## [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.包装器

### 12. 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C++03(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，公司实际项目开发中也用得比较多，所以我们要作为一个重点去学习。**C++11增加的语法特性非常篇幅非常多，我们这里没办法一一讲解，所以本节课程主要讲解实际中比较实用的语法。
>
> https://en.cppreference.com/w/cpp/11
>
> **小故事:**
>
> 1998年是C++标准委员会成立的第一年，本来计划以后每5年视实际需要更新一次标准，C++国际标准委员会在研究C++ 03的下一个版本的时候，一开始计划是2007年发布，所以最初这个标准叫C++ 07。但是到06年的时候，官方觉得2007年肯定完不成C++ 07，而且官方觉得2008年可能也完不成。最后干脆叫C++ 0x。x的意思是不知道到底能在07还是08还是09年完成。结果2010年的时候也没完成，最后在2011年终于完成了C++标准。所以最终定名为C++11。

# 2. 统一的列表初始化

## 2.1 {}初始化

在C++98中，标准允许使用花括号{}对数组或者结构体元素进行统一的列表初始值设定。比如：

```
struct Point
{
    int _x;
    int _y;
};
int main()
{
    int array1[] = { 1, 2, 3, 4, 5 };
    int array2[5] = { 0 };
    Point p = { 1, 2 };
    return 0;
}
```

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。**

```
struct Point
{
    int _x;
    int _y;
};

int main()
{
    int x1 = 1;
    int x2{ 2 };

    int array1[]{ 1, 2, 3, 4, 5 };
    int array2[5]{ 0 };
    Point p{ 1, 2 };

    // C++11中列表初始化也可以适用于new表达式中
    int* pa = new int[4]{ 0 };

    return 0;
}
```

创建对象时也可以使用列表初始化方式调用构造函数初始化

```
class Date
{
public:
    Date(int year, int month, int day)
        :_year(year)
        ,_month(month)
        ,_day(day)
    {
        cout << "Date(int year, int month, int day)" << endl;
    }

private:
    int _year;
    int _month;
    int _day;
```

```
};

int main()
{
    Date d1(2022, 1, 1); // old style

    // C++11支持的列表初始化，这里会调用构造函数初始化
    Date d2{ 2022, 1, 2 };
    Date d3 = { 2022, 1, 3 };

    return 0;
}
```

## 2.2 std::initializer_list

**std::initializer_list的介绍文档：**

http://www.cplusplus.com/reference/initializer_list/initializer_list/

**std::initializer_list是什么类型：**

```
int main()
{
    // the type of il is an initializer_list
    auto il = { 10, 20, 30 };
    cout << typeid(il).name() << endl;

    return 0;
}
```

**std::initializer_list使用场景：**

std::initializer_list一般是作为构造函数的参数，C++11对STL中的不少容器就增加std::initializer_list作为参数的构造函数，这样初始化容器对象就更方便了。也可以作为operator=的参数，这样就可以用大括号赋值。

http://www.cplusplus.com/reference/list/list/list/

http://www.cplusplus.com/reference/vector/vector/vector/

http://www.cplusplus.com/reference/map/map/map/

http://www.cplusplus.com/reference/vector/vector/operator=/

```
int main()
{
    vector<int> v = { 1,2,3,4 };
    list<int> lt = { 1,2 };

    // 这里{"sort", "排序"}会先初始化构造一个pair对象
    map<string, string> dict = { {"sort", "排序"}, {"insert", "插入"} };

 // 使用大括号对容器赋值
 v = {10, 20, 30};

    return 0;
```

> **让模拟实现的vector也支持{}初始化和赋值**

```
namespace bit
{
 template<class T>
 class vector {
 public:
     typedef T* iterator;

     vector(initializer_list<T> l)
     {
         _start = new T[l.size()];
         _finish = _start + l.size();
         _endofstorage = _start + l.size();

         iterator vit = _start;
         typename initializer_list<T>::iterator lit = l.begin();
         while (lit != l.end())
         {
             *vit++ = *lit++;
         }

         //for (auto e : l)
         //    *vit++ = e;
     }

     vector<T>& operator=(initializer_list<T> l) {
         vector<T> tmp(l);
         std::swap(_start, tmp._start);
         std::swap(_finish, tmp._finish);
         std::swap(_endofstorage, tmp._endofstorage);

         return *this;
     }
 private:
     iterator _start;
     iterator _finish;
     iterator _endofstorage;
 };
}
```

# 3. 声明

c++11提供了多种简化声明的方式，尤其是在使用模板时。

## 3.1 auto

> 在C++98中auto是一个存储类型的说明符，表明变量是局部自动存储类型，但是局部域中定义局部的变量默认就是自动存储类型，所以auto就没什么价值了。**C++11中废弃auto原来的用法，将其用于实现自动类型腿断。这样要求必须进行显示初始化，让编译器将定义对象的类型设置为初始化值的类型。**

```
int main()
{
 int i = 10;
 auto p = &i;
```

```
 auto pf = strcpy;

 cout << typeid(p).name() << endl;
 cout << typeid(pf).name() << endl;

 map<string, string> dict = { {"sort", "排序"}, {"insert", "插入"} };
 //map<string, string>::iterator it = dict.begin();
 auto it = dict.begin();

 return 0;
}
```

### 3.2 decltype

关键字decltype将变量的类型声明为表达式指定的类型。

```
// decltype的一些使用使用场景
template<class T1, class T2>
void F(T1 t1, T2 t2)
{
 decltype(t1 * t2) ret;
 cout << typeid(ret).name() << endl;
}

int main()
{
 const int x = 1;
 double y = 2.2;

 decltype(x * y) ret; // ret的类型是double
 decltype(&x) p;      // p的类型是int*
 cout << typeid(ret).name() << endl;
 cout << typeid(p).name() << endl;

 F(1, 'a');

 return 0;
}
```

### 3.3 nullptr

由于C++中NULL被定义成字面量0，这样就可能回带来一些问题，因为0既能指针常量，又能表示整形常量。所以出于清晰和安全的角度考虑，C++11中新增了nullptr，用于表示空指针。

```
#ifndef NULL
#ifdef __cplusplus
#define NULL    0
#else
#define NULL    ((void *)0)
#endif
#endif
```

## 4 范围for循环

这个我们在前面的课程中已经进行了非常详细的讲解，这里就不进行讲解了，请参考C++入门+STL容器部分的课件讲解。

## 5 智能指针

> 这个我们在智能指针课程中已经会进行了非常详细的讲解，这里就不进行讲解了，请参考智能指针部分课件讲解。

## 6 STL中一些变化

> **新容器**
>
> 用橘色圈起来是C++11中的一些几个新容器，但是实际最有用的是unordered_map和unordered_set。这两个我们前面已经进行了非常详细的讲解，其他的大家了解一下即可。

**Containers**

| | |
|---|---|
| <array> | Array header (header) |
| <bitset> | Bitset header (header) |
| <deque> | Deque header (header) |
| <forward_list> | Forward list (header) |
| <list> | List header (header) |
| <map> | Map header (header) |
| <queue> | Queue header (header) |
| <set> | Set header (header) |
| <stack> | Stack header (header) |
| <unordered_map> | Unordered map header (header) |
| <unordered_set> | Unordered set header (header) |
| <vector> | Vector header (header) |

> **容器中的一些新方法**
>
> 如果我们再细细去看会发现基本每个容器中都增加了一些C++11的方法，但是其实很多都是用得比较少的。
>
> 比如提供了cbegin和cend方法返回const迭代器等等，但是实际意义不大，因为begin和end也是可以返回const迭代器的，这些都是属于锦上添花的操作。
>
> 实际上C++11更新后，容器中增加的新方法最后用的插入接口函数的右值引用版本：
>
> http://www.cplusplus.com/reference/vector/vector/emplace_back/
>
> http://www.cplusplus.com/reference/vector/vector/push_back/
>
> http://www.cplusplus.com/reference/map/map/insert/
>
> http://www.cplusplus.com/reference/map/map/emplace/
>
> 但是这些接口到底意义在哪？网上都说他们能提高效率，他们是如何提高效率的？
>
> 请看下面的右值引用和移动语义章节的讲解。另外emplace还涉及模板的可变参数，也需要再继续深入学习后面章节的知识。

# 7 右值引用和移动语义

## 7.1 左值引用和右值引用

传统的C++语法中就有引用的语法，而C++11中新增了的右值引用语法特性，所以从现在开始我们之前学习的引用就叫做左值引用。**无论左值引用还是右值引用，都是给对象取别名。**

### 什么是左值？什么是左值引用？

左值是一个表示数据的表达式(如变量名或解引用的指针)，**我们可以获取它的地址+可以对它赋值，左值可以出现赋值符号的左边，右值不能出现在赋值符号左边。**定义时const修饰符后的左值，不能给他赋值，但是可以取它的地址。左值引用就是给左值的引用，给左值取别名。

```
int main()
{
// 以下的p、b、c、*p都是左值
int* p = new int(0);
int b = 1;
const int c = 2;

// 以下几个是对上面左值的左值引用
int*& rp = p;
int& rb = b;
const int& rc = c;
int& pvalue = *p;

return 0;
}
```

### 什么是右值？什么是右值引用？

右值也是一个表示数据的表达式，如：字面常量、表达式返回值，函数返回值(这个不能是左值引用返回)等等，**右值可以出现在赋值符号的右边，但是不能出现出现在赋值符号的左边，右值不能取地址。**右值引用就是对右值的引用，给右值取别名。

```
int main()
{
double x = 1.1, y = 2.2;

// 以下几个都是常见的右值
10;
x + y;
fmin(x, y);

// 以下几个都是对右值的右值引用
int&& rr1 = 10;
double&& rr2 = x + y;
double&& rr3 = fmin(x, y);

// 这里编译会报错：error C2106: “=”: 左操作数必须为左值
10 = 1;
x + y = 1;
fmin(x, y) = 1;

return 0;
}
```

需要注意的是右值是不能取地址的，但是给右值取别名后，会导致右值被存储到特定位置，且可以取到该位置的地址，也就是说例如：不能取字面量10的地址，但是rr1引用后，可以对rr1取地址，也可以修改rr1。如果不想rr1被修改，可以用const int&& rr1 去引用，是不是感觉很神奇，这个了解一下实际中右值引用的使用场景并不在于此，这个特性也不重要。

```
int main()
{
    double x = 1.1, y = 2.2;
    int&& rr1 = 10;
    const double&& rr2 = x + y;

    rr1 = 20;
    rr2 = 5.5;  // 报错

    return 0;
}
```

## 7.2 左值引用与右值引用比较

**左值引用总结：**

1. 左值引用只能引用左值，不能引用右值。
2. 但是const左值引用既可引用左值，也可引用右值。

```
int main()
{
    // 左值引用只能引用左值，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左值引用既可引用左值，也可引用右值。
    const int& ra3 = 10;
    const int& ra4 = a;

    return 0;
}
```

**右值引用总结：**

1. 右值引用只能右值，不能引用左值。
2. 但是右值引用可以move以后的左值。

```
int main()
{
    // 右值引用只能右值，不能引用左值。
    int&& r1 = 10;

    // error C2440: “初始化”: 无法从“int”转换为“int &&”
    // message : 无法将左值绑定到右值引用
    int a = 10;
    int&& r2 = a;

    // 右值引用可以引用move以后的左值
```

```
    int&& r3 = std::move(a);

    return 0;
}
```

## 7.3 右值引用使用场景和意义

前面我们可以看到左值引用既可以引用左值和又可以引用右值，那为什么C++11还要提出右值引用呢？是不是化蛇添足呢？下面我们来看看左值引用的短板，右值引用是如何补齐这个短板的！

```
namespace bit
{
    class string
    {
    public:
        typedef char* iterator;
        iterator begin()
        {
            return _str;
        }

        iterator end()
        {
            return _str + _size;
        }

        string(const char* str = "")
            :_size(strlen(str))
            , _capacity(_size)
        {
            //cout << "string(char* str)" << endl;

            _str = new char[_capacity + 1];
            strcpy(_str, str);
        }

        // s1.swap(s2)
        void swap(string& s)
        {
            ::swap(_str, s._str);
            ::swap(_size, s._size);
            ::swap(_capacity, s._capacity);
        }

        // 拷贝构造
        string(const string& s)
            :_str(nullptr)
        {
            cout << "string(const string& s) -- 深拷贝" << endl;

            string tmp(s._str);
            swap(tmp);
        }

        // 赋值重载
        string& operator=(const string& s)
        {
```

```
        cout << "string& operator=(string s) -- 深拷贝" << endl;
        string tmp(s);
        swap(tmp);

        return *this;
    }

    // 移动构造
    string(string&& s)
        :_str(nullptr)
        ,_size(0)
        ,_capacity(0)
    {
        cout << "string(string&& s) -- 移动语义" << endl;
        swap(s);
    }

    // 移动赋值
    string& operator=(string&& s)
    {
        cout << "string& operator=(string&& s) -- 移动语义" << endl;
        swap(s);

        return *this;
    }

    ~string()
    {
        delete[] _str;
        _str = nullptr;
    }

    char& operator[](size_t pos)
    {
        assert(pos < _size);
        return _str[pos];
    }

    void reserve(size_t n)
    {
        if (n > _capacity)
        {
            char* tmp = new char[n + 1];
            strcpy(tmp, _str);
            delete[] _str;
            _str = tmp;

            _capacity = n;
        }
    }

    void push_back(char ch)
    {
        if (_size >= _capacity)
        {
            size_t newcapacity = _capacity == 0 ? 4 : _capacity * 2;
            reserve(newcapacity);
        }
```

```
            _str[_size] = ch;
            ++_size;
            _str[_size] = '\0';
        }

        //string operator+=(char ch)
        string& operator+=(char ch)
        {
            push_back(ch);
            return *this;
        }

        const char* c_str() const
        {
            return _str;
        }
    private:
        char* _str;
        size_t _size;
        size_t _capacity; // 不包含最后做标识的\0
    };
}
```

**左值引用的使用场景：**

做参数和做返回值都可以提高效率。

```
void func1(bit::string s)
{}

void func2(const bit::string& s)
{}

int main()
{
    bit::string s1("hello world");
    // func1和func2的调用我们可以看到左值引用做参数减少了拷贝，提高效率的使用场景和价值
    func1(s1);
    func2(s1);

    // string operator+=(char ch)  传值返回存在深拷贝
    // string& operator+=(char ch) 传左值引用没有拷贝提高了效率
    s1 += '!';

    return 0;
}
```

**左值引用的短板：**

但是当函数返回对象是一个局部变量，出了函数作用域就不存在了，就不能使用左值引用返回，只能传值返回。例如：bit::string to_string(int value)函数中可以看到，这里只能使用传值返回，传值返回会导致至少1次拷贝构造(如果是一些旧一点的编译器可能是两次拷贝构造)。

```
bit::string to_string(int value)
{
    bit::string str;
    //...

    return str;
}

int main()
{
    bit::string ret2 = bit::to_string(-1234);

    return 0;
}
```

拷贝构造

拷贝构造

本来应该是两次拷贝构造，但是新一点的编译器一般都会优化，优化后变成了一次拷贝构造。

```
bit::string to_string(int value)
{
    bit::string str;
    //...

    return str;
}

int main()
{
    bit::string ret2 = bit::to_string(-1234);

    return 0;
}
```

本来应该是两次拷贝构造，但是新一点的编译器一般都会优化，优化后变成了一次拷贝构造。

```
namespace bit
{
    bit::string to_string(int value)
    {
        bool flag = true;
        if (value < 0)
        {
            flag = false;
            value = 0 - value;
        }

        bit::string str;
        while (value > 0)
        {
            int x = value % 10;
            value /= 10;

            str += ('0' + x);
        }

        if (flag == false)
        {
            str += '-';
        }
```

```
        std::reverse(str.begin(), str.end());
        return str;
    }
}

int main()
{
 // 在bit::string to_string(int value)函数中可以看到，这里
 // 只能使用传值返回，传值返回会导致至少1次拷贝构造(如果是一些旧一点的编译器可能是两次拷贝构造)。
    bit::string ret1 = bit::to_string(1234);
    bit::string ret2 = bit::to_string(-1234);

    return 0;
}
```

```
// 拷贝构造
string(const string& s)
    :_str(nullptr)
{
    cout << "string(const string& s) -- 深拷贝" << endl;

    string tmp(s._str);
    swap(tmp);
}

bit::string to_string(int value)
{
    bit::string str;
    //...

    return str;
}

int main()
{
    bit::string ret2 = bit::to_string(-1234);

    return 0;
}
```

to_string的返回值是一个右值，用这个右值构造ret2，如果没有移动构造，调用就会匹配调用拷贝构造，因为const左值引用是可以引用右值的，这里就是一个深拷贝。

**右值引用和移动语义解决上述问题：**

在bit::string中增加移动构造，**移动构造本质是将参数右值的资源窃取过来，占位已有，那么就不用做深拷贝了，所以它叫做移动构造，就是窃取别人的资源来构造自己。**

```
// 移动构造
string(string&& s)
 :_str(nullptr)
```

```
 ,_size(0)
 ,_capacity(0)
{
    cout << "string(string&& s) -- 移动语义" << endl;
    swap(s);
}

int main()
{
    bit::string ret2 = bit::to_string(-1234);

    return 0;
}
```

再运行上面bit::to_string的两个调用，我们会发现，这里没有调用深拷贝的拷贝构造，而是调用了移动构造，移动构造中没有新开空间，拷贝数据，所以效率提高了。

```
// 移动构造
string(string&& s)
    :_str(nullptr)
    , _size(0)
    , _capacity(0)
{
    cout << "string(string&& s) -- 移动语义" << endl;
    swap(s);
}

// 拷贝构造
string(const string& s)
    :_str(nullptr)
{
    cout << "string(const string& s) -- 深拷贝" << endl;

    string tmp(s._str);
    swap(tmp);
}

bit::string to_string(int value)
{
    bit::string str;
    //...

    return str;
}

int main()
{
    bit::string ret2 = bit::to_string(-1234);

    return 0;
}
```

to_string的返回值是一个右值，用这个右值构造ret2，如果既有拷贝构造又有移动构造，调用就会匹配调用移动构造，因为编译器会选择最匹配的参数调用。那么这里就是一个移动语义

**不仅仅有移动构造，还有移动赋值：**

在bit::string类中增加移动赋值函数，再去调用bit::to_string(1234)，不过这次是将bit::to_string(1234)返回的右值对象赋值给ret1对象，这时调用的是移动构造。

```
// 移动赋值
string& operator=(string&& s)
{
 cout << "string& operator=(string&& s) -- 移动语义" << endl;
 swap(s);

 return *this;
}

int main()
{
    bit::string ret1;
    ret1 = bit::to_string(1234);

    return 0;
}

// 运行结果：
// string(string&& s) -- 移动语义
// string& operator=(string&& s) -- 移动语义
```

这里运行后，我们看到调用了一次移动构造和一次移动赋值。因为如果是用一个已经存在的对象接收，编译器就没办法优化了。bit::to_string函数中会先用str生成构造生成一个临时对象，但是我们可以看到，编译器很聪明的在这里把str识别成了右值，调用了移动构造。然后在把这个临时对象做为bit::to_string函数调用的返回值赋值给ret1，这里调用的移动赋值。

**STL中的容器都是增加了移动构造和移动赋值：**

http://www.cplusplus.com/reference/string/string/string/

http://www.cplusplus.com/reference/vector/vector/vector/

## 7.4 右值引用引用左值及其一些更深入的使用场景分析

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);
}
```

```
int main()
{
    bit::string s1("hello world");
    // 这里s1是左值，调用的是拷贝构造
    bit::string s2(s1);
    // 这里我们把s1 move处理以后，会被当成右值，调用移动构造
    // 但是这里要注意，一般是不要这样用的，因为我们会发现s1的
    // 资源被转移给了s3，s1被置空了。
    bit::string s3(std::move(s1));

    return 0;
}
```

**STL容器插入接口函数也增加了右值引用版本：**

http://www.cplusplus.com/reference/list/list/push_back/

http://www.cplusplus.com/reference/vector/vector/push_back/

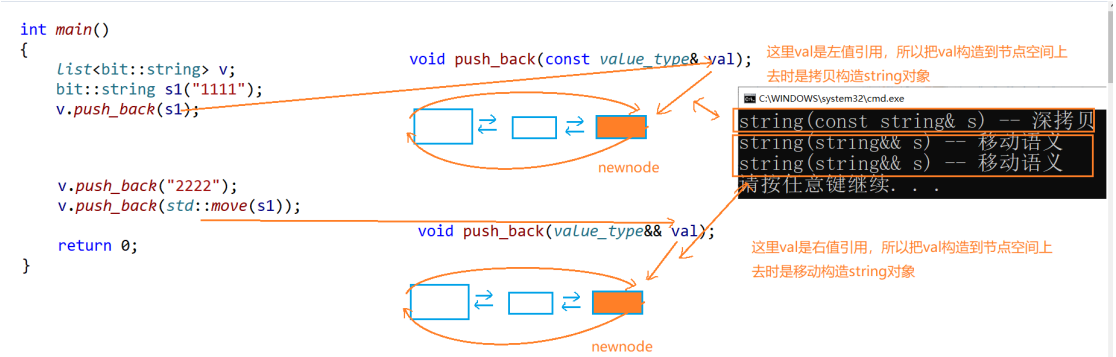
```
void push_back (value_type&& val);

int main()
{
    list<bit::string> lt;
    bit::string s1("1111");
 // 这里调用的是拷贝构造
    lt.push_back(s1);

 // 下面调用都是移动构造
    lt.push_back("2222");
    lt.push_back(std::move(s1));

    return 0;
}

运行结果：
// string(const string& s) -- 深拷贝
// string(string&& s) -- 移动语义
// string(string&& s) -- 移动语义
```




## 7.5 完美转发

**模板中的&& 万能引用**

```
void Fun(int &x){ cout << "左值引用" << endl; }
void Fun(const int &x){ cout << "const 左值引用" << endl; }
```

```cpp
void Fun(int &&x){ cout << "右值引用" << endl; }
void Fun(const int &&x){ cout << "const 右值引用" << endl; }

// 模板中的&&不代表右值引用，而是万能引用，其既能接收左值又能接收右值。
// 模板的万能引用只是提供了能够接收同时接收左值引用和右值引用的能力，
// 但是引用类型的唯一作用就是限制了接收的类型，后续使用中都退化成了左值，
// 我们希望能够在传递过程中保持它的左值或者右值的属性，就需要用我们下面学习的完美转发
template<typename T>
void PerfectForward(T&& t)
{
    Fun(t);
}

int main()
{
    PerfectForward(10);           // 右值

    int a;
    PerfectForward(a);            // 左值
    PerfectForward(std::move(a)); // 右值

    const int b = 8;
    PerfectForward(b);            // const 左值
    PerfectForward(std::move(b)); // const 右值

    return 0;
}
```

**std::forward 完美转发在传参的过程中保留对象原生类型属性**

```cpp
void Fun(int &x){ cout << "左值引用" << endl; }
void Fun(const int &x){ cout << "const 左值引用" << endl; }

void Fun(int &&x){ cout << "右值引用" << endl; }
void Fun(const int &&x){ cout << "const 右值引用" << endl; }

// std::forward<T>(t)在传参的过程中保持了t的原生类型属性。
template<typename T>
void PerfectForward(T&& t)
{
    Fun(std::forward<T>(t));
}

int main()
{
    PerfectForward(10);           // 右值

    int a;
    PerfectForward(a);            // 左值
    PerfectForward(std::move(a)); // 右值

    const int b = 8;
    PerfectForward(b);            // const 左值
    PerfectForward(std::move(b)); // const 右值

    return 0;
```

```
}
```

**完美转发实际中的使用场景：**

```
template<class T>
struct ListNode
{
    ListNode* _next = nullptr;
    ListNode* _prev = nullptr;
    T _data;
};

template<class T>
class List
{
    typedef ListNode<T> Node;
public:
    List()
    {
        _head = new Node;
        _head->_next = _head;
        _head->_prev = _head;
    }

    void PushBack(T&& x)
    {
        //Insert(_head, x);
        Insert(_head, std::forward<T>(x));
    }

    void PushFront(T&& x)
    {
        //Insert(_head->_next, x);
        Insert(_head->_next, std::forward<T>(x));
    }

    void Insert(Node* pos, T&& x)
    {
        Node* prev = pos->_prev;
        Node* newnode = new Node;
        newnode->_data = std::forward<T>(x); // 关键位置

        // prev newnode pos
        prev->_next = newnode;
        newnode->_prev = prev;
        newnode->_next = pos;
        pos->_prev = newnode;
    }

    void Insert(Node* pos, const T& x)
    {
        Node* prev = pos->_prev;
        Node* newnode = new Node;
        newnode->_data = x; // 关键位置

        // prev newnode pos
        prev->_next = newnode;
```

```
        newnode->_prev = prev;
        newnode->_next = pos;
        pos->_prev = newnode;
    }
private:
    Node* _head;
};

int main()
{
    List<bit::string> lt;
    lt.PushBack("1111");
    lt.PushFront("2222");

    return 0;
}
```

## 8 新的类功能

**默认成员函数**

原来C++类中，有6个默认成员函数：

1. 构造函数
2. 析构函数
3. 拷贝构造函数
4. 拷贝赋值重载
5. 取地址重载
6. const 取地址重载

最后重要的是前4个，后两个用处不大。默认成员函数就是我们不写编译器会生成一个默认的。

C++11 新增了两个：移动构造函数和移动赋值运算符重载。

针对移动构造函数和移动赋值运算符重载有一些需要注意的点如下：

- 如果你没有自己实现移动构造函数，且没有实现析构函数 、拷贝构造、拷贝赋值重载中的任意一个。那么编译器会自动生成一个默认移动构造。默认生成的移动构造函数，对于内置类型成员会执行逐成员按字节拷贝，自定义类型成员，则需要看这个成员是否实现移动构造，如果实现了就调用移动构造，没有实现就调用拷贝构造。
- 如果你没有自己实现移动赋值重载函数，且没有实现析构函数 、拷贝构造、拷贝赋值重载中的任意一个，那么编译器会自动生成一个默认移动赋值。默认生成的移动构造函数，对于内置类型成员会执行逐成员按字节拷贝，自定义类型成员，则需要看这个成员是否实现移动赋值，如果实现了就调用移动赋值，没有实现就调用拷贝赋值。(默认移动赋值跟上面移动构造完全类似)
- 如果你提供了移动构造或者移动赋值，编译器不会自动提供拷贝构造和拷贝赋值。

```
// 以下代码在vs2013中不能体现，在vs2019下才能演示体现上面的特性。
class Person
{
public:
    Person(const char* name = "", int age = 0)
        :_name(name)
        , _age(age)
    {}

    /*Person(const Person& p)
```

```
        :_name(p._name)
        ,_age(p._age)
    {}*/

    /*Person& operator=(const Person& p)
    {
        if(this != &p)
        {
            _name = p._name;
            _age = p._age;
        }
        return *this;
    }*/

    /*~Person()
    {}*/

private:
    bit::string _name;
    int _age;
};

int main()
{
    Person s1;
    Person s2 = s1;
    Person s3 = std::move(s1);
    Person s4;
    s4 = std::move(s2);

    return 0;
}
```

**类成员变量初始化**

C++11允许在类定义时给成员变量初始缺省值，默认生成构造函数会使用这些缺省值初始化，这个我们在雷和对象默认就讲了，这里就不再细讲了。

**强制生成默认函数的关键字default:**

C++11可以让你更好的控制要使用的默认函数。假设你要使用某个默认的函数，但是因为一些原因这个函数没有默认生成。比如：我们提供了拷贝构造，就不会生成移动构造了，那么我们可以使用default关键字显示指定移动构造生成。

```
class Person
{
public:
    Person(const char* name = "", int age = 0)
        :_name(name)
        , _age(age)
    {}

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

```
    Person(Person&& p) = default;

private:
    bit::string _name;
    int _age;
};

int main()
{
    Person s1;
    Person s2 = s1;
    Person s3 = std::move(s1);

    return 0;
}
```

**禁止生成默认函数的关键字delete:**

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，并且只声明补丁已，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 Person
{
public:
    Person(const char* name = "", int age = 0)
        :_name(name)
        , _age(age)
    {}

    Person(const Person& p) = delete;

private:
    bit::string _name;
    int _age;
};

int main()
{
    Person s1;
    Person s2 = s1;
    Person s3 = std::move(s1);

    return 0;
}
```

**继承和多态中的final与override关键字**

这个我们在继承和多态章节已经进行了详细讲解这里就不再细讲，需要的话去复习继承和多台章节吧。

## 9 可变参数模板

C++11的新特性可变参数模板能够让您创建可以接受可变参数的函数模板和类模板，相比C++98/03，类模版和函数模版中只能含固定数量的模版参数，可变模版参数无疑是一个巨大的改进。然而由于可变模版参数比较抽象，使用起来需要一定的技巧，所以这块还是比较晦涩的。现阶段呢，我们掌握一些基础的可变参数模板特性就够我们用了，所以这里我们点到为止，以后大家如果有需要，再可以深入学习。

下面就是一个基本可变参数的函数模板

```
// Args是一个模板参数包，args是一个函数形参参数包
// 声明一个参数包Args...args，这个参数包中可以包含0到任意个模板参数。
template <class ...Args>
void ShowList(Args... args)
{}
```

上面的参数args前面有省略号，所以它就是一个可变模版参数，我们把带省略号的参数称为“参数包”，它里面包含了0到N（N>=0）个模版参数。我们无法直接获取参数包args中的每个参数的，只能通过展开参数包的方式来获取参数包中的每个参数，这是使用可变模版参数的一个主要特点，也是最大的难点，即如何展开可变模版参数。由于语法不支持使用args[i]这样方式获取可变参数，所以我们的用一些奇招来一一获取参数包的值。

**递归函数方式展开参数包**

```
// 递归终止函数
template <class T>
void ShowList(const T& t)
{
    cout << t << endl;
}

// 展开函数
template <class T, class ...Args>
void ShowList(T value, Args... args)
{
    cout << value <<" ";
    ShowList(args...);
}

int main()
{
    ShowList(1);
    ShowList(1, 'A');
    ShowList(1, 'A', std::string("sort"));

    return 0;
}
```

**逗号表达式展开参数包**

这种展开参数包的方式，不需要通过递归终止函数，是直接在expand函数体中展开的, printarg不是一个递归终止函数，只是一个处理参数包中每一个参数的函数。这种就地展开参数包的方式实现的关键是逗号表达式。我们知道逗号表达式会按顺序执行逗号前面的表达式。

expand函数中的逗号表达式：(printarg(args), 0)，也是按照这个执行顺序，先执行printarg(args)，再得到逗号表达式的结果0。同时还用到了C++11的另外一个特性——初始化列表，通过初始化列表来初始化一个变长数组, {(printarg(args), 0)...}将会展开成((printarg(arg1),0), (printarg(arg2),0), (printarg(arg3),0), etc... )，最终会创建一个元素值都为0的数组int arr[sizeof...

(Args)]。由于是逗号表达式，在创建数组的过程中会先执行逗号表达式前面的部分printarg(args)打印出参数，也就是说在构造int数组的过程中就将参数包展开了，这个数组的目的纯粹是为了在数组构造的过程展开参数包

```
template <class T>
void PrintArg(T t)
{
    cout << t << " ";
}

//展开函数
template <class ...Args>
void ShowList(Args... args)
{
    int arr[] = { (PrintArg(args), 0)... };
    cout << endl;
}

int main()
{
    ShowList(1);
    ShowList(1, 'A');
    ShowList(1, 'A', std::string("sort"));

    return 0;
}
```

**STL容器中的empalce相关接口函数：**

http://www.cplusplus.com/reference/vector/vector/emplace_back/

http://www.cplusplus.com/reference/list/list/emplace_back/

```
template <class... Args>
void emplace_back (Args&&... args);
```

首先我们看到的emplace系列的接口，支持模板的可变参数，并且万能引用。那么相对insert和emplace系列接口的优势到底在哪里呢？

```
int main()
{
    std::list< std::pair<int, char> > mylist;
    // emplace_back支持可变参数，拿到构建pair对象的参数后自己去创建对象
    // 那么在这里我们可以看到除了用法上，和push_back没什么太大的区别
    mylist.emplace_back(10, 'a');
    mylist.emplace_back(20, 'b');
    mylist.emplace_back(make_pair(30, 'c'));
    mylist.push_back(make_pair(40, 'd'));
    mylist.push_back({ 50, 'e' });

    for (auto e : mylist)
        cout << e.first << ":" << e.second << endl;

    return 0;
}
```

```
int main()
{
 // 下面我们试一下带有拷贝构造和移动构造的bit::string，再试试呢
 // 我们会发现其实差别也不到，emplace_back是直接构造了，push_back
 // 是先构造，再移动构造，其实也还好。
    std::list< std::pair<int, bit::string> > mylist;
    mylist.emplace_back(10, "sort");
    mylist.emplace_back(make_pair(20, "sort"));
    mylist.push_back(make_pair(30, "sort"));
    mylist.push_back({ 40, "sort"});

    return 0;
}
```

## 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>

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;  // 名字
    double _price; // 价格
    int _evaluate; // 评价

    Goods(const char* str, double price, int evaluate)
        :_name(str)
        , _price(price)
        , _evaluate(evaluate)
    {}
};


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

```
    }
};

struct ComparePriceGreater
{
    bool operator()(const Goods& gl, const Goods& gr)
    {
        return gl._price > gr._price;
    }
};

int main()
{
    vector<Goods> v = { { "苹果", 2.1, 5 }, { "香蕉", 3, 4 }, { "橙子", 2.2,
3 }, { "菠萝", 1.5, 4 } };

    sort(v.begin(), v.end(), ComparePriceLess());
 sort(v.begin(), v.end(), ComparePriceGreater());
}
```

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C++11语法中出现了Lambda表达式。

### 10.2 lambda表达式

```
int main()
{
    vector<Goods> v = { { "苹果", 2.1, 5 }, { "香蕉", 3, 4 }, { "橙子", 2.2,
3 }, { "菠萝", 1.5, 4 } };
    sort(v.begin(), v.end(), [](const Goods& g1, const Goods& g2){
        return g1._price < g2._price; });
    sort(v.begin(), v.end(), [](const Goods& g1, const Goods& g2){
        return g1._price > g2._price; });
    sort(v.begin(), v.end(), [](const Goods& g1, const Goods& g2){
        return g1._evaluate < g2._evaluate; });
    sort(v.begin(), v.end(), [](const Goods& g1, const Goods& g2){
        return g1._evaluate > g2._evaluate; });
}
```

上述代码就是使用C++11中的lambda表达式来解决，可以看出lambda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。

```
Rate r1(rate); 函数对象底层代码
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)
```

```
auto r2 = [=](double monty, int year)->double{return mont
lea   eax,[rate]        lambda表达式底层代码
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 包装器

### function包装器

function包装器 也叫作适配器。C++中的function本质是一个类模板，也是一个包装器。

那么我们来看看，我们为什么需要function呢?

```
ret = func(x);
// 上面func可能是什么呢？那么func可能是函数名？函数指针？函数对象(仿函数对象)？也有可能是lamber表达式对象？所以这些都是可调用的类型！如此丰富的类型，可能会导致模板的效率低下！为什么呢？我们继续往下看
template<class F, class T>
T useF(F f, T x)
{
    static int count = 0;
    cout << "count:" << ++count << endl;
    cout << "count:" << &count << endl;

    return f(x);
}

double f(double i)
{
    return i / 2;
}

struct Functor
{
    double operator()(double d)
    {
        return d / 3;
    }
};

int main()
{
```

```
    // 函数名
    cout << useF(f, 11.11) << endl;

    // 函数对象
    cout << useF(Functor(), 11.11) << endl;

    // lamber表达式
    cout << useF([](double d)->double{ return d/4; }, 11.11) << endl;

    return 0;
}
```

**通过上面的程序验证，我们会发现useF函数模板实例化了三份。**

包装器可以很好的解决上面的问题

```
std::function在头文件<functional>

// 类模板原型如下
template <class T> function;     // undefined

template <class Ret, class... Args>
class function<Ret(Args...)>;

模板参数说明：
Ret: 被调用函数的返回类型
Args…: 被调用函数的形参
```

```
// 使用方法如下：
#include <functional>

int f(int a, int b)
{
    return a + b;
}

struct Functor
{
public:
    int operator() (int a, int b)
    {
        return a + b;
    }
};

class Plus
{
public:
    static int plusi(int a, int b)
    {
        return a + b;
    }

    double plusd(double a, double b)
    {
        return a + b;
    }
```

```
};

int main()
{
    // 函数名(函数指针)
    std::function<int(int, int)> func1 = f;
    cout << func1(1, 2) << endl;

    // 函数对象
    std::function<int(int, int)> func2 = Functor();
    cout << func2(1, 2) << endl;

    // lamber表达式
    std::function<int(int, int)> func3 = [](const int a, const int b)
{return a + b; };

    cout << func3(1, 2) << endl;

    // 类的成员函数
    std::function<int(int, int)> func4 = &Plus::plusi;
    cout << func4(1, 2) << endl;

    std::function<double(Plus, double, double)> func5 = &Plus::plusd;
    cout << func5(Plus(), 1.1, 2.2) << endl;

    return 0;
}
```

有了包装器，如何解决模板的效率低下，实例化多份的问题呢?

```
#include <functional>

template<class F, class T>
T useF(F f, T x)
{
    static int count = 0;
    cout << "count:" << ++count << endl;
    cout << "count:" << &count << endl;

    return f(x);
}

double f(double i)
{
    return i / 2;
}

struct Functor
{
    double operator()(double d)
    {
        return d / 3;
    }
};

int main()
{
```

```
    // 函数名
    std::function<double(double)> func1 = f;
    cout << useF(func1, 11.11) << endl;

    // 函数对象
    std::function<double(double)> func2 = Functor();
    cout << useF(func2, 11.11) << endl;

    // lamber表达式
    std::function<double(double)> func3 = [](double d)->double{ return d /
4; };
    cout << useF(func3, 11.11) << endl;

    return 0;
}
```

包装器的其他一些场景：

https://leetcode-cn.com/problems/evaluate-reverse-polish-notation/submissions/

```
class Solution {
public:
int evalRPN(vector<string>& tokens) {
  stack<int> st;
     for(auto& str : tokens)
     {
         if(str == "+" || str == "-" || str == "*" || str == "/")
         {
             int right = st.top();
             st.pop();
             int left = st.top();
             st.pop();

             switch(str[0])
          {
                 case '+':
                     st.push(left+right);
                     break;
                 case '-':
                     st.push(left-right);
                     break;
                 case '*':
                     st.push(left*right);
                     break;
                 case '/':
                     st.push(left/right);
                     break;
             }
         }
         else
         {
             // 1、atoi  itoa
             // 2、sprintf scanf
             // 3、stoi  to_string  C++11
             st.push(stoi(str));
         }
     }
```

```
        return st.top();
    }
};

// 使用包装器以后的玩法
class Solution {
public:
int evalRPN(vector<string>& tokens) {
  stack<int> st;
     map<string, function<int(int, int)>> opFuncMap =
    {
        { "+",  [](int i, int j){return i + j; } },
        { "-",  [](int i, int j){return i - j; } },
        { "*",  [](int i, int j){return i * j; } },
  { "/",  [](int i, int j){return i / j; } }
    };

  for(auto& str : tokens)
  {
        if(opFuncMap.find(str) != opFuncMap.end())
        {
            int right = st.top();
            st.pop();
            int left = st.top();
            st.pop();

            st.push(opFuncMap[str](left, right));
      }
        else
        {
            // 1、atoi  itoa
            // 2、sprintf scanf
            // 3、stoi  to_string  C++11
            st.push(stoi(str));
        }
    }

    return st.top();
}
};
```

## bind

std::bind函数定义在头文件中，**是一个函数模板，它就像一个函数包装器(适配器)，接受一个可调用对象（callable object），生成一个新的可调用对象来“适应”原对象的参数列表。**一般而言，我们用它可以把一个原本接收N个参数的函数fn，通过绑定一些参数，返回一个接收M个（M可以大于N，但这么做没什么意义）参数的新函数。同时，使用std::bind函数还可以实现参数顺序调整等操作。

```
// 原型如下：
template <class Fn, class... Args>
/* unspecified */ bind (Fn&& fn, Args&&... args);

// with return type (2)
template <class Ret, class Fn, class... Args>
/* unspecified */ bind (Fn&& fn, Args&&... args);
```

可以将bind函数看作是一个通用的函数适配器，它接受一个可调用对象，生成一个新的可调用对象来“适应”原对象的参数列表。
调用bind的一般形式：auto newCallable = bind(callable,arg_list);

其中，newCallable本身是一个可调用对象，arg_list是一个逗号分隔的参数列表，对应给定的callable的参数。**当我们调用newCallable时，newCallable会调用callable,并传给它arg_list中的参数。**

arg_list中的参数可能包含形如_n的名字，其中n是一个整数，这些参数是“占位符”，表示newCallable的参数，它们占据了传递给newCallable的参数的“位置”。数值n表示生成的可调用对象中参数的位置：_1为newCallable的第一个参数，_2为第二个参数，以此类推。

```
// 使用举例
#include <functional>

int Plus(int a, int b)
{
    return a + b;
}

class Sub
{
public:
    int sub(int a, int b)
    {
        return a - b;
    }
};

int main()
{
    //表示绑定函数plus 参数分别由调用 func1 的第一，二个参数指定
    std::function<int(int, int)> func1 = std::bind(Plus, placeholders::_1,
placeholders::_2);
    //auto func1 = std::bind(Plus, placeholders::_1, placeholders::_2);

    //func2的类型为 function<void(int, int, int)> 与func1类型一样
    //表示绑定函数 plus 的第一，二为： 1， 2
    auto  func2 = std::bind(Plus, 1, 2);
    cout << func1(1, 2) << endl;
    cout << func2() << endl;

    Sub s;
    // 绑定成员函数
    std::function<int(int, int)> func3 = std::bind(&Sub::sub, s,
placeholders::_1, placeholders::_2);

    // 参数调换顺序
```

```
    std::function<int(int, int)> func4 = std::bind(&Sub::sub, s,
placeholders::_2, placeholders::_1);
    cout << func3(1, 2) << endl;
    cout << func4(1, 2) << endl;

    return 0;
}
```

## 12 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

```
#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。

### 11.4 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>
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

#### 11.6 支持两个线程交替打印，一个打印奇数，一个打印偶数

本节主要演示了condition_variable的使用，condition_variable熟悉我们linux课程已经讲过了，他们用来进行线程之间的互相通知。condition_variable和Linux posix的条件变量并没有什么大的区别，主要还是面向对象实现的。条件变量的文档如下：https://cplusplus.com/reference/condition_variable/

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

void two_thread_print()
{
    std::mutex mtx;
    condition_variable c;
    int n = 100;
    bool flag = true;

    thread t1([&](){
        int i = 0;
        while (i < n)
        {
            unique_lock<mutex> lock(mtx);
            c.wait(lock, [&]()->bool{return flag; });
            cout << i << endl;
            flag = false;
            i += 2; // 偶数
            c.notify_one();
        }
    });

    thread t2([&](){
        int j = 1;
        while (j < n)
        {
            unique_lock<mutex> lock(mtx);
            c.wait(lock, [&]()->bool{return !flag; });
            cout << j << endl;
            j += 2; // 奇数
            flag = true;
            c.notify_one();
        }
```

```
    });

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

int main()
{
    two_thread_print();

    return 0;
}
```