11.4 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。所谓原子操作：即不可被中断的一个或一系列操作，C++11引入的原子操作类型，使得线程间数据的同步变得非常高效。

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

```
#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;
}
```

在C++11中，**程序员不需要对原子类型变量进行加锁解锁操作，线程能够对原子类型变量互斥的访问。**

更为普遍的，程序员可以**使用atomic类模板，定义出需要的任意原子类型。**

```
atmoic<T> t;    // 声明一个类型为T的原子类型变量t
```

注意：原子类型通常属于"资源型"数据，多个线程只能访问单个原子类型的拷贝，因此**在C++11中，原子类型只能从其模板参数中进行构造，不允许原子类型进行拷贝构造、移动构造以及operator=**等，为了防止意外，标准库已经将atmoic模板类中的拷贝构造、移动构造、赋值运算符重载默认删除掉了。

```
#include <atomic>
int main()
{
    atomic<int> a1(0);
    //atomic<int> a2(a1);    // 编译失败
    atomic<int> a2(0);
    //a2 = a1;               // 编译失败
    return 0;
}
```

原子操作

11.5 lock_guard与unique_lock

在多线程环境下，如果想要保证某个变量的安全性，只要将其设置成对应的原子类型即可，即高效又不容易出现死锁问题。但是有些情况下，我们可能需要保证一段代码的安全性，那么就只能通过锁的方式来进行控制。

比如：一个线程对变量number进行加一100次，另外一个减一100次，每次操作加一或者减一之后，输出number的结果，要求：number最后的值为1。

```
#include <thread>
#include <mutex>

int number = 0;
mutex g_lock;

int ThreadProc1()
{
    for (int i = 0; i < 100; i++)
    {
        g_lock.lock();
        ++number;
        cout << "thread 1 :" << number << endl;
        g_lock.unlock();
    }

    return 0;
}

int ThreadProc2()
{
    for (int i = 0; i < 100; i++)
    {
        g_lock.lock();
        --number;
        cout << "thread 2 :" << number << endl;
```

```
        g_lock.unlock();
    }

    return 0;
}

int main()
{
    thread t1(ThreadProc1);
    thread t2(ThreadProc2);

    t1.join();
    t2.join();

    cout << "number:" << number << endl;
    system("pause");
    return 0;
}
```

上述代码的缺陷：**锁控制不好时，可能会造成死锁**，最常见的**比如在锁中间代码返回，或者在锁的范围内抛异常。**因此：C++11采用RAII的方式对锁进行了封装，即lock_guard和unique_lock。

11.5.1 Mutex的种类

在C++11中，Mutex总共包了四个互斥量的种类：

1. std::mutex

   C++11提供的最基本的互斥量，该类的对象之间不能拷贝，也不能进行移动。mutex最常用的三个函数：

函数名	函数功能
lock()	上锁：锁住互斥量
unlock()	解锁：释放对互斥量的所有权
try_lock()	尝试锁住互斥量，如果互斥量被其他线程占有，则当前线程也不会被阻塞

   注意，线程函数调用lock()时，可能会发生以下三种情况：

   - 如果该互斥量当前没有被锁住，则调用线程将该互斥量锁住，直到调用 unlock之前，该线程一直拥有该锁
   - 如果当前互斥量被其他线程锁住，则当前的调用线程被阻塞住
   - 如果当前互斥量被当前调用线程锁住，则会产生死锁(deadlock)

   线程函数调用try_lock()时，可能会发生以下三种情况：

   - 如果当前互斥量没有被其他线程占有，则该线程锁住互斥量，直到该线程调用 unlock 释放互斥量
   - 如果当前互斥量被其他线程锁住，则当前调用线程返回 false，而并不会被阻塞掉
   - 如果当前互斥量被当前调用线程锁住，则会产生死锁(deadlock)

2. std::recursive_mutex

其**允许同一个线程对互斥量多次上锁（即递归上锁），来获得对互斥量对象的多层所有权，释放互斥量时需要调用与该锁层次深度相同次数的 unlock()**，除此之外，std::recursive_mutex 的特性和 std::mutex 大致相同。

3. std::timed_mutex

   比 std::mutex 多了两个成员函数，try_lock_for()，try_lock_until() 。

   - try_lock_for()

     接受一个时间范围，表示在这一段时间范围之内线程如果没有获得锁则被阻塞住（与 std::mutex 的 try_lock() 不同，try_lock 如果被调用时没有获得锁则直接返回 false），如果在此期间其他线程释放了锁，则该线程可以获得对互斥量的锁，如果超时（即在指定时间内还是没有获得锁），则返回 false。

   - try_lock_until()

     接受一个时间点作为参数，在指定时间点未到来之前线程如果没有获得锁则被阻塞住，如果在此期间其他线程释放了锁，则该线程可以获得对互斥量的锁，如果超时（即在指定时间内还是没有获得锁），则返回 false。

4. std::recursive_timed_mutex

11.5.2 lock_guard

std::lock_gurad 是 C++11 中定义的模板类。定义如下：

```
template<class _Mutex>
class lock_guard
{
public:
    // 在构造lock_gard时，_Mtx还没有被上锁
    explicit lock_guard(_Mutex& _Mtx)
        : _MyMutex(_Mtx)
    {
        _MyMutex.lock();
    }

    // 在构造lock_gard时，_Mtx已经被上锁，此处不需要再上锁
    lock_guard(_Mutex& _Mtx, adopt_lock_t)
        : _MyMutex(_Mtx)
    {}

    ~lock_guard() _NOEXCEPT
    {
        _MyMutex.unlock();
    }

    lock_guard(const lock_guard&) = delete;
    lock_guard& operator=(const lock_guard&) = delete;

private:
    _Mutex& _MyMutex;
};
```

> 通过上述代码可以看到，**lock_guard类模板主要是通过RAII的方式，对其管理的互斥量进行了封装**，在需要加锁的地方，只需要用上述介绍的**任意互斥体实例化一个lock_guard，调用构造函数成功上锁，出作用域前，lock_guard对象要被销毁，调用析构函数自动解锁，可以有效避免死锁问题。**
>
> **lock_guard的缺陷：太单一，用户没有办法对该锁进行控制**，因此C++11又提供了unique_lock。

11.5.3 unique_lock

> 与lock_gard类似，**unique_lock类模板也是采用RAII的方式对锁进行了封装，并且也是以独占所有权的方式管理mutex对象的上锁和解锁操作，即其对象之间不能发生拷贝。**在构造(或移动(move)赋值)时，unique_lock 对象需要传递一个 Mutex 对象作为它的参数，新创建的 unique_lock 对象负责传入的 Mutex 对象的上锁和解锁操作。**使用以上类型互斥量实例化unique_lock的对象时，自动调用构造函数上锁，unique_lock对象销毁时自动调用析构函数解锁，可以很方便的防止死锁问题。**
>
> 与lock_guard不同的是，unique_lock更加的灵活，提供了更多的成员函数：
>
> - **上锁/解锁操作**：lock、try_lock、try_lock_for、try_lock_until和unlock
> - **修改操作**：移动赋值、交换(swap：与另一个unique_lock对象互换所管理的互斥量所有权)、释放(release：返回它所管理的互斥量对象的指针，并释放所有权)
> - **获取属性**：owns_lock(返回当前对象是否上了锁)、operator bool()(与owns_lock()的功能相同)、mutex(返回当前unique_lock所管理的互斥量的指针)。
>
> lock_guard和unique_lock