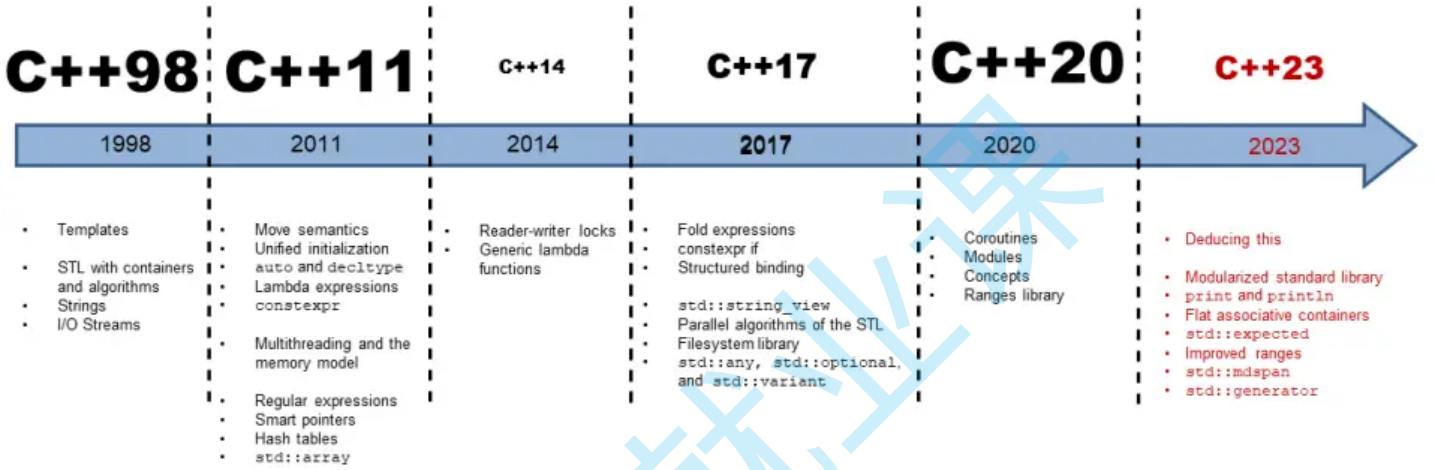
11.C++11

1. C++11的发展历史

C++11 是 C++ 的第二个主要版本，并且是从 C++98 起的最重要更新。它引入了大量更改，标准化了既有实践，并改进了对 C++ 程序员可用的抽象。在它最终由 ISO 在 2011 年 8 月 12 日采纳前，人们曾使用名称“C++0x”，因为它曾被期待在 2010 年之前发布。C++03 与 C++11 期间花了 8 年时间，故而这是迄今为止最长的版本间隔。从那时起，C++ 有规律地每 3 年更新一次。



2. 列表初始化

2.1 C++98传统的{}

C++98中一般数组和结构体可以用{}进行初始化。

```
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;
}
```

2.2 C++11中的{}

- C++11以后想统一初始化方式，试图实现一切对象皆可用{}初始化，{}初始化也叫做列表初始化。
- 内置类型支持，自定义类型也支持，自定义类型本质是类型转换，中间会产生临时对象，最后优化了以后变成直接构造。
- {}初始化的过程中，可以省略掉=
- C++11列表初始化的本意是想实现一个大统一的初始化方式，其次他在有些场景下带来的不少便利，如容器push/inset多参数构造的对象时，{}初始化会很方便

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

struct Point
{
        int _x;
        int _y;
};

class Date
{
public:
        Date(int year = 1, int month = 1, int day = 1)
                :_year(year)
                , _month(month)
                , _day(day)
        {
                cout << "Date(int year, int month, int day)" << endl;
        }

        Date(const Date& d)
                :_year(d._year)
                , _month(d._month)
                , _day(d._day)
        {
                cout << "Date(const Date& d)" << endl;
        }
private:
        int _year;
        int _month;
        int _day;
};

// 一切皆可用列表初始化，且可以不加=
```

```cpp
int main()
{
        // C++98支持的
        int a1[] = { 1, 2, 3, 4, 5 };
        int a2[5] = { 0 };
        Point p = { 1, 2 };

        // C++11支持的
        // 内置类型支持
        int x1 = { 2 };
        // 自定义类型支持
        // 这里本质是用{ 2025, 1, 1}构造一个Date临时对象
        // 临时对象再去拷贝构造d1，编译器优化后合二为一变成{ 2025, 1, 1}直接构造初始化d1
        // 运行一下，我们可以验证上面的理论，发现是没调用拷贝构造的
        Date d1 = { 2025, 1, 1};
        // 这里d2引用的是{ 2024, 7, 25 }构造的临时对象
        const Date& d2 = { 2024, 7, 25 };

        // 需要注意的是C++98支持单参数时类型转换，也可以不用{}
        Date d3 = { 2025};
        Date d4 = 2025;

        // 可以省略掉=
        Point p1 { 1, 2 };
        int x2 { 2 };
        Date d6 { 2024, 7, 25 };
        const Date& d7 { 2024, 7, 25 };

        // 不支持，只有{}初始化，才能省略=
        // Date d8 2025;

        vector<Date> v;
        v.push_back(d1);
        v.push_back(Date(2025, 1, 1));

        // 比起有名对象和匿名对象传参，这里{}更有性价比
        v.push_back({ 2025, 1, 1 });

        return 0;
}
```


2.3 C++11中的std::initializer_list

- 上面的初始化已经很方便，但是对象容器初始化还是不太方便，比如一个vector对象，我想用N个值去构造初始化，那么我们得实现很多个构造函数才能支持，`vector<int> v1 = {1,2,3};vector<int> v2 = {1,2,3,4,5};`
- C++11库中提出了一个std::initializer_list的类，`auto il = { 10, 20, 30 };  // the type of il is an initializer_list`，这个类的本质是底层开一个数组，将数据拷贝过来，std::initializer_list内部有两个指针分别指向数组的开始和结束。
- 这是他的文档：initializer_list，std::initializer_list支持迭代器遍历。
- 容器支持一个std::initializer_list的构造函数，也就支持任意多个值构成的`{x1,x2,x3...}`进行初始化。STL中的容器支持任意多个值构成的`{x1,x2,x3...}`进行初始化，就是通过std::initializer_list的构造函数支持的。

```
// STL中的容器都增加了一个initializer_list的构造
vector (initializer_list<value_type> il, const allocator_type& alloc = allocator_type());
list (initializer_list<value_type> il, const allocator_type& alloc = allocator_type());
map (initializer_list<value_type> il,const key_compare& comp = key_compare(),const allocator_type& alloc = allocator_type());
// ...

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

     vector(initializer_list<T> l)
     {
         for (auto e : l)
                 push_back(e)
     }
 private:
     iterator _start = nullptr;
     iterator _finish = nullptr;
     iterator _endofstorage = nullptr;
 };

// 另外，容器的赋值也支持initializer_list的版本
vector& operator= (initializer_list<value_type> il);
map& operator= (initializer_list<value_type> il);
```

```
#include<iostream>
```

```cpp
#include<vector>
#include<string>
#include<map>
using namespace std;

int main()
{
    std::initializer_list<int> mylist;
    mylist = { 10, 20, 30 };
    cout << sizeof(mylist) << endl;

    // 这里begin和end返回的值initializer_list对象中存的两个指针
    // 这两个指针的值跟i的地址跟接近，说明数组存在栈上
    int i = 0;
    cout << mylist.begin() << endl;
    cout << mylist.end() << endl;
    cout << &i << endl;

    // {}列表中可以有任意多个值
    // 这两个写法语义上还是有差别的，第一个v1是直接构造，
    // 第二个v2是构造临时对象+临时对象拷贝v2+优化为直接构造
    vector<int> v1({ 1,2,3,4,5 });
    vector<int> v2 = { 1,2,3,4,5 };
    const vector<int>& v3 = { 1,2,3,4,5 };

    // 这里是pair对象的{}初始化和map的initializer_list构造结合到一起用了
    map<string, string> dict = { {"sort", "排序"}, {"string", "字符串"}};

    // initializer_list版本的赋值支持
    v1 = { 10,20,30,40,50 };

    return 0;
}
```


3. 右值引用和移动语义

C++98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，C++11之后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。

3.1 左值和右值

- 左值是一个表示数据的表达式(如变量名或解引用的指针)，一般是有持久状态，存储在内存中，我们可以获取它的地址，左值可以出现赋值符号的左边，也可以出现在赋值符号右边。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。

- 右值也是一个表示数据的表达式，要么是字面值常量、要么是表达式求值过程中创建的临时对象等，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。
- 值得一提的是，左值的英文简写为lvalue，右值的英文简写为rvalue。传统认为它们分别是left value、right value 的缩写。现代C++中，lvalue 被解释为loactor value的缩写，可意为存储在内存中、有明确存储地址可以取地址的对象，而 rvalue 被解释为 read value，指的是那些可以提供数据值，但是不可以寻址，例如：临时变量，字面量常量，存储于寄存器中的变量等，也就是说左值和右值的核心区别就是能否取地址。

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

int main()
{
    // 左值：可以取地址
    // 以下的p、b、c、*p、s、s[0]就是常见的左值
    int* p = new int(0);
    int b = 1;
    const int c = b;
    *p = 10;
    string s("111111");
    s[0] = 'x';

    cout << &c << endl;
    cout << (void*)&s[0] << endl;

    // 右值：不能取地址
    double x = 1.1, y = 2.2;
    // 以下几个10、x + y、fmin(x, y)、string("11111")都是常见的右值
    10;
    x + y;
    fmin(x, y);
    string("11111");

    //cout << &10 << endl;
    //cout << &(x+y) << endl;
    //cout << &(fmin(x, y)) << endl;
    //cout << &string("11111") << endl;

    return 0;
}
```


3.2 左值引用和右值引用

- `Type& r1 = x; Type&& rr1 = y;`第一个语句就是左值引用，左值引用就是给左值取别名，第二个就是右值引用，同样的道理，右值引用就是给右值取别名。
- 左值引用不能直接引用右值，但是const左值引用可以引用右值
- 右值引用不能直接引用左值，但是右值引用可以引用move(左值)
- `template <class T> typename remove_reference<T>::type&& move (T&& arg);`
- move是库里面的一个函数模板，本质内部是进行强制类型转换，当然他还涉及一些引用折叠的知识，这个我们后面会细讲。
- 需要注意的是变量表达式都是左值属性，也就意味着一个右值被右值引用绑定后，右值引用变量变量表达式的属性是左值
- 语法层面看，左值引用和右值引用都是取别名，不开空间。从汇编底层的角度看下面代码中r1和rr1汇编层实现，底层都是用指针实现的，没什么区别。底层汇编等实现和上层语法表达的意义有时是背离的，所以不要然到一起去理解，互相佐证，这样反而是陷入迷途。

```
template <class _Ty>
remove_reference_t<_Ty>&& move(_Ty&& _Arg)
{    // forward _Arg as movable
    return static_cast<remove_reference_t<_Ty>&&>(_Arg);
}

#include<iostream>
using namespace std;

int main()
{
    // 左值：可以取地址
    // 以下的p、b、c、*p、s、s[0]就是常见的左值
    int* p = new int(0);
    int b = 1;
    const int c = b;
    *p = 10;
    string s("111111");
    s[0] = 'x';
    double x = 1.1, y = 2.2;

    // 左值引用给左值取别名
    int& r1 = b;
    int*& r2 = p;
    int& r3 = *p;
    string& r4 = s;
    char& r5 = s[0];

    // 右值引用给右值取别名
```

```
    int&& rr1 = 10;
    double&& rr2 = x + y;
    double&& rr3 = fmin(x, y);
    string&& rr4 = string("11111");

    // 左值引用不能直接引用右值，但是const左值引用可以引用右值
    const int& rx1 = 10;
    const double& rx2 = x + y;
    const double& rx3 = fmin(x, y);
    const string& rx4 = string("11111");

    // 右值引用不能直接引用左值，但是右值引用可以引用move(左值)
    int&& rrx1 = move(b);
    int*&& rrx2 = move(p);
    int&& rrx3 = move(*p);
    string&& rrx4 = move(s);
    string&& rrx5 = (string&&)s;

    // b、r1、rr1都是变量表达式，都是左值
    cout << &b << endl;
    cout << &r1 << endl;
    cout << &rr1 << endl;

    // 这里要注意的是，rr1的属性是左值，所以不能再被右值引用绑定，除非move一下
    int& r6 = r1;
    // int&& rrx6 = rr1;
    int&& rrx6 = move(rr1);

    return 0;
}
```


3.3 引用延长生命周期

右值引用可用于为临时对象延长生命周期，const 的左值引用也能延长临时对象生存期，但这些对象无法被修改。

```
int main()
{
    std::string s1 = "Test";
    //  std::string&& r1 = s1;           // 错误：不能绑定到左值

    const std::string& r2 = s1 + s1;     // OK：到 const 的左值引用延长生存期
    //  r2 += "Test";                    // 错误：不能通过到 const 的引用修改

    std::string&& r3 = s1 + s1;          // OK：右值引用延长生存期
```

```
    r3 += "Test";                          // OK：能通过到非 const 的引用修改

    std::cout << r3 << '\n';

    return 0;
}
```

3.4 左值和右值的参数匹配

- C++98中，我们实现一个const左值引用作为参数的函数，那么实参传递左值和右值都可以匹配。
- C++11以后，分别重载左值引用、const左值引用、右值引用作为形参的f函数，那么实参是左值会匹配f(左值引用)，实参是const左值会匹配f(const 左值引用)，实参是右值会匹配f(右值引用)。
- 右值引用变量在用于表达式时属性是左值，这个设计这里会感觉跟怪，下一小节我们讲右值引用的使用场景时，就能体会这样设计的价值了

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

void f(int& x)
{
    std::cout << "左值引用重载 f(" << x << ")\n";
}

void f(const int& x)
{
    std::cout << "到 const 的左值引用重载 f(" << x << ")\n";
}

void f(int&& x)
{
    std::cout << "右值引用重载 f(" << x << ")\n";
}

int main()
{
    int i = 1;
    const int ci = 2;

    f(i);  // 调用 f(int&)
    f(ci); // 调用 f(const int&)
    f(3);  // 调用 f(int&&)，如果没有 f(int&&) 重载则会调用 f(const int&)
    f(std::move(i)); // 调用 f(int&&)

    // 右值引用变量在用于表达式时是左值
```

```
    int&& x = 1;
    f(x);            // 调用 f(int& x)
    f(std::move(x)); // 调用 f(int&& x)

    return 0;
}
```

3.5 右值引用和移动语义的使用场景

3.5.1 左值引用主要使用场景回顾

左值引用主要使用场景是在函数中左值引用传参和左值引用传返回值时减少拷贝，同时还可以修改实参和修改返回对象的价值。左值引用已经解决大多数场景的拷贝效率问题，但是有些场景不能使用传左值引用返回，如addStrings和generate函数，C++98中的解决方案只能是被迫使用输出型参数解决。那么C++11以后这里可以使用右值引用做返回值解决吗？显然是不可能的，因为这里的本质是返回对象是一个局部对象，函数结束这个对象就析构销毁了，右值引用返回也无法概念对象已经析构销毁的事实。

```
class Solution {
public:
    // 传值返回需要拷贝
    string addStrings(string num1, string num2) {
        string str;
        int end1 = num1.size()-1, end2 = num2.size()-1;
        // 进位
        int next = 0;
        while(end1 >= 0 || end2 >= 0)
        {
            int val1 = end1 >= 0 ? num1[end1--]-'0' : 0;
            int val2 = end2 >= 0 ? num2[end2--]-'0' : 0;

            int ret = val1 + val2+next;
            next = ret / 10;
            ret = ret % 10;

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

        if(next == 1)
            str += '1';

        reverse(str.begin(), str.end());

        return str;
```

```
    }
};

class Solution {
public:
    // 这里的传值返回拷贝代价就太大了
    vector<vector<int>> generate(int numRows) {
        vector<vector<int>> vv(numRows);
        for(int i = 0; i < numRows; ++i)
        {
            vv[i].resize(i+1, 1);
        }

        for(int i = 2; i < numRows; ++i)
        {
            for(int j = 1; j < i; ++j)
            {
                vv[i][j] = vv[i-1][j] + vv[i-1][j-1];
            }
        }

        return vv;
    }
};
```

3.5.2 移动构造和移动赋值

- 移动构造函数是一种构造函数，类似拷贝构造函数，移动构造函数要求第一个参数是该类类型的引用，但是不同的是要求这个参数是右值引用，如果还有其他参数，额外的参数必须有缺省值。
- 移动赋值是一个赋值运算符的重载，他跟拷贝赋值构成函数重载，类似拷贝赋值函数，移动赋值函数要求第一个参数是该类类型的引用，但是不同的是要求这个参数是右值引用。
- 对于像string/vector这样的深拷贝的类或者包含深拷贝的成员变量的类，移动构造和移动赋值才有意义，因为移动构造和移动赋值的第一个参数都是右值引用的类型，他的本质是要“窃取”引用的右值对象的资源，而不是像拷贝构造和拷贝赋值那样去拷贝资源，从提高效率。下面的bit::string样例实现了移动构造和移动赋值，我们需要结合场景理解。

```
#define _CRT_SECURE_NO_WARNINGS 1
#include<iostream>
#include<assert.h>
#include<string.h>
#include<algorithm>
using namespace std;

namespace bit
```

```cpp
{
    class string
    {
    public:
        typedef char* iterator;
        typedef const char* const_iterator;

        iterator begin()
        {
                return _str;
        }
        iterator end()
        {
                return _str + _size;
        }

        const_iterator begin() const
        {
                return _str;
        }

        const_iterator end() const
        {
                return _str + _size;
        }

        string(const char* str = "")
                :_size(strlen(str))
                , _capacity(_size)
        {
                cout << "string(char* str)-构造" << endl;
                _str = new char[_capacity + 1];
                strcpy(_str, str);
        }

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

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

```

```
56                  reserve(s._capacity);
57                  for (auto ch : s)
58                  {
59                          push_back(ch);
60                  }
61          }
62
63          // 移动构造
64          string(string&& s)
65          {
66                  cout << "string(string&& s) -- 移动构造" << endl;
67                  swap(s);
68          }
69
70          string& operator=(const string& s)
71          {
72                  cout << "string& operator=(const string& s) -- 拷贝赋值" <<
    endl;
73                  if (this != &s)
74                  {
75                          _str[0] = '\0';
76                          _size = 0;
77
78                          reserve(s._capacity);
79                          for (auto ch : s)
80                          {
81                                  push_back(ch);
82                          }
83                  }
84
85                  return *this;
86          }
87
88          // 移动赋值
89          string& operator=(string&& s)
90          {
91                  cout << "string& operator=(string&& s) -- 移动赋值" << endl;
92                  swap(s);
93                  return *this;
94          }
95
96          ~string()
97          {
98                  cout << "~string() -- 析构" << endl;
99                  delete[] _str;
100                 _str = nullptr;
101         }
```

```
102
103         char& operator[](size_t pos)
104         {
105                 assert(pos < _size);
106                 return _str[pos];
107         }
108
109         void reserve(size_t n)
110         {
111                 if (n > _capacity)
112                 {
113                         char* tmp = new char[n + 1];
114                         if (_str)
115                         {
116                                 strcpy(tmp, _str);
117                                 delete[] _str;
118                         }
119                         _str = tmp;
120                         _capacity = n;
121                 }
122         }
123
124         void push_back(char ch)
125         {
126                 if (_size >= _capacity)
127                 {
128                         size_t newcapacity = _capacity == 0 ? 4 : _capacity *
    2;
129                         reserve(newcapacity);
130                 }
131
132                 _str[_size] = ch;
133                 ++_size;
134                 _str[_size] = '\0';
135         }
136
137         string& operator+=(char ch)
138         {
139                 push_back(ch);
140                 return *this;
141         }
142
143         const char* c_str() const
144         {
145                 return _str;
146         }
147
```

```
148         size_t size() const
149         {
150                 return _size;
151         }
152     private:
153         char* _str = nullptr;
154         size_t _size = 0;
155         size_t _capacity = 0;
156     };
157 }
158
159 int main()
160 {
161     bit::string s1("xxxxx");
162     // 拷贝构造
163     bit::string s2 = s1;
164     // 构造+移动构造，优化后直接构造
165     bit::string s3 = bit::string("yyyyy");
166     // 移动构造
167     bit::string s4 = move(s1);
168     cout << "********************************" << endl;
169
170     return 0;
171 }
```

3.5.3 右值引用和移动语义解决传值返回问题

```
1  namespace bit
2  {
3      string addStrings(string num1, string num2)
4      {
5              string str;
6              int end1 = num1.size() - 1, end2 = num2.size() - 1;
7              int next = 0;
8              while (end1 >= 0 || end2 >= 0)
9              {
10                     int val1 = end1 >= 0 ? num1[end1--] - '0' : 0;
11                     int val2 = end2 >= 0 ? num2[end2--] - '0' : 0;
12
13                     int ret = val1 + val2 + next;
14                     next = ret / 10;
15                     ret = ret % 10;
16
17                     str += ('0' + ret);
18             }
```

```
19
20              if (next == 1)
21                      str += '1';
22
23              reverse(str.begin(), str.end());
24
25              cout << "******************************" << endl;
26
27              return str;
28      }
29  }
30
31  // 场景1
32  int main()
33  {
34      bit::string ret = bit::addStrings("11111", "2222");
35      cout << ret.c_str() << endl;
36
37      return 0;
38  }
39
40  // 场景2
41  int main()
42  {
43      bit::string ret;
44      ret = bit::addStrings("11111", "2222");
45      cout << ret.c_str() << endl;
46
47      return 0;
48  }
```


右值对象构造，只有拷贝构造，没有移动构造的场景

- 图1展示了vs2019 debug环境下编译器对拷贝的优化，左边为不优化的情况下，两次拷贝构造，右边为编译器优化的场景下连续步骤中的拷贝合二为一变为一次拷贝构造。
- 需要注意的是在vs2019的release和vs2022的debug和release，下面代码优化为非常恐怖，会直接将str对象的构造，str拷贝构造临时对象，临时对象拷贝构造ret对象，合三为一，变为直接构造。变为直接构造。要理解这个优化要结合局部对象生命周期和栈帧的角度理解，如图3所示。
- linux下可以将下面代码拷贝到test.cpp文件，编译时用 `g++ test.cpp -fno-elide-constructors` 的方式关闭构造优化，运行结果可以看到图1左边没有优化的两次拷贝。

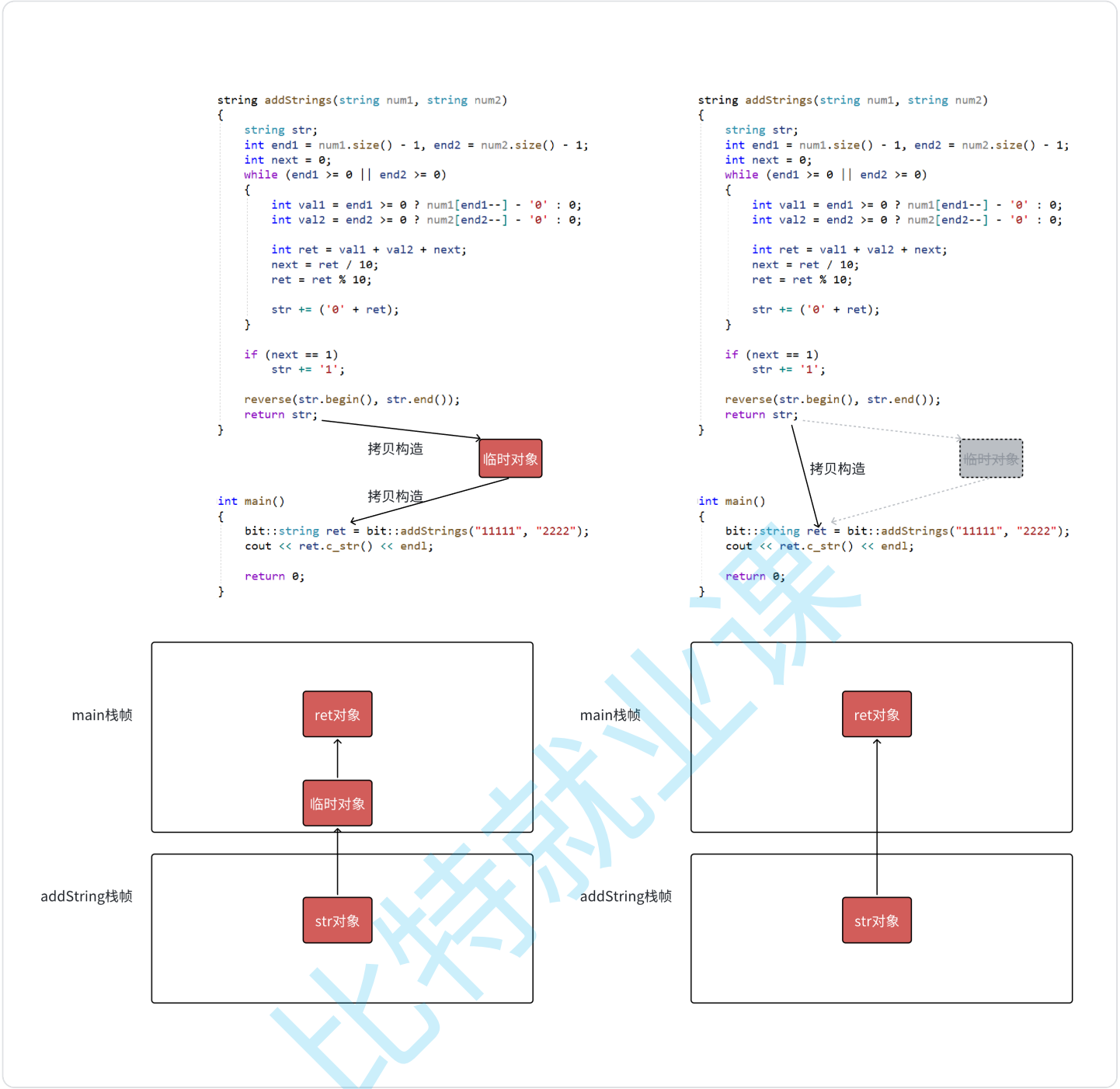


图1

右值对象构造，有拷贝构造，也有移动构造的场景

- 图2展示了vs2019 debug环境下编译器对拷贝的优化，左边为不优化的情况下，两次移动构造，右边为编译器优化的场景下连续步骤中的拷贝合二为一变为一次移动构造。
- 需要注意的是在vs2019的release和vs2022的debug和release，下面代码优化为非常恐怖，会直接将str对象的构造，str拷贝构造临时对象，临时对象拷贝构造ret对象，合三为一，变为直接构造。要理解这个优化要结合局部对象生命周期和栈帧的角度理解，如图3所示。
- linux下可以将下面代码拷贝到test.cpp文件，编译时用 `g++ test.cpp -fno-elide-constructors` 的方式关闭构造优化，运行结果可以看到图1左边没有优化的两次移动。

```cpp
string addStrings(string num1, string num2)
{
    string str;
    int end1 = num1.size() - 1, end2 = num2.size() - 1;
    int next = 0;
    while (end1 >= 0 || end2 >= 0)
    {
        int val1 = end1 >= 0 ? num1[end1--] - '0' : 0;
        int val2 = end2 >= 0 ? num2[end2--] - '0' : 0;

        int ret = val1 + val2 + next;
        next = ret / 10;
        ret = ret % 10;

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

    if (next == 1)
        str += '1';

    reverse(str.begin(), str.end());
    return str;
}

int main()
{
    bit::string ret = bit::addStrings("11111", "2222");
    cout << ret.c_str() << endl;

    return 0;
}
```

移动构造

临时对象

移动构造

```cpp
string addStrings(string num1, string num2)
{
    string str;
    int end1 = num1.size() - 1, end2 = num2.size() - 1;
    int next = 0;
    while (end1 >= 0 || end2 >= 0)
    {
        int val1 = end1 >= 0 ? num1[end1--] - '0' : 0;
        int val2 = end2 >= 0 ? num2[end2--] - '0' : 0;

        int ret = val1 + val2 + next;
        next = ret / 10;
        ret = ret % 10;

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

    if (next == 1)
        str += '1';

    reverse(str.begin(), str.end());
    return str;
}

int main()
{
    bit::string ret = bit::addStrings("11111", "2222");
    cout << ret.c_str() << endl;

    return 0;
}
```

移动构造

临时对象

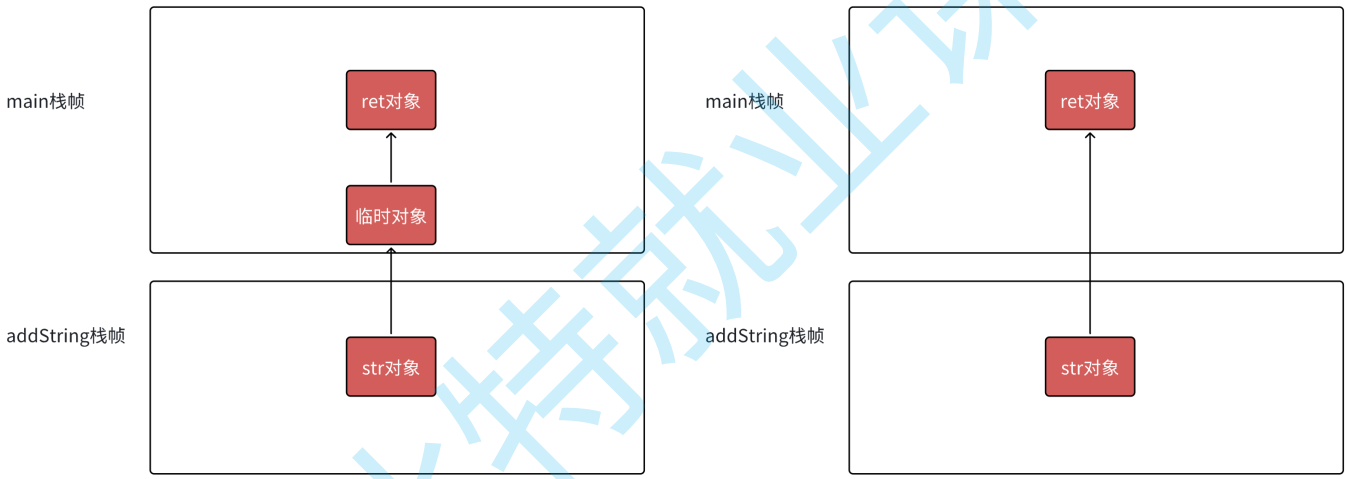


图2

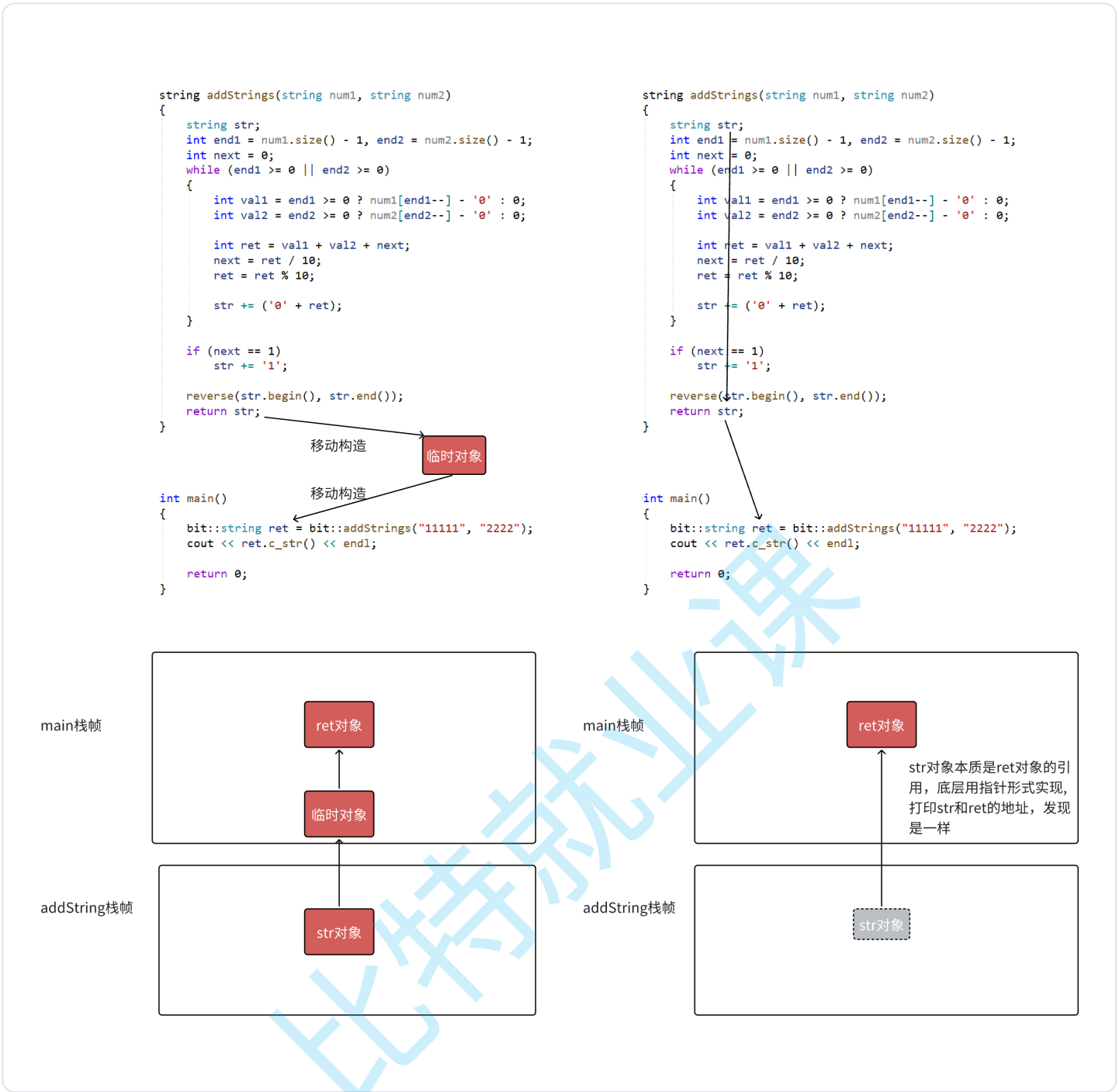


图3

右值对象赋值，只有拷贝构造和拷贝赋值，没有移动构造和移动赋值的场景

- 图4左边展示了vs2019 debug和 `g++ test.cpp -fno-elide-constructors` 关闭优化环境下编译器的处理，一次拷贝构造，一次拷贝赋值。
- 需要注意的是在vs2019的release和vs2022的debug和release，下面代码会进一步优化，直接构造要返回的临时对象，str本质是临时对象的引用，底层角度用指针实现。运行结果的角度，我们可以看到str的析构是在赋值以后，说明str就是临时对象的别名。

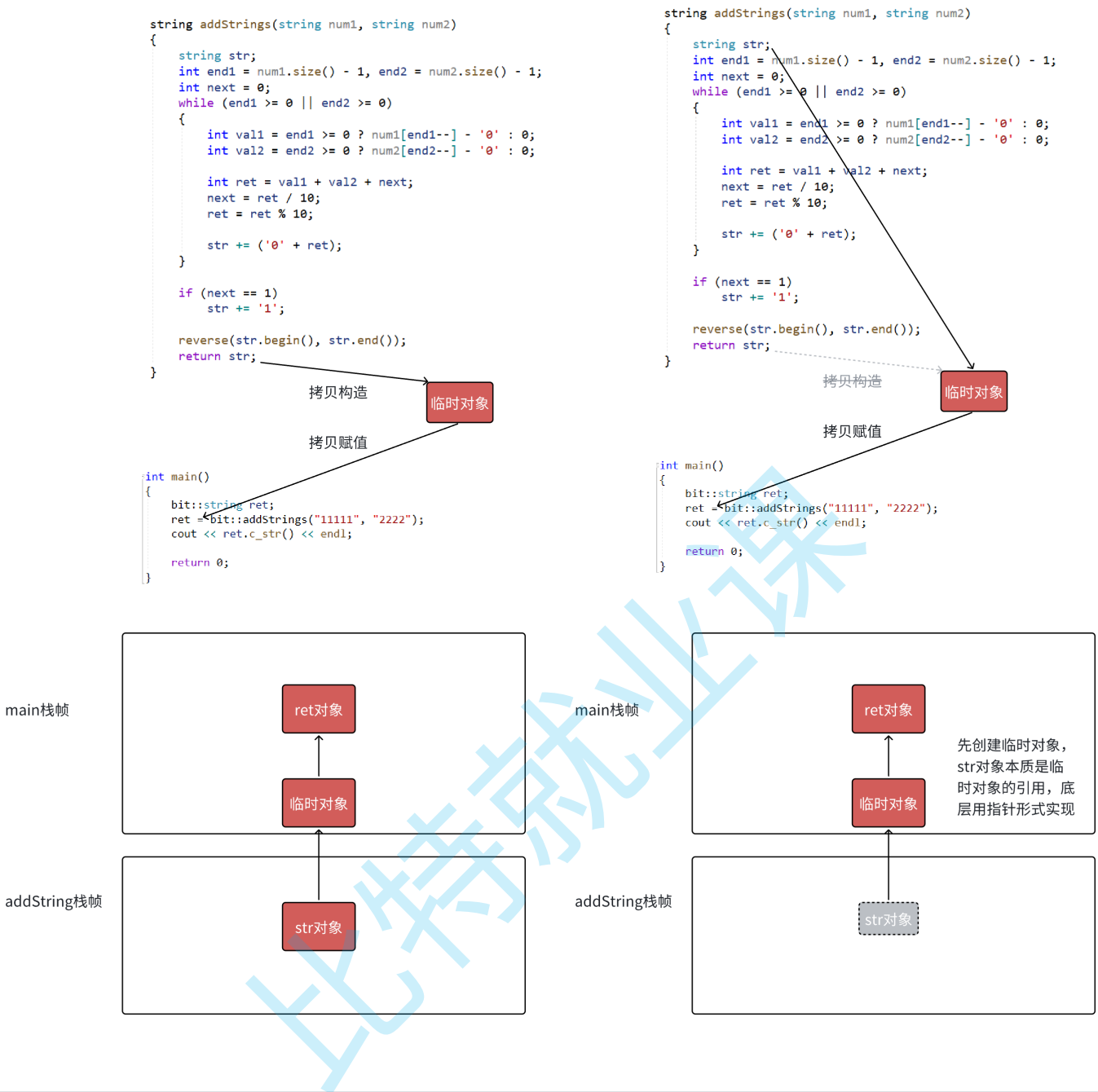


图4

右值对象赋值，既有拷贝构造和拷贝赋值，也有移动构造和移动赋值的场景

- 图5左边展示了vs2019 debug和 `g++ test.cpp -fno-elide-constructors` 关闭优化环境下编译器的处理，一次移动构造，一次移动赋值。
- 需要注意的是在vs2019的release和vs2022的debug和release，下面代码会进一步优化，直接构造要返回的临时对象，str本质是临时对象的引用，底层角度用指针实现。运行结果的角度，我们可以看到str的析构是在赋值以后，说明str就是临时对象的别名。

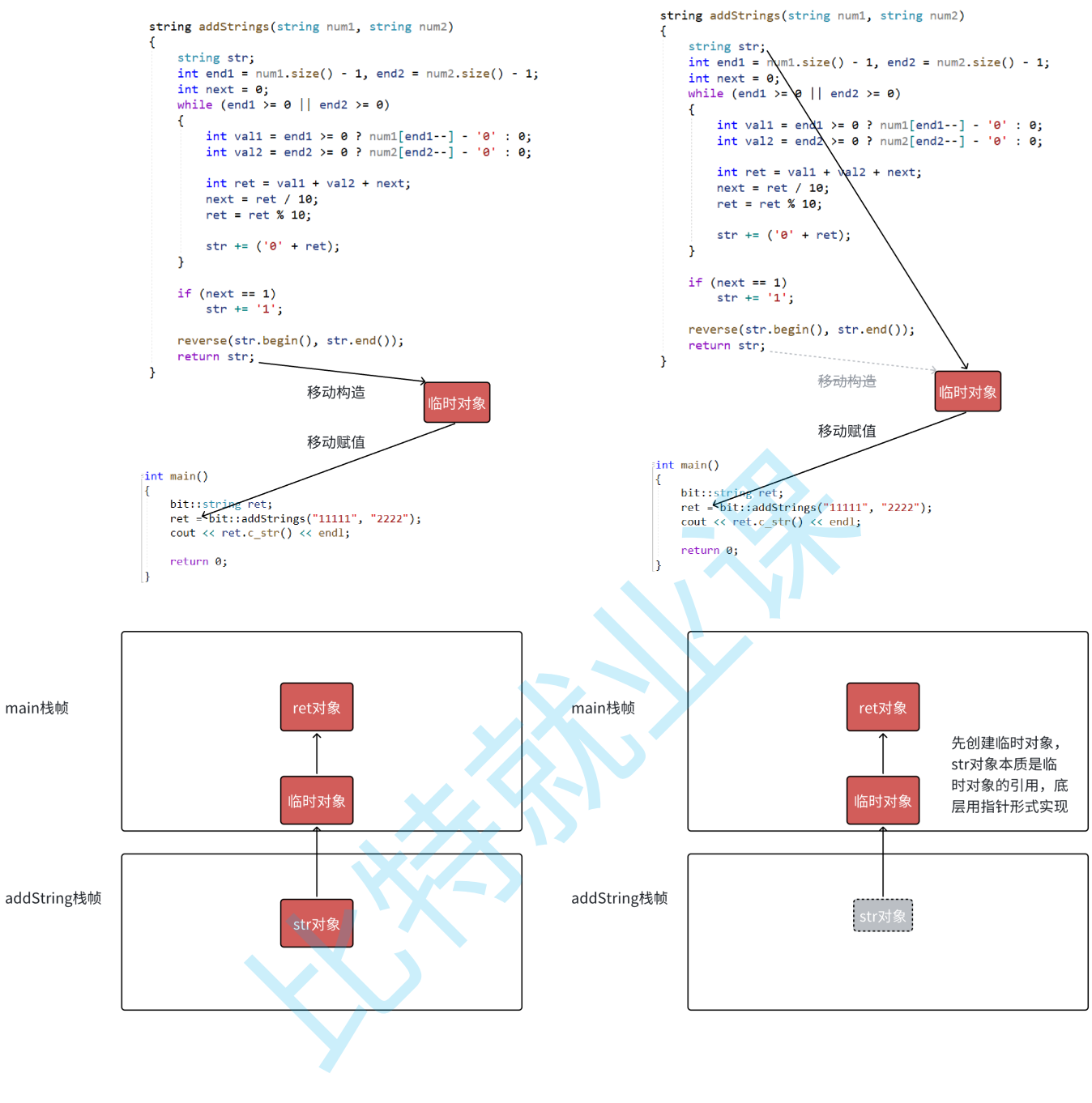


图5

3.5.4 右值引用和移动语义在传参中的提效

- 查看STL文档我们发现C++11以后容器的push和insert系列的接口否增加的右值引用版本
- 当实参是一个左值时，容器内部继续调用拷贝构造进行拷贝，将对象拷贝到容器空间中的对象
- 当实参是一个右值，容器内部则调用移动构造，右值对象的资源到容器空间的对象上
- 把我们之前模拟实现的bit::list拷贝过来，支持右值引用参数版本的push_back和insert
- 其实这里还有一个emplace系列的接口，但是这个涉及可变参数模板，我们需要把可变参数模板讲解以后再讲解emplace系列的接口。

```cpp
// void push_back (const value_type& val);
// void push_back (value_type&& val);
// iterator insert (const_iterator position, value_type&& val);
// iterator insert (const_iterator position, const value_type& val);

int main()
{
    std::list<bit::string> lt;

    bit::string s1("1111111111111111111");
    lt.push_back(s1);
    cout << "************************" << endl;

    lt.push_back(bit::string("22222222222222222222222222"));
    cout << "************************" << endl;

    lt.push_back("33333333333333333333333333");
    cout << "************************" << endl;

    lt.push_back(move(s1));
    cout << "************************" << endl;

    return 0;
}
```

运行结果：

```
string(char* str)
string(const string& s) -- 拷贝构造
************************
string(char* str)
string(string&& s) -- 移动构造
~string() -- 析构
************************
string(char* str)
string(string&& s) -- 移动构造
~string() -- 析构
************************
string(string&& s) -- 移动构造
************************
~string() -- 析构
~string() -- 析构
~string() -- 析构
~string() -- 析构
~string() -- 析构
```

```cpp
// List.h
// 以下代码为了控制课件篇幅，把跟这里无关的接口都删除了
namespace bit
{
    template<class T>
    struct ListNode
    {
            ListNode<T>* _next;
            ListNode<T>* _prev;

            T _data;

            ListNode(const T& data = T())
                    :_next(nullptr)
                    ,_prev(nullptr)
                    ,_data(data)
            {}

            ListNode(T&& data)
                    :_next(nullptr)
                    , _prev(nullptr)
                    , _data(move(data))
            {}
    };

    template<class T, class Ref, class Ptr>
    struct ListIterator
    {
            typedef ListNode<T> Node;
            typedef ListIterator<T, Ref, Ptr> Self;
            Node* _node;

            ListIterator(Node* node)
                    :_node(node)
            {}

            Self& operator++()
            {
                    _node = _node->_next;
                    return *this;
            }

            Ref operator*()
            {
                    return _node->_data;
            }

```

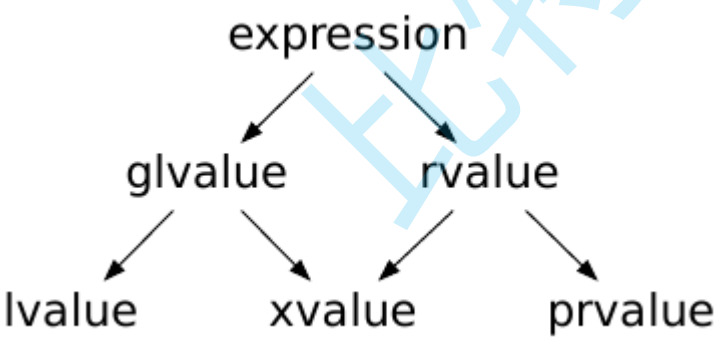
```
48              bool operator!=(const Self& it)
49              {
50                      return _node != it._node;
51              }
52      };
53
54      template<class T>
55      class list
56      {
57              typedef ListNode<T> Node;
58      public:
59              typedef ListIterator<T, T&, T*> iterator;
60              typedef ListIterator<T, const T&, const T*> const_iterator;
61
62              iterator begin()
63              {
64                      return iterator(_head->_next);
65              }
66
67              iterator end()
68              {
69                      return iterator(_head);
70              }
71
72              void empty_init()
73              {
74                      _head = new Node();
75                      _head->_next = _head;
76                      _head->_prev = _head;
77              }
78
79              list()
80              {
81                      empty_init();
82              }
83
84              void push_back(const T& x)
85              {
86                      insert(end(), x);
87              }
88
89              void push_back(T&& x)
90              {
91                      insert(end(), move(x));
92              }
93
94              iterator insert(iterator pos, const T& x)
```

```
95              {
96                      Node* cur = pos._node;
97                      Node* newnode = new Node(x);
98                      Node* prev = cur->_prev;
99
100                     // prev  newnode  cur
101                     prev->_next = newnode;
102                     newnode->_prev = prev;
103                     newnode->_next = cur;
104                     cur->_prev = newnode;
105
106                     return iterator(newnode);
107             }
108
109             iterator insert(iterator pos, T&& x)
110             {
111                     Node* cur = pos._node;
112                     Node* newnode = new Node(move(x));
113                     Node* prev = cur->_prev;
114
115                     // prev  newnode  cur
116                     prev->_next = newnode;
117                     newnode->_prev = prev;
118                     newnode->_next = cur;
119                     cur->_prev = newnode;
120
121                     return iterator(newnode);
122             }
123     private:
124             Node* _head;
125     };
126 }
127
128 // Test.cpp
129 #include"List.h"
130 int main()
131 {
132     bit::list<bit::string> lt;
133     cout << "**************************" << endl;
134
135     bit::string s1("11111111111111111111");
136     lt.push_back(s1);
137     cout << "**************************" << endl;
138
139     lt.push_back(bit::string("22222222222222222222222222"));
140     cout << "**************************" << endl;
141
```

```
142     lt.push_back("333333333333333333333333333");
143     cout << "************************" << endl;
144
145     lt.push_back(move(s1));
146     cout << "************************" << endl;
147
148     return 0;
149 }
```

3.6 类型分类

- C++11以后，进一步对类型进行了划分，右值被划分纯右值(pure value，简称prvalue)和将亡值(expiring value，简称xvalue)。
- 纯右值是指那些字面值常量或求值结果相当于字面值或是一个不具名的临时对象。如：`42、true、nullptr` 或者类似 `str.substr(1, 2)、str1 + str2` 传值返回函数调用，或者整形 `a、b，a++，a+b` 等。纯右值和将亡值C++11中提出的，C++11中的纯右值概念划分等价于C++98中的右值。
- 将亡值是指返回右值引用的函数的调用表达式和转换为右值引用的转换函数的调用表达，如 `move(x)、static_cast<X&&>(x)`
- 泛左值(generalized value，简称glvalue)，泛左值包含将亡值和左值。
- 值类别 - cppreference.com 和 Value categories这两个关于值类型的中文和英文的官方文档，有兴趣可以了解细节。



3.7 引用折叠

- C++中不能直接定义引用的引用如 `int& && r = i;`，这样写会直接报错，通过模板或 typedef 中的类型操作可以构成引用的引用。
- 通过模板或 typedef 中的类型操作可以构成引用的引用时，这时C++11给出了一个引用折叠的规则：右值引用的右值引用折叠成右值引用，所有其他组合均折叠成左值引用。
- 下面的程序中很好的展示了模板和typedef时构成引用的引用时的引用折叠规则，大家需要一个一个仔细理解一下。

- 像f2这样的函数模板中，T&& x参数看起来是右值引用参数，但是由于引用折叠的规则，他传递左值时就是左值引用，传递右值时就是右值引用，有些地方也把这种函数模板的参数叫做万能引用。
- Function(T&& t)函数模板程序中，假设实参是int右值，模板参数T的推导int，实参是int左值，模板参数T的推导int&，再结合引用折叠规则，就实现了实参是左值，实例化出左值引用版本形参的Function，实参是右值，实例化出右值引用版本形参的Function。

```cpp
// 由于引用折叠限定，f1实例化以后总是一个左值引用
template<class T>
void f1(T& x)
{}

// 由于引用折叠限定，f2实例化后可以是左值引用，也可以是右值引用
template<class T>
void f2(T&& x)
{}

int main()
{
    typedef int& lref;
    typedef int&& rref;
    int n = 0;

    lref& r1 = n;  // r1 的类型是 int&
    lref&& r2 = n; // r2 的类型是 int&
    rref& r3 = n;  // r3 的类型是 int&
    rref&& r4 = 1; // r4 的类型是 int&&

    // 没有折叠->实例化为void f1(int& x)
    f1<int>(n);
    f1<int>(0);  // 报错

    // 折叠->实例化为void f1(int& x)
    f1<int&>(n);
    f1<int&>(0); // 报错

    // 折叠->实例化为void f1(int& x)
    f1<int&&>(n);
    f1<int&&>(0); // 报错

    // 折叠->实例化为void f1(const int& x)
    f1<const int&>(n);
    f1<const int&>(0);

    // 折叠->实例化为void f1(const int& x)
    f1<const int&&>(n);
```

```
    f1<const int&&>(0);

    // 没有折叠->实例化为void f2(int&& x)
    f2<int>(n);   // 报错
    f2<int>(0);

    // 折叠->实例化为void f2(int& x)
    f2<int&>(n);
    f2<int&>(0);  // 报错

    // 折叠->实例化为void f2(int&& x)
    f2<int&&>(n); // 报错
    f2<int&&>(0);

    return 0;
}
```

```
template<class T>
void Function(T&& t)
{
    int a = 0;
    T x = a;
    //x++;
    cout << &a << endl;
    cout << &x << endl << endl;
}

int main()
{
    // 10是右值，推导出T为int，模板实例化为void Function(int&& t)
    Function(10);                                   // 右值

    int a;
    // a是左值，推导出T为int&，引用折叠，模板实例化为void Function(int& t)
    Function(a);                                    // 左值

    // std::move(a)是右值，推导出T为int，模板实例化为void Function(int&& t)
    Function(std::move(a));                         // 右值

    const int b = 8;
    // a是左值，推导出T为const int&，引用折叠，模板实例化为void Function(const int& t)
    // 所以Function内部会编译报错，x不能++
    Function(b);                                    // const 左值

```

```
    // std::move(b)右值，推导出T为const int，模板实例化为void Function(const int&& t)
    // 所以Function内部会编译报错，x不能++
    Function(std::move(b));                    // const 右值

    return 0;
}
```

3.8 完美转发

- Function(T&& t)函数模板程序中，传左值实例化以后是左值引用的Function函数，传右值实例化以后是右值引用的Function函数。
- 但是结合我们在5.2章节的讲解，变量表达式都是左值属性，也就意味着一个右值被右值引用绑定后，右值引用变量表达式的属性是左值，也就是说Function函数中t的属性是左值，那么我们把t传递给下一层函数Fun，那么匹配的都是左值引用版本的Fun函数。这里我们想要保持t对象的属性，就需要使用完美转发实现。
- `template <class T> T&& forward (typename remove_reference<T>::type& arg);`
- `template <class T> T&& forward (typename remove_reference<T>::type&& arg);`
- 完美转发forward本质是一个函数模板，他主要还是通过引用折叠的方式实现，下面示例中传递给Function的实参是右值，T被推导为int，没有折叠，forward内部t被强转为右值引用返回；传递给Function的实参是左值，T被推导为int&，引用折叠为左值引用，forward内部t被强转为左值引用返回。

```
template <class _Ty>
_Ty&& forward(remove_reference_t<_Ty>& _Arg) noexcept
{   // forward an lvalue as either an lvalue or an rvalue
    return static_cast<_Ty&&>(_Arg);
}

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; }

template<class T>
void Function(T&& t)
{
    Fun(t);
```

```
17      //Fun(forward<T>(t));
18  }
19
20  int main()
21  {
22      // 10是右值，推导出T为int，模板实例化为void Function(int&& t)
23      Function(10);                          // 右值
24
25      int a;
26      // a是左值，推导出T为int&，引用折叠，模板实例化为void Function(int& t)
27      Function(a);                           // 左值
28
29      // std::move(a)是右值，推导出T为int，模板实例化为void Function(int&& t)
30      Function(std::move(a));                // 右值
31
32      const int b = 8;
33      // a是左值，推导出T为const int&，引用折叠，模板实例化为void Function(const int&
    t)
34      Function(b);                           // const 左值
35
36      // std::move(b)右值，推导出T为const int，模板实例化为void Function(const int&&
    t)
37      Function(std::move(b));                // const 右值
38
39      return 0;
40  }
```

4. 可变参数模板

4.1 基本语法及原理

- C++11支持可变参数模板，也就是说支持可变数量参数的函数模板和类模板，可变数目的参数被称为参数包，存在两种参数包：模板参数包，表示零或多个模板参数；函数参数包：表示零或多个函数参数。
- `template <class ...Args> void Func(Args... args) {}`
- `template <class ...Args> void Func(Args&... args) {}`
- `template <class ...Args> void Func(Args&&... args) {}`
- 我们用省略号来指出一个模板参数或函数参数的表示一个包，在模板参数列表中，class...或typename...指出接下来的参数表示零或多个类型列表；在函数参数列表中，类型名后面跟...指出接下来表示零或多个形参对象列表；函数参数包可以用左值引用或右值引用表示，跟前面普通模板一样，每个参数实例化时遵循引用折叠规则。
- 可变参数模板的原理跟模板类似，本质还是去实例化对应类型和个数的多个函数。

- 这里我们可以使用sizeof...运算符去计算参数包中参数的个数。

```cpp
template <class ...Args>
void Print(Args&&... args)
{
    cout << sizeof...(args) << endl;
}

int main()
{
    double x = 2.2;
    Print();                            // 包里有0个参数
    Print(1);                           // 包里有1个参数
    Print(1, string("xxxxx"));          // 包里有2个参数
    Print(1.1, string("xxxxx"), x);     // 包里有3个参数

    return 0;
}


// 原理1：编译本质这里会结合引用折叠规则实例化出以下四个函数
void Print();
void Print(int&& arg1);
void Print(int&& arg1, string&& arg2);
void Print(double&& arg1, string&& arg2, double& arg3);

// 原理2：更本质去看没有可变参数模板，我们实现出这样的多个函数模板才能支持
//       这里的功能，有了可变参数模板，我们进一步被解放，他是类型泛化基础
//       上叠加数量变化，让我们泛型编程更灵活。
void Print();

template <class T1>
void Print(T1&& arg1);

template <class T1, class T2>
void Print(T1&& arg1, T2&& arg2);

template <class T1, class T2, class T3>
void Print(T1&& arg1, T2&& arg2, T3&& arg3);
// ...
```


4.2 包扩展

- 对于一个参数包，我们除了能计算他的参数个数，我们能做的唯一的事情就是扩展它，当扩展一个包时，我们还要提供用于每个扩展元素的模式，扩展一个包就是将它分解为构成的元素，对每个元素应用模式，获得扩展后的列表。我们通过在模式的右边放一个省略号(...)来触发扩展操作。底层的实现细节如图1所示。
- C++还支持更复杂的包扩展，直接将参数包依次展开依次作为实参给一个函数去处理。

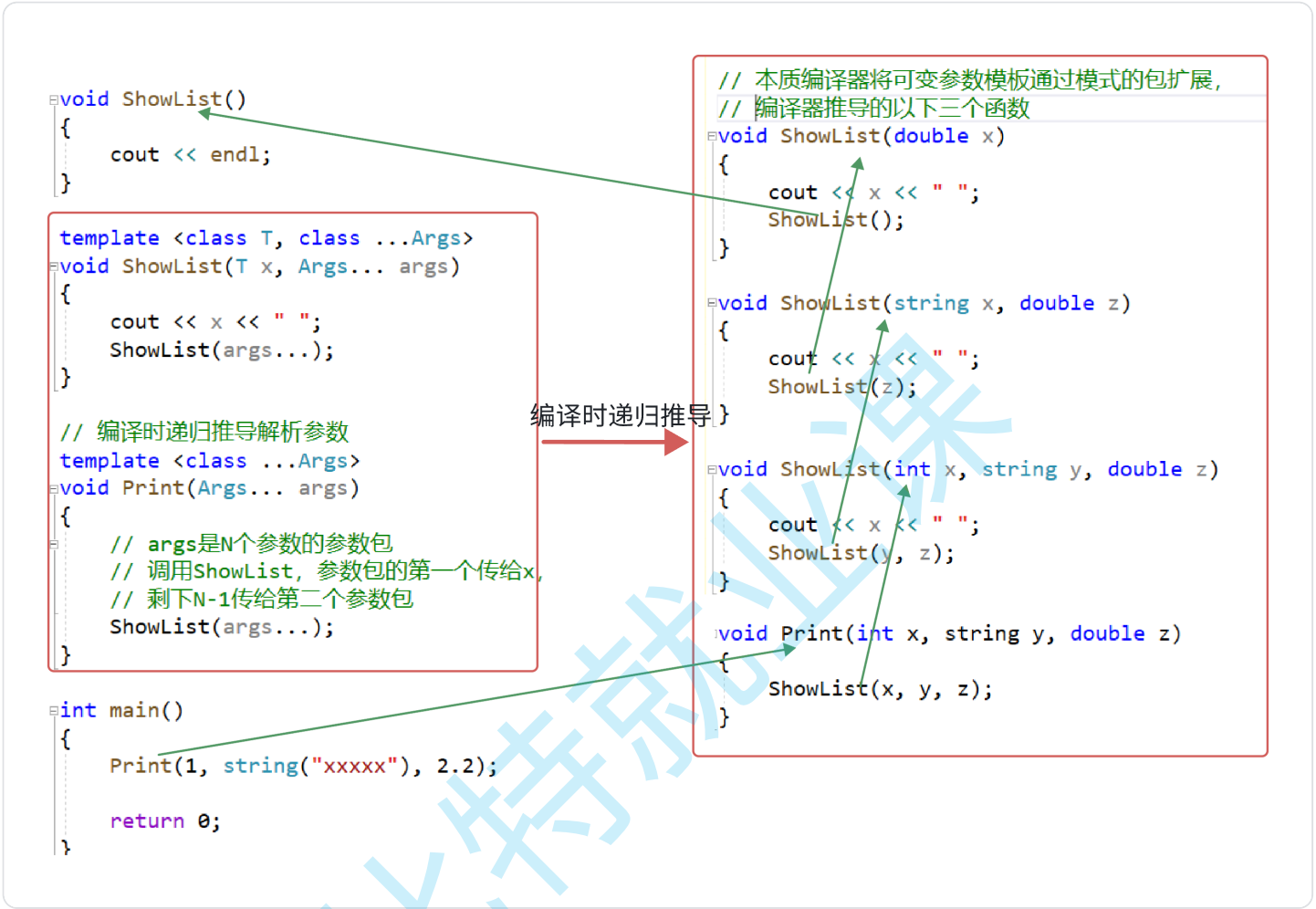


图1

```
// 可变模板参数
// 参数类型可变
// 参数个数可变
// 打印参数包内容
//template <class ...Args>
//void Print(Args... args)
//{
//        // 可变参数模板编译时解析
//        // 下面是运行获取和解析，所以不支持这样用
//        cout << sizeof...(args) << endl;
//        for (size_t i = 0; i < sizeof...(args); i++)
//        {
//                cout << args[i] << " ";
//        }
```

```cpp
//        cout << endl;
//}

void ShowList()
{
    // 编译器时递归的终止条件，参数包是0个时，直接匹配这个函数
    cout << endl;
}

template <class T, class ...Args>
void ShowList(T x, Args... args)
{
    cout << x << " ";
    // args是N个参数的参数包
    // 调用ShowList，参数包的第一个传给x，剩下N-1传给第二个参数包
    ShowList(args...);
}

// 编译时递归推导解析参数
template <class ...Args>
void Print(Args... args)
{
    ShowList(args...);
}

int main()
{
    Print();
    Print(1);
    Print(1, string("xxxxx"));
    Print(1, string("xxxxx"), 2.2);

    return 0;
}

//template <class T, class ...Args>
//void ShowList(T x, Args... args)
//{
//        cout << x << " ";
//        Print(args...);
//}

// Print(1, string("xxxxx"), 2.2);调用时
// 本质编译器将可变参数模板通过模式的包扩展，编译器推导的以下三个重载函数函数
//void ShowList(double x)
//{
//        cout << x << " ";
```

```
//        ShowList();
//}
//
//void ShowList(string x, double z)
//{
//        cout << x << " ";
//        ShowList(z);
//}
//
//void ShowList(int x, string y, double z)
//{
//        cout << x << " ";
//        ShowList(y, z);
//}

//void Print(int x, string y, double z)
//{
//        ShowList(x, y, z);
//}
```

```
template <class T>
const T& GetArg(const T& x)
{
    cout << x << " ";
    return x;
}

template <class ...Args>
void Arguments(Args... args)
{}

template <class ...Args>
void Print(Args... args)
{
    // 注意GetArg必须返回或者到的对象，这样才能组成参数包给Arguments
    Arguments(GetArg(args)...);
}

// 本质可以理解为编译器编译时，包的扩展模式
// 将上面的函数模板扩展实例化为下面的函数
// 是不是很抽象，C++11以后，只能说委员会的大佬设计语法思维跳跃得太厉害
//void Print(int x, string y, double z)
//{
//        Arguments(GetArg(x), GetArg(y), GetArg(z));
//}
```

```
26
27 int main()
28 {
29     Print(1, string("xxxxx"), 2.2);
30
31     return 0;
32 }
```

4.3 empalce系列接口

- `template <class... Args> void emplace_back (Args&&... args);`
- `template <class... Args> iterator emplace (const_iterator position, Args&&... args);`
- C++11以后STL容器新增了empalce系列的接口，empalce系列的接口均为模板可变参数，功能上兼容push和insert系列，但是empalce还支持新玩法，假设容器为container<T>，empalce还支持直接插入构造T对象的参数，这样有些场景会更高效一些，可以直接在容器空间上构造T对象。
- emplace_back总体而言是更高效，推荐以后使用emplace系列替代insert和push系列
- 第二个程序中我们模拟实现了list的emplace和emplace_back接口，这里把参数包不段往下传递，最终在结点的构造中直接去匹配容器存储的数据类型T的构造，所以达到了前面说的empalce支持直接插入构造T对象的参数，这样有些场景会更高效一些，可以直接在容器空间上构造T对象。
- 传递参数包过程中，如果是 `Args&&... args` 的参数包，要用完美转发参数包，方式如下 `std::forward<Args>(args)...` ，否则编译时包扩展后右值引用变量表达式就变成了左值。

```
1  #include<list>
2
3  // emplace_back总体而言是更高效，推荐以后使用emplace系列替代insert和push系列
4  int main()
5  {
6      list<bit::string> lt;
7      // 传左值，跟push_back一样，走拷贝构造
8      bit::string s1("111111111111");
9      lt.emplace_back(s1);
10     cout << "**********************************" << endl;
11
12     // 右值，跟push_back一样，走移动构造
13     lt.emplace_back(move(s1));
14     cout << "**********************************" << endl;
15
16     // 直接把构造string参数包往下传，直接用string参数包构造string
17     // 这里达到的效果是push_back做不到的
18     lt.emplace_back("111111111111");
```

```
    cout << "********************************" << endl;

    list<pair<bit::string, int>> lt1;
    // 跟push_back一样
    // 构造pair + 拷贝/移动构造pair到list的节点中data上
    pair<bit::string, int> kv("苹果", 1);
    lt1.emplace_back(kv);
    cout << "********************************" << endl;

    // 跟push_back一样
    lt1.emplace_back(move(kv));
    cout << "********************************" << endl;

    ////////////////////////////////////////////////////////////////
    // 直接把构造pair参数包往下传，直接用pair参数包构造pair
    // 这里达到的效果是push_back做不到的
    lt1.emplace_back("苹果", 1);
    cout << "********************************" << endl;

    return 0;
}
```

```
// List.h
namespace bit
{
    template<class T>
    struct ListNode
    {
            ListNode<T>* _next;
            ListNode<T>* _prev;

            T _data;

            ListNode(T&& data)
                    :_next(nullptr)
                    , _prev(nullptr)
                    , _data(move(data))
            {}

            template <class... Args>
            ListNode(Args&&... args)
                    : _next(nullptr)
                    , _prev(nullptr)
                    , _data(std::forward<Args>(args)...)
            {}
```

```
24      };
25
26      template<class T, class Ref, class Ptr>
27      struct ListIterator
28      {
29              typedef ListNode<T> Node;
30              typedef ListIterator<T, Ref, Ptr> Self;
31              Node* _node;
32
33              ListIterator(Node* node)
34                      :_node(node)
35              {}
36
37              // ++it;
38              Self& operator++()
39              {
40                      _node = _node->_next;
41                      return *this;
42              }
43
44              Self& operator--()
45              {
46                      _node = _node->_prev;
47                      return *this;
48              }
49
50              Ref operator*()
51              {
52                      return _node->_data;
53              }
54
55              bool operator!=(const Self& it)
56              {
57                      return _node != it._node;
58              }
59      };
60
61      template<class T>
62      class list
63      {
64              typedef ListNode<T> Node;
65      public:
66              typedef ListIterator<T, T&, T*> iterator;
67              typedef ListIterator<T, const T&, const T*> const_iterator;
68
69              iterator begin()
70              {
```

```
71                      return iterator(_head->_next);
72              }
73
74              iterator end()
75              {
76                      return iterator(_head);
77              }
78
79              void empty_init()
80              {
81                      _head = new Node();
82                      _head->_next = _head;
83                      _head->_prev = _head;
84              }
85
86              list()
87              {
88                      empty_init();
89              }
90
91              void push_back(const T& x)
92              {
93                      insert(end(), x);
94              }
95
96              void push_back(T&& x)
97              {
98                      insert(end(), move(x));
99              }
100
101             iterator insert(iterator pos, const T& x)
102             {
103                     Node* cur = pos._node;
104                     Node* newnode = new Node(x);
105                     Node* prev = cur->_prev;
106
107                     // prev  newnode  cur
108                     prev->_next = newnode;
109                     newnode->_prev = prev;
110                     newnode->_next = cur;
111                     cur->_prev = newnode;
112
113                     return iterator(newnode);
114             }
115
116             iterator insert(iterator pos, T&& x)
117             {
```

```
118                     Node* cur = pos._node;
119                     Node* newnode = new Node(move(x));
120                     Node* prev = cur->_prev;
121 
122                     // prev  newnode  cur
123                     prev->_next = newnode;
124                     newnode->_prev = prev;
125                     newnode->_next = cur;
126                     cur->_prev = newnode;
127 
128                     return iterator(newnode);
129             }
130 
131             template <class... Args>
132             void emplace_back(Args&&... args)
133             {
134                     insert(end(), std::forward<Args>(args)...);
135             }
136 
137             // 原理：本质编译器根据可变参数模板生成对应参数的函数
138             /*void emplace_back(string& s)
139             {
140                     insert(end(), std::forward<string>(s));
141             }
142 
143             void emplace_back(string&& s)
144             {
145                     insert(end(), std::forward<string>(s));
146             }
147 
148             void emplace_back(const char* s)
149             {
150                     insert(end(), std::forward<const char*>(s));
151             }
152             */
153 
154             template <class... Args>
155             iterator insert(iterator pos, Args&&... args)
156             {
157                     Node* cur = pos._node;
158                     Node* newnode = new Node(std::forward<Args>(args)...);
159                     Node* prev = cur->_prev;
160 
161                     // prev  newnode  cur
162                     prev->_next = newnode;
163                     newnode->_prev = prev;
164                     newnode->_next = cur;
```

```
165                     cur->_prev = newnode;
166
167                     return iterator(newnode);
168             }
169     private:
170             Node* _head;
171     };
172 }
173
174 // Test.cpp
175 #include"List.h"
176
177 // emplace_back总体而言是更高效，推荐以后使用emplace系列替代insert和push系列
178 int main()
179 {
180     bit::list<bit::string> lt;
181     // 传左值，跟push_back一样，走拷贝构造
182     bit::string s1("111111111111");
183     lt.emplace_back(s1);
184     cout << "**********************************" << endl;
185
186     // 右值，跟push_back一样，走移动构造
187     lt.emplace_back(move(s1));
188     cout << "**********************************" << endl;
189
190     // 直接把构造string参数包往下传，直接用string参数包构造string
191     // 这里达到的效果是push_back做不到的
192     lt.emplace_back("111111111111");
193     cout << "**********************************" << endl;
194
195     bit::list<pair<bit::string, int>> lt1;
196     // 跟push_back一样
197     // 构造pair + 拷贝/移动构造pair到list的节点中data上
198     pair<bit::string, int> kv("苹果", 1);
199     lt1.emplace_back(kv);
200     cout << "**********************************" << endl;
201
202     // 跟push_back一样
203     lt1.emplace_back(move(kv));
204     cout << "**********************************" << endl;
205
206     //////////////////////////////////////////////////////////////////
207     // 直接把构造pair参数包往下传，直接用pair参数包构造pair
208     // 这里达到的效果是push_back做不到的
209     lt1.emplace_back("苹果", 1);
210     cout << "**********************************" << endl;
211
```

```
    return 0;
}
```

5. 新的类功能

5.1 默认的移动构造和移动赋值

- 原来C++类中，有6个默认成员函数：构造函数/析构函数/拷贝构造函数/拷贝赋值重载/取地址重载/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 新增了两个默认成员函数，移动构造函数和移动赋值运算符重载。
- 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。

```
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;
    }*/
```

```
23
24      /*~Person()
25      {}*/
26
27  private:
28      bit::string _name;
29      int _age;
30  };
31
32  int main()
33  {
34      Person s1;
35      Person s2 = s1;
36      Person s3 = std::move(s1);
37      Person s4;
38      s4 = std::move(s2);
39
40      return 0;
41  }
```

5.2 成员变量声明时给缺省值

成员变量声明时给缺省值是给初始化列表用的，如果没有显示在初始化列表初始化，就会在初始化列表用这个却绳子初始化，这个我们在类和对象部分讲过了，忘了就去复习吧。

5.3 defult和delete

- 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。
- 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。

```
1  class Person
2  {
3  public:
4          Person(const char* name = "", int age = 0)
5                  :_name(name)
6                  , _age(age)
7          {}
8
9          Person(const Person& p)
10                 :_name(p._name)
```

```
                ,_age(p._age)
        {}

        Person(Person&& p) = default;

       //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;
}
```

5.4 final与override

这个我们在继承和多态章节已经进行了详细讲过了，忘了就去复习吧。

6. STL中一些变化

- 下图1圈起来的就是STL中的新容器，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。
- STL中容器的新接口也不少，最重要的就是右值引用和移动语义相关的push/insert/emplace系列接口和移动构造和移动赋值，还有initializer_list版本的构造等，这些前面都讲过了，还有一些无关痛痒的如cbegin/cend等需要时查查文档即可。
- 容器的范围for遍历，这个在容器部分也讲过了。

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)

7. lambda

7.1 lambda表达式语法

- `lambda`表达式本质是一个匿名函数对象，跟普通函数不同的是他可以定义在函数内部。`lambda`表达式语法使用层而言没有类型，所以我们一般是用auto或者模板参数定义的对象去接收`lambda`对象。
- lambda表达式的格式：`[capture-list] (parameters)-> return type { function boby }`
- `[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，捕捉列表可以传值和传引用捕捉，具体细节7.2中我们再细讲。捕捉列表为空也不能省略。
- `(parameters)`：参数列表，与普通函数的参数列表功能类似，如果不需要参数传递，则可以连同()一起省略
- `->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。一般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。
- `{function boby}`：函数体，函数体内的实现跟普通函数完全类似，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，函数体为空也不能省略。

```
1 int main()
2 {
3     // 一个简单的lambda表达式
```

```cpp
    auto add1 = [](int x, int y)->int {return x + y; };
    cout << add1(1, 2) << endl;

    // 1、捕捉为空也不能省略
    // 2、参数为空可以省略
    // 3、返回值可以省略，可以通过返回对象自动推导
    // 4、函数题不能省略
    auto func1 = []
    {
            cout << "hello bit" << endl;
            return 0;
    };

    func1();

    int a = 0, b = 1;
    auto swap1 = [](int& x, int& y)
    {
            int tmp = x;
            x = y;
            y = tmp;
    };
    swap1(a, b);
    cout << a << ":" << b << endl;
    return 0;
}
```

7.2 捕捉列表

- `lambda` 表达式中默认只能用 `lambda` 函数体和参数中的变量，如果想用外层作用域中的变量就需要进行捕捉
- 第一种捕捉方式是在捕捉列表中显示的传值捕捉和传引用捕捉，捕捉的多个变量用逗号分割。[x，y， &z] 表示x和y值捕捉，z引用捕捉。
- 第二种捕捉方式是在捕捉列表中隐式捕捉，我们在捕捉列表写一个=表示隐式值捕捉，在捕捉列表写一个&表示隐式引用捕捉，这样我们 `lambda` 表达式中用了那些变量，编译器就会自动捕捉那些变量。
- 第三种捕捉方式是在捕捉列表中混合使用隐式捕捉和显示捕捉。[=, &x]表示其他变量隐式值捕捉，x引用捕捉；[&, x, y]表示其他变量引用捕捉，x和y值捕捉。当使用混合捕捉时，第一个元素必须是&或=，并且&混合捕捉时，后面的捕捉变量必须是值捕捉，同理=混合捕捉时，后面的捕捉变量必须是引用捕捉。

- `lambda`表达式如果在函数局部域中，他可以捕捉`lambda`位置之前定义的变量，不能捕捉静态局部变量和全局变量，静态局部变量和全局变量也不需要捕捉，`lambda`表达式中可以直接使用。这也意味着`lambda`表达式如果定义在全局位置，捕捉列表必须为空。
- 默认情况下，`lambda`捕捉列表是被const修饰的，也就是说传值捕捉的过来的对象不能修改，mutable加在参数列表的后面可以取消其常量性，也就说使用该修饰符后，传值捕捉的对象就可以修改了，但是修改还是形参对象，不会影响实参。使用该修饰符后，参数列表不可省略(即使参数为空)。

```
int x = 0;
// 捕捉列表必须为空，因为全局变量不用捕捉就可以用，没有可被捕捉的变量
auto func1 = []()
{
    x++;
};

int main()
{
    // 只能用当前lambda局部域和捕捉的对象和全局对象
    int a = 0, b = 1, c = 2, d = 3;
    auto func1 = [a, &b]
    {
        // 值捕捉的变量不能修改，引用捕捉的变量可以修改
        //a++;
        b++;
        int ret = a + b;
        return ret;
    };
    cout << func1() << endl;

    // 隐式值捕捉
    // 用了哪些变量就捕捉哪些变量
    auto func2 = [=]
    {
        int ret = a + b + c;
        return ret;
    };
    cout << func2() << endl;

    // 隐式引用捕捉
    // 用了哪些变量就捕捉哪些变量
    auto func3 = [&]
    {
        a++;
        c++;
        d++;
```

```
38      };
39      func3();
40      cout << a <<" "<< b <<" "<< c <<" "<< d <<endl;
41
42      // 混合捕捉1
43      auto func4 = [&, a, b]
44      {
45          //a++;
46          //b++;
47          c++;
48          d++;
49
50          return a + b + c + d;
51      };
52      func4();
53      cout << a << " " << b << " " << c << " " << d << endl;
54
55      // 混合捕捉1
56      auto func5 = [=, &a, &b]
57      {
58          a++;
59          b++;
60          /*c++;
61          d++;*/
62
63          return a + b + c + d;
64      };
65      func5();
66      cout << a << " " << b << " " << c << " " << d << endl;
67
68      // 局部的静态和全局变量不能捕捉，也不需要捕捉
69      static int m = 0;
70
71      auto func6 = []
72      {
73          int ret = x + m;
74          return ret;
75      };
76
77      // 传值捕捉本质是一种拷贝,并且被const修饰了
78      // mutable相当于去掉const属性，可以修改了
79      // 但是修改了不会影响外面被捕捉的值，因为是一种拷贝
80      auto func7 = [=]()mutable
81      {
82          a++;
83          b++;
84          c++;
```

```
        d++;

        return a + b + c + d;
    };
    cout << func7() << endl;
    cout << a << " " << b << " " << c << " " << d << endl;

    return 0;
}
```

7.3 lambda的应用

- 在学习 `lambda` 表达式之前，我们的使用的可调用对象只有函数指针和仿函数对象，函数指针的类型定义起来比较麻烦，仿函数要定义一个类，相对会比较麻烦。使用 `lambda` 去定义可调用对象，既简单又方便。
- `lambda` 在很多其他地方用起来也很好用。比如线程中定义线程的执行函数逻辑，智能指针中定制删除器等， `lambda` 的应用还是很广泛的，以后我们会不断接触到。

```
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 } };

    // 类似这样的场景，我们实现仿函数对象或者函数指针支持商品中
    // 不同项的比较，相对还是比较麻烦的，那么这里lambda就很好用了
    sort(v.begin(), v.end(), ComparePriceLess());
    sort(v.begin(), v.end(), ComparePriceGreater());

    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;
            });

    return 0;
}
```


7.4 lambda的原理

- `lambda` 的原理和范围for很像，编译后从汇编指令层的角度看，压根就没有 `lambda` 和范围for这样的东西。范围for底层是迭代器，而lambda底层是仿函数对象，也就说我们写了一个 `lambda` 以后，编译器会生成一个对应的仿函数的类。
- 仿函数的类名是编译按一定规则生成的，保证不同的 `lambda` 生成的类名不同，lambda参数/返回类型/函数体就是仿函数operator()的参数/返回类型/函数体， `lambda` 的捕捉列表本质是生成的仿函数类的成员变量，也就是说捕捉列表的变量都是 `lambda` 类构造函数的实参，当然隐式捕捉，编译器要看使用哪些就传那些对象。
- 上面的原理，我们可以透过汇编层了解一下，下面第二段汇编层代码印证了上面的原理。

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

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

private:
    double _rate;
};

int main()
{
    double rate = 0.49;

    // lambda
    auto r2 = [rate](double money, int year) {
                    return money * rate * year;
            };

    // 函数对象
    Rate r1(rate);
    r1(10000, 2);

    r2(10000, 2);

    auto func1 = [] {
                    cout << "hello world" << endl;
            };
    func1();

    return 0;
}

```

```cpp
    // lambda
    auto r2 = [rate](double money, int year) {
                    return money * rate * year;
            };
// 捕捉列表的rate，可以看到作为lambda_1类构造函数的参数传递了，这样要拿去初始化成员变量
```

```
// 下面operator()中才能使用
00D8295C  lea         eax,[rate]
00D8295F  push        eax
00D82960  lea         ecx,[r2]
00D82963  call        `main'::`2'::<lambda_1>::<lambda_1> (0D81F80h)

        // 函数对象
        Rate r1(rate);
00D82968  sub         esp,8
00D8296B  movsd       xmm0,mmword ptr [rate]
00D82970  movsd       mmword ptr [esp],xmm0
00D82975  lea         ecx,[r1]
00D82978  call        Rate::Rate (0D81438h)
        r1(10000, 2);
00D8297D  push        2
00D8297F  sub         esp,8
00D82982  movsd       xmm0,mmword ptr [__real@40c3880000000000 (0D89B50h)]
00D8298A  movsd       mmword ptr [esp],xmm0
00D8298F  lea         ecx,[r1]
00D82992  call        Rate::operator() (0D81212h)

// 汇编层可以看到r2 lambda对象调用本质还是调用operator()，类型是lambda_1,这个类型名
// 的规则是编译器自己定制的，保证不同的lambda不冲突
        r2(10000, 2);
00D82999  push        2
00D8299B  sub         esp,8
00D8299E  movsd       xmm0,mmword ptr [__real@40c3880000000000 (0D89B50h)]
00D829A6  movsd       mmword ptr [esp],xmm0
00D829AB  lea         ecx,[r2]
00D829AE  call        `main'::`2'::<lambda_1>::operator() (0D824C0h)
```

8. 包装器

8.1 function

```
template <class T>
class function;     // undefined

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

- `std::function` 是一个类模板，也是一个包装器。`std::function` 的实例对象可以包装存储其他的可以调用对象，包括函数指针、仿函数、`lambda`、`bind` 表达式等，存储的可调用对象被称为 `std::function` 的*目标*。若 `std::function` 不含目标，则称它为空。调用空 `std::function` 的*目标*导致抛出 std::bad_function_call 异常。
- 以上是 `function` 的原型，他被定义<functional>头文件中。std::function - cppreference.com 是function的官方文件链接。
- 函数指针、仿函数、`lambda` 等可调用对象的类型各不相同，`std::function` 的优势就是统一类型，对他们都可以进行包装，这样在很多地方就方便声明可调用对象的类型，下面的第二个代码样例展示了 `std::function` 作为map的参数，实现字符串和可调用对象的映射表功能。

```cpp
#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:
    Plus(int n = 10)
            :_n(n)
    {}

    static int plusi(int a, int b)
    {
            return a + b;
    }

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

private:
```

```
    int _n;
};

int main()
{
    // 包装各种可调用对象
    function<int(int, int)> f1 = f;
    function<int(int, int)> f2 = Functor();
    function<int(int, int)> f3 = [](int a, int b) {return a + b; };

    cout << f1(1, 1) << endl;
    cout << f2(1, 1) << endl;
    cout << f3(1, 1) << endl;

    // 包装静态成员函数
    // 成员函数要指定类域并且前面加&才能获取地址
    function<int(int, int)> f4 = &Plus::plusi;
    cout << f4(1, 1) << endl;

    // 包装普通成员函数
    // 普通成员函数还有一个隐含的this指针参数，所以绑定时传对象或者对象的指针过去都可以
    function<double(Plus*, double, double)> f5 = &Plus::plusd;
    Plus pd;
    cout << f5(&pd, 1.1, 1.1) << endl;

    function<double(Plus, double, double)> f6 = &Plus::plusd;
    cout << f6(pd, 1.1, 1.1) << endl;
    cout << f6(pd, 1.1, 1.1) << endl;

    function<double(Plus&&, double, double)> f7 = &Plus::plusd;
    cout << f7(move(pd), 1.1, 1.1) << endl;
    cout << f7(Plus(), 1.1, 1.1) << endl;

    return 0;
}
```


150. 逆波兰表达式求值 - 力扣（LeetCode）

```
// 传统方式的实现
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
            {
                st.push(stoi(str));
            }
         }

         return st.top();
    }
};

// 使用map映射string和function的方式实现
// 这种方式的最大优势之一是方便扩展，假设还有其他运算，我们增加map中的映射即可
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        stack<int> st;
        // function作为map的映射可调用对象的类型
        map<string, function<int(int, int)>> opFuncMap = {
            {"+", [](int x, int y){return x + y;}},
            {"-", [](int x, int y){return x - y;}},
            {"*", [](int x, int y){return x * y;}},
            {"/", [](int x, int y){return x / y;}}
        };

```

```
        for(auto& str : tokens)
        {
            if(opFuncMap.count(str)) // 操作符
            {
                int right = st.top();
                st.pop();
                int left = st.top();
                st.pop();

                int ret = opFuncMap[str](left, right);
                st.push(ret);
            }
            else
            {
                st.push(stoi(str));
            }
        }

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

8.2 bind

```
simple(1)
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);
```

- `bind` 是一个函数模板，它也是一个可调用对象的包装器，可以把他看做一个函数适配器，对接收的fn可调用对象进行处理后返回一个可调用对象。`bind` 可以用来调整参数个数和参数顺序。`bind` 也在<functional>这个头文件中。
- 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。_1/_2/_3....这些占位符放到placeholders的一个命名空间中。

```cpp
#include<functional>

using placeholders::_1;
using placeholders::_2;
using placeholders::_3;

int Sub(int a, int b)
{
    return (a - b) * 10;
}

int SubX(int a, int b, int c)
{
    return (a - b - c) * 10;
}

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

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

int main()
{
    auto sub1 = bind(Sub, _1, _2);
    cout << sub1(10, 5) << endl;

    // bind 本质返回的一个仿函数对象
    // 调整参数顺序（不常用）
    // _1代表第一个实参
    // _2代表第二个实参
    // ...
    auto sub2 = bind(Sub, _2, _1);
```

```
42      cout << sub2(10, 5) << endl;
43
44      // 调整参数个数 （常用）
45      auto sub3 = bind(Sub, 100, _1);
46      cout << sub3(5) << endl;
47
48      auto sub4 = bind(Sub, _1, 100);
49      cout << sub4(5) << endl;
50
51      // 分别绑死第123个参数
52      auto sub5 = bind(SubX, 100, _1, _2);
53      cout << sub5(5, 1) << endl;
54
55      auto sub6 = bind(SubX, _1, 100, _2);
56      cout << sub6(5, 1) << endl;
57
58      auto sub7 = bind(SubX, _1, _2, 100);
59      cout << sub7(5, 1) << endl;
60
61      // 成员函数对象进行绑死，就不需要每次都传递了
62      function<double(Plus&&, double, double)> f6 = &Plus::plusd;
63      Plus pd;
64      cout << f6(move(pd), 1.1, 1.1) << endl;
65      cout << f6(Plus(), 1.1, 1.1) << endl;
66
67      // bind一般用于，绑死一些固定参数
68      function<double(double, double)> f7 = bind(&Plus::plusd, Plus(), _1, _2);
69      cout << f7(1.1, 1.1) << endl;
70
71
72      // 计算复利的lambda
73      auto func1 = [](double rate, double money, int year)->double {
74              double ret = money;
75              for (int i = 0; i < year; i++)
76              {
77                      ret += ret * rate;
78              }
79
80              return ret - money;
81      };
82
83      // 绑死一些参数，实现出支持不同年华利率，不同金额和不同年份计算出复利的结算利息
84      function<double(double)> func3_1_5 = bind(func1, 0.015, _1, 3);
85      function<double(double)> func5_1_5 = bind(func1, 0.015, _1, 5);
86      function<double(double)> func10_2_5 = bind(func1, 0.025, _1, 10);
87      function<double(double)> func20_3_5 = bind(func1, 0.035, _1, 30);
88
```

```
89      cout << func3_1_5(1000000) << endl;
90      cout << func5_1_5(1000000) << endl;
91      cout << func10_2_5(1000000) << endl;
92      cout << func20_3_5(1000000) << endl;
93
94      return 0;
95  }
```

9. 智能指针

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