10 C++11

使用GNU C++编译器（g++）编译包含C++11语法特性的代码，需要加上“-std=c++11”编译选项。例如：

```
g++ -std=c++11 hello.cpp
```

10.1 简化代码

10.1.1 类型推导

10.1.1.1 auto

10.1.1.1.1 auto的基本用法

```
// auto1.cpp

// auto的基本用法

#include <iostream>
#include <typeinfo>

using namespace std;

int main(void) {
    auto a = 1; // a:int，常限定被丢弃
    cout << typeid(a).name() << endl; // i

    auto b = new auto(2); // b:int*
    cout << typeid(b).name() << endl; // Pi

    auto const* c = &a, d = 3; // c:int const*, d:int const
    cout << typeid(c).name() << endl; // PKi
    cout << typeid(d).name() << endl; // i
    // cout << ++d << endl; // 编译错误，d带有常限定

    // auto const* e = &a, f; // 编译错误，虽然根据e的初值&a已经可以推导出auto，但
                              // f仍然需要被显式初始化，且不能与之前的推断发生矛盾
    // auto const* g = &a, h = 3.0; // 编译错误，类型推导不能有二义性

    // 在旧语法中，auto变量即自动变量存储于栈区，而static变量即静态变量存储于
    // 静态区，二者不可同时使用，但C++11的auto关键字已不再作为存储类型指示符

    static auto i = 4.;
    cout << typeid(i).name() << endl; // d

    // auto int j; // 编译错误，C++11的auto关键字已不再作为存储类型指示符

    // auto k; // 编译错误，auto并不能代表一个实际的类型声明，它只是一个类
               // 型声明的占位符。使用auto声明的变量必须马上初始化，以让编
               // 译器推断出它的实际类型，并在编译时将auto替换为真正的类型

    return 0;
}
```

10.1.1.1.2 auto同指针或引用结合使用

```cpp
// auto2.cpp

// auto同指针或引用结合使用

#include <iostream>
#include <typeinfo>

using namespace std;

int main(void) {
    int a = 0;

    auto* b = &a; // auto=int, b:int*
    cout << typeid(b).name() << endl; // Pi

    // 即使不把auto声明为指针，其亦可被推导为指针类型

    auto c = &a; // auto=int*, c:int*
    cout << typeid(c).name() << endl; // Pi

    // 当表达式带有引用属性时，auto会抛弃其引用属性，直接推导为原始类型

    auto& d = a; // auto=int, d:int&
    cout << &d << ' ' << &a << endl; // 0x7ffc19fd2540 0x7ffc19fd2540
    auto e = d; // auto=int, e:int
    cout << &e << ' ' << &d << endl; // 0x7ffc19fd2544 0x7ffc19fd2540

    // 当表达式带有CV（只读/挥发）限定时，auto会抛弃其CV限定

    auto const f = a; // auto=int, f:int const
    // cout << ++f << endl; // 编译错误，f带有常限定
    auto g = f; // auto=int, g:int
    cout << ++g << endl; // 1

    // 但如果auto和引用或指针结合使用，表达式的CV限定会被保留下来

    auto const& h = a; // auto=int, h:int const&
    // cout << ++h << endl; // 编译错误，h带有常限定
    auto& i = h; // auto=int const, i:int const&
    // cout << ++i << endl; // 编译错误，i带有常限定
    auto* j = &h; // auto=int const, j:int const*
    // cout << ++*j << endl; // 编译错误，j带有常限定

    return 0;
}
```

10.1.1.1.3 auto使用的限制

```cpp
// auto3.cpp

// auto使用的限制

#include <iostream>
```

```cpp
#include <typeinfo>

using namespace std;

// auto不能用于函数的参数

// void foo(auto a = 0) {
//     cout << typeid(a).name() << endl;
// }

class A {
public:
    int x = 0;

    // auto不能用于类的非静态成员变量

    // auto y = 1;

    static auto const z = 2;
};

template<typename T>
class B {
public:
    B(T const& arg) : var(arg) {}

    T var;
};

int main (void) {
    // auto不能用于函数的参数

    // foo(1);

    // auto不能用于类的非静态成员变量

    A a;
    cout << a.x << ' ' /*<< a.y << ' ' */<< a.z << endl; // 0 2

    // auto不能用于模板的类型实参

    B<int> b(0);
    // B<auto> c = b;

    // auto不能用于数组元素

    int d[10];
    // auto e[10] = d;

    // 但可用于指向数组元素的指针和引用整个数组的引用

    auto f = d; // auto=int*, arr3:int*, arr1代表数组首元素的地址
    cout << typeid(f).name() << endl; // Pi
    auto& g = d; // auto=int[10], arr4:int (&)[10], arr1代表数组整体
    cout << typeid(g).name() << endl; // A10_i

```

```
    return 0;
}
```

10.1.1.1.4 何时使用auto?

```
// auto4.cpp

// 何时使用auto?

#include <iostream>
#include <map>

using namespace std;

// 借助auto，可以减少模板的类型参数

class A {
public:
    A(int arg = 0) : var(arg) {}

    int get(void) const {
        return var;
    }

    void set(int arg) {
        var = arg;
    }

private:
    int var;
};

class B {
public:
    B(char const* arg = "") : var(arg) {}

    char const* get(void) const {
        return var;
    }

    void set(char const* arg) {
        var = arg;
    }

private:
    char const* var;
};

template<typename V, typename X>
void foo(X const& x) {
    V var = x.get();
    cout << var << endl;

    // 使用var...
}
```

```cpp

template<typename X>
void bar(X const& x) {
    auto var = x.get();
    cout << var << endl;

    // 使用var...
}

int main(void) {
    // 借助auto，可以减少模板的类型参数

    A a(1234);
    B b("abcd");

    foo<int>(a); // 1234
    foo<char const*>(b); // abcd

    bar(a); // 1234
    bar(b); // abcd

    // 借助auto，可以简化复杂类型的书写

    multimap<string, int> sn;

    sn.insert(make_pair("张飞", 100000));
    sn.insert(make_pair("赵云", 150000));
    sn.insert(make_pair("张飞", 200000));
    sn.insert(make_pair("关羽", 250000));
    sn.insert(make_pair("赵云", 300000));
    sn.insert(make_pair("关羽", 350000));

    pair<multimap<string, int>::iterator,
        multimap<string, int>::iterator> range1 = sn.equal_range("张飞");
    int sum1 = 0;
    for (multimap<string, int>::iterator it = range1.first;
        it != range1.second; ++it)
        sum1 += it->second;
    cout << sum1 << endl; // 300000

    auto range2 = sn.equal_range("张飞");
    int sum2 = 0;
    for (auto it = range2.first; it != range2.second; ++it)
        sum2 += it->second;
    cout << sum2 << endl; // 300000

    return 0;
}
```

10.1.1.2 decltype

10.1.1.2.1 计算表达式的类型

```cpp
// decltype1.cpp

// 计算表达式的类型

#include <iostream>
#include <typeinfo>

using namespace std;

int main(void) {
    int a = 0;

    // decltype的值就是表达式的类型本身

    decltype(a) b = 1; // b:int
    cout << typeid(b).name() << endl; // i

    // decltype在编译时而非运行时计算表达式的类型

    decltype(++a) c = a; // c:int&
    cout << a << endl; // 0

    // decltype会保留表达式的引用属性和CV限定

    int const& d = 3;
    decltype(d) e = d; // e:int const&
    cout << &e << ' ' << &d << endl; // 0x7ffcf815f3dc 0x7ffcf815f3dc
    // cout << ++e << endl; // 编译错误，e带有常限定

    // decltype可以和指针联用

    decltype(a) *f = &a, **g = &f; // f:int*, g:int**
    cout << typeid(f).name() << endl; // Pi
    cout << typeid(g).name() << endl; // PPi

    // decltype可以和引用及CV限定符联用

    decltype(a) const& h = a; // h:int const&
    cout << &h << ' ' << &a << endl; // 0x7ffcf815f3d8 0x7ffcf815f3d8
    // cout << ++h << endl; // 编译错误，h带有常限定

    return 0;
}
```

10.1.1.2.2 类型计算规则

```cpp
// decltype2.cpp

// 类型计算规则

#include <iostream>
```

```cpp

using namespace std;

class Obj {
public:
    Obj(int arg) : var(arg) {}

    int var;

    static int const sta = 0;
};

ostream& operator<< (ostream& os, Obj const& obj) {
    return os << '(' << obj.var << ',' << obj.sta << ')';
}

// 规则1：对标识符及成员标识符表达式，直接取表达式的类型

void rule1(void) {
    int a = 0;
    int const volatile& b = a;

    decltype(a) c = a; // c:int
    decltype(b) d = b; // d:int const volatile&

    Obj e = 1;
    decltype(e.var) f = 2; // f:int
    decltype(Obj::sta) g = 3; // g:int const

    cout << a << ' ' << b << ' ' << c << ' ' << d << ' ' <<
            e << ' ' << f << ' ' << g << endl;
}

// 规则2：对函数调用表达式，取函数返回值的类型

void rule2(void) {
    int&  lvr(void); // 返回左值引用
    int&& rvr(void); // 返回右值引用
    int   prv(void); // 返回纯右值
    Obj   obj(void); // 返回类对象

    int const&  clvr(void); // 返回常左值引用
    int const&& crvr(void); // 返回常右值引用
    int const   cprv(void); // 返回常纯右值
    Obj const   cobj(void); // 返回常类对象

    int i = 0;
    decltype(lvr()) a = i; // a:int&
    decltype(rvr()) b = 1; // b:int&&
    decltype(prv()) c = 2; // c:int
    decltype(obj()) d = 3; // d:Obj

    int j = 4;
    decltype(clvr())e = j; // e:int const&
    decltype(crvr())f = 5; // f:int const&&
    decltype(cprv())g = 6; // g:int, 基本类型返回值忽略CV限定
```

```
    decltype(cobj())h = 7; // h:Obj const, 类类型返回值CV限定有效

    cout << a << ' ' << b << ' ' << c << ' ' << d << ' ' <<
            e << ' ' << f << ' ' << g << ' ' << h << endl;
}

// 规则3：对其它表达式，若表达式的类型为左值则取该类型的左值引用，否则取表达式的类型

void rule3(void) {
    int a = 0;

    decltype(a) b = 1; // b:int, 根据规则1
    decltype((a)) c = a; // c : int&, a是左值，因此括号表达式(a)
                         // 也是左值，由规则3，c的类型为左值引用

    Obj const d = 2;
    decltype(d.var) e = 3; // e:int, 根据规则1
    decltype((d.var)) f = d.var; // f:int const&, 常对象的成员变量都是常
                                 // 左值，由规则3，f的类型为常左值引用

    int g = 4, h = 5;
    decltype(++g)   i = g; // i:int&
    decltype(g++)   j = 6; // j:int
    decltype(g = h) k = g; // k:int&
    decltype(g + h) l = 7; // l:int

    cout << a << ' ' << b << ' ' << c << ' ' << d << ' ' <<
            e << ' ' << f << ' ' << g << ' ' << h << ' ' <<
            i << ' ' << j << ' ' << k << ' ' << l << endl;
}

int main(void) {
    rule1();
    rule2();
    rule3();

    return 0;
}
```

10.1.1.2.3 何时使用decltype?

```
// decltype3.cpp

// 何时使用decltype?

#include <iostream>
#include <vector>
#include <map>

using namespace std;

// 打印可写容器中的元素

template<typename C>
void printWritable(C& c) {
```

```
    cout << "可写容器: ";

    for (typename C::iterator it = c.begin(); it != c.end(); ++it)
        cout << *it << ' ';

    cout << endl;
}

// 打印只读容器中的元素

template<typename C>
void printReadonly(C const& c) {
    cout << "只读容器: ";

    for (typename C::const_iterator it = c.begin(); it != c.end(); ++it)
        cout << *it << ' ';

    cout << endl;
}

// 打印任意容器中的元素

template<typename C>
void print(C& c) {
    cout << "任意容器: ";

    for (decltype(c.begin()) it = c.begin(); it != c.end(); ++it)
        cout << *it << ' ';

    cout << endl;
}

int main(void) {
    vector<int> wv; // 可写容器
    wv.push_back(10);
    wv.push_back(20);
    wv.push_back(30);
    wv.push_back(40);
    wv.push_back(50);
    vector<int> const rv = wv; // 只读容器

    printWritable(wv);
    printReadonly(rv);

    print(wv);
    print(rv);

    map<string, vector<int> > scores;
    scores["张飞"].push_back(70);
    scores["张飞"].push_back(85);
    scores["赵云"].push_back(75);
    scores["赵云"].push_back(90);
    scores["关羽"].push_back(80);
    scores["关羽"].push_back(95);

    // ...
```

```
71
72      int sum1 = 0; // 在冗长的代码中确定某个变量的类型将十分困难
73      for (size_t i = 0; i < scores["张飞"].size(); ++i)
74          sum1 += scores["张飞"][i];
75      cout << sum1 << endl; // 155
76
77      decltype(scores)::mapped_type::value_type sum2 = // 通过decltype可以很方便
    地
78          decltype(scores)::mapped_type::value_type(); // 根据变量表达式计算其类型
79      for (size_t i = 0; i < scores["张飞"].size(); ++i)
80          sum2 += scores["张飞"][i];
81      cout << sum2 << endl; // 155
82
83      return 0;
84  }
```

10.1.1.2.4 返回类型后置

```
1   // decltype4.cpp
2
3   // 返回类型后置
4
5   #include <iostream>
6   #include <typeinfo>
7
8   using namespace std;
9
10  template<typename X, typename Y>
11  auto add(X const& x, Y const& y) -> decltype(x + y) {
12      return x + y;
13  }
14
15  double foo(int arg) {
16      return arg / 2.0;
17  }
18
19  int foo(double arg) {
20      return int(arg * 2);
21  }
22
23  template<typename T>
24  auto bar(T const& arg) -> decltype(foo(arg)) {
25      return foo(arg);
26  }
27
28  int main(void) {
29      auto a = add(1, 0.5);
30      cout << typeid(a).name() << ' ' << a << endl; // d 1.5
31
32      auto b = add (1, 'A');
33      cout << typeid(b).name() << ' ' << b << endl; // i 66
34
35      auto c = bar(1);
36      cout << typeid(c).name() << ' ' << c << endl; // d 0.5
37
```

```
    auto d = bar(0.5);
    cout << typeid(d).name() << ' ' << d << endl; // i 1

    return 0;
}
```

10.1.2 模板改进

10.1.2.1 为模板定义别名

10.1.2.1.1 不能用typedef为模板定义别名

```
// alias1.cpp

// 不能用typedef为模板定义别名

#include <iostream>
#include <typeinfo>

using namespace std;

template<typename A, typename B>
class X {};

// 可以用typedef为具体类型定义别名

typedef unsigned int uint_t;

// 当然也包括由模板实例化产生的类

typedef X<string, int> xsi_t;

// 但不能用typedef为模板或部分模板定义别名

// typedef X x_t;
// typedef X<string, B> xs_t;

int main(void) {
    uint_t a; // a:unsigned int
    xsi_t b; // b:X<string, int>
    // x_t<string, int> c;
    // xs_t<int> d;

    cout << typeid(a).name() << endl;
    cout << typeid(b).name() << endl;
    // cout << typeid(c).name() << endl;
    // cout << typeid(d).name() << endl;

    return 0;
}
```

10.1.2.1.2 可以用using为包括模板在内的任何类型定义别名

```cpp
// alias2.cpp

// 可以用using为包括模板在内的任何类型定义别名

#include <iostream>
#include <typeinfo>

using namespace std;

template<typename A, typename B>
class X {};

// 可以用using为具体类型定义别名

using uint_t = unsigned int;

// 当然也包括由模板实例化产生的类

using xsi_t = X<string, int>;

// 甚至可以用using为模板或部分模板定义别名

template<typename A, typename B> using x_t = X<A, B>;
template<typename B> using xs_t = X<string, B>;

int main(void) {
    uint_t a; // a:unsigned int
    xsi_t b; // b:X<string, int>
    x_t<string, int> c; // c:X<string, int>
    xs_t<int> d; // d:X<string, int>

    cout << typeid(a).name() << endl;
    cout << typeid(b).name() << endl;
    cout << typeid(c).name() << endl;
    cout << typeid(d).name() << endl;

    return 0;
}
```

10.1.2.2 函数模板的缺省模板参数

10.1.2.2.1 C++98/03中的函数模板不能带有缺省模板参数

在C++98/03中，只有类模板可以带有缺省模板参数，函数模板不能带有缺省模板参数。

```cpp
// defparam1.cpp

// C++98/03中的函数模板不能带有缺省模板参数

#include <iostream>
#include <typeinfo>

using namespace std;
```

```cpp

// 在C++98/03中，只有类模板可以带有缺省模板参数

template<typename A = int, typename B = double, typename C = string>
class X {
public:
    static void foo(void) {
        cout << typeid(A).name() << ' ' << typeid(B).name() << ' ' << typeid(C).name() << endl;
    }
};

// 在C++98/03中，函数模板不能带有缺省模板参数

// template<typename A = int, typename B = double, typename C = string>
// void foo(void) {
//     cout << typeid(A).name() << ' ' << typeid(B).name() << ' ' << typeid(C).name() << endl;
// }

int main(void) {
    // 在C++98/03中，只有类模板可以带有缺省模板参数

    X<char, short, long>::foo();
    X<char, short>::foo();
    X<char>::foo();
    X<>::foo(); // "<>"不可省略

    // 在C++98/03中，函数模板不能带有缺省模板参数

    // foo<char, short, long>();
    // foo<char, short>();
    // foo<char>();
    // foo<>();
    // foo();

    return 0;
}

// g++ -std=c++98 defparam1.cpp
```

10.1.2.2.2 C++11中的类模板和函数模板都可以带有缺省模板参数

```cpp
// defparam2.cpp

// C++11中的类模板和函数模板都可以带有缺省模板参数

#include <iostream>
#include <typeinfo>

using namespace std;

// 在C++11中，类模板可以带有缺省模板参数

template<typename A = int, typename B = double, typename C = string>
```

```cpp
class X {
public:
    static void foo(void) {
        cout << typeid(A).name() << ' ' << typeid(B).name() << ' ' << typeid(C).name() << endl;
    }
};

// 在C++11中，函数模板也可以带有缺省模板参数

template<typename A = int, typename B = double, typename C = string>
void foo(void) {
    cout << typeid(A).name() << ' ' << typeid(B).name() << ' ' << typeid(C).name() << endl;
}

// 只要满足隐式推断的条件，函数模板参数的缺省值不是非写在参数表最后

template<typename A = int, typename B, typename C = string>
A bar(B b) {
    cout << typeid(A).name() << ' ' << typeid(B).name() << ' ' << typeid(C).name() << endl;

    A a = A();
    C c = C();

    return a;
}

// 无法隐式推断的模板参数取缺省值，否则取隐式推断的
// 类型。只要隐式推断生效，模板参数的缺省值即被忽略

template<typename A = int, typename B = double, typename C = string>
A hum(B b) {
    cout << typeid(A).name() << ' ' << typeid(B).name() << ' ' << typeid(C).name() << endl;

    A a = A();
    C c = C();

    return a;
}

int main(void) {
    // 在C++11中，类模板可以带有缺省模板参数

    X<char, short, long>::foo();
    X<char, short>::foo();
    X<char>::foo();
    X<>::foo(); // "<>"不可省略

    // 在C++11中，函数模板也可以带有缺省模板参数

    foo<char, short, long>();
    foo<char, short>();
    foo<char>();
```

```
    foo<>();
    foo(); // "<>"可以省略

    // 只要满足隐式推断的条件，函数模板参数的缺省值不是非写在参数表最后

    bar<char, short, long>(100);
    bar<char, short>(100);
    bar<char>(100); // 隐式推断B=int
    bar<>(100); // 隐式推断B=int
    bar(100); // 隐式推断B=int

    // 无法隐式推断的模板参数取缺省值，否则取隐式推断的
    // 类型。只要隐式推断生效，模板参数的缺省值即被忽略

    hum(100); // 隐式推断B=int，忽略其缺省值double

    return 0;
}
```

10.1.2.3 模板实参表的右尖括号

10.1.2.3.1 C++98/03标准模板实参表的右尖括号

在C++98/03标准中，连续的两个右尖括号会被编译器解释为右移位操作符，而非模板实参表结束，因此它们之间至少应保留一个空格。

```
// delimiter1.cpp

// C++98/03标准模板实参表的右尖括号

#include <iostream>
#include <typeinfo>

using namespace std;

template<typename T>
void ptype(void) {
    cout << typeid(T).name() << endl;
}

template<typename T>
class Dummy {};

template<int x>
int square(void) {
    return x * x;
}

int main(void) {
    // 在C++98/03标准中，连续的两个右尖括号会被编译器解释为右移位
    // 操作符，而非模板实参表结束，因此它们之间至少应保留一个空格

    // ptype<Dummy<int>>(); // 编译错误
    ptype<Dummy<int> >();

```

```
    // 这里的">>"会被正确解释为右移位操作符

    cout << square<6>>1>() << endl; // 9

    return 0;
}

// g++ -std=c++98 delimiter1.cpp
```

10.1.2.3.2 C++11标准模板实参表的右尖括号

在C++11标准中，要求编译器对模板的右尖括号做单独处理，使之能正确区分右移位操作符和模板实参表结束，因此无需再留空格。

```
// delimiter2.cpp

// C++11标准模板实参表的右尖括号

#include <iostream>
#include <typeinfo>

using namespace std;

template<typename T>
void ptype(void) {
    cout << typeid(T).name() << endl;
}

template<typename T>
class Dummy {};

template<int x>
int square(void) {
    return x * x;
}

int main(void) {
    // 在C++11标准中，要求编译器对模板的右尖括号做单独处理，使之
    // 能正确区分右移位操作符和模板实参表结束，因此无需再留空格

    ptype<Dummy<int>>();

    // 如果不加小括号，这里的">>"会被错误解释为模板参数表结束和大于号，导致编译错误

    // cout << square<6>>1>() << endl; // 编译错误
    cout << square<(6>>1)> () << endl; // 9

    return 0;
}
```

10.1.3 列表初始化

10.1.3.1 C++98/03标准没有通用的初始化语法

在C++98/03中，对象初始化方法有很多种，每一种都有各自的适用范围和作用。没有任何一种初始化方法可以通用于所有情况。

```cpp
// initlist1.cpp

// C++98/03标准没有通用的初始化语法

#include <iostream>
#include <algorithm>
#include <iterator>

using namespace std;

struct Student {
    char name[256];

    struct Date {int y, m, d;} bday;
};

class Complex {
public:
    Complex(double r = 0, double i = 0) : r(r), i(i) {}

    // 有公有拷贝构造函数，不一定会被调用

    Complex(Complex const& complex) : r(complex.r), i(complex.i) {
        cout << "拷贝构造: " << &complex << "->" << this << endl;
    }

    friend ostream& operator<<(ostream& os, Complex const& complex) {
        return os << '(' << complex.r << showpos << complex.i << noshowpos
<< "i)";
    }

private:
    // 无公有拷贝构造函数，编译器一定报错

    // Complex(Complex const& complex);

    double r, i;
};

int main(void) {
    int a = 123; // 赋值形式的初始化
    cout << a << endl;

    double b(1.23); // 构造形式的初始化
    cout << b << endl;

    int c[] = {100, 200, 300}; // 列表形式的初始化
    copy(c, c + sizeof(c) / sizeof(c[0]), ostream_iterator<int>(cout, " "));
```

```
    cout << endl;

    Student d = {"张飞", {2011, 11, 11}}; // 列表形式的初始化
    cout << d.name << ',' << d.bday.y << '-' << d.bday.m << '-' << d.bday.d
<< endl;

    Complex e(1.2, 3.4); // 构造形式的初始化
    cout << e << endl;

    // 对拷贝构造函数的调用，可能会被优化掉，但对拷贝构造函数的语法检查，编译器依然会做

    Complex f = Complex(1.2, 3.4); // 赋值形式的初始化
    cout << f << endl;

    return 0;
}

// g++ -std=c++98 initlist1.cpp
```

10.1.3.2 C++11标准提供了通用的列表形式的初始化语法

在C++11中，列表形式的初始化语法被大大扩充了，可用于任何类型对象及其数组的初始化，无论该对象或数组是被静态还是动态创建的。

```
// initlist2.cpp

// C++11标准提供了通用的列表形式的初始化语法

#include <iostream>
#include <algorithm>
#include <iterator>

using namespace std;

struct Student {
    char name[256];

    struct Date {int y, m, d;} bday;
};

class Complex {
public:
    Complex(double r = 0, double i = 0) : r(r), i(i) {}

    Complex const operator+(Complex const& complex) const {
        // 列表形式的初始化语法甚至可以用在函数的返回值上

        return {r + complex.r, i + complex.i};
    }

    friend ostream& operator<< (ostream& os, Complex const& complex) {
        return os << '(' << complex.r << showpos << complex.i << noshowpos
<< "i)";
    }

```

```cpp
private:
    // 无公有拷贝构造函数，编译器并不报错

    Complex(Complex const& complex);

    double r, i;
};

int main(void) {
    int a  = {123};
    cout << a << endl;

    double b{1.23}; // 左花括号前面的等号写不写都一样
    cout << b << endl;

    int c[]{100, 200, 300};
    copy(c, c + sizeof(c) / sizeof(c[0]), ostream_iterator<decltype(*c)>(cout, " "));
    cout << endl;

    Student d{"张飞", {2011, 11, 11}};
    cout << d.name << ',' << d.bday.y << '-' << d.bday.m << '-' << d.bday.d << endl;

    Complex e{1.2, 3.4};
    cout << e << endl;

    Complex f = {1.2, 3.4}; // 左花括号前面有没有等号，都不需要拷贝构造
    cout << f << endl;

    cout << e + f << endl;

    Complex* g {new Complex{1.2, 3.4}};
    cout << *g << endl;
    delete g;

    g = new Complex[3]{{1.1, 2.2}, {3.3, 4.4}, {5.5, 6.6}};
    copy(g, g + 3, ostream_iterator<decltype(*g)>(cout, " "));
    cout << endl;
    delete[] g;

    cout << Complex{1.2, 3.4} << endl; // 匿名对象

    Complex const (&h)[3]{{1.1, 2.2}, {3.3, 4.4}, {5.5, 6.6}}; // 匿名数组
    copy(h, h + 3, ostream_iterator<decltype(*h)>(cout, " "));
    cout << endl;

    return 0;
}
```

10.1.3.3 聚合类型与非聚合类型的列表初始化

何为聚合类型?

- 任意类型（聚合类型或非聚合类型）的数组
- 满足以下条件的类:
    - 无自定义构造函数
    - 无私有或保护的非静态成员变量
    - 无基类
    - 无虚函数
    - 无通过“=”或“{}”在声明的同时初始化的非静态成员变量
- 聚合类型的元素可以是聚合类型的也可以是非聚合类型的

对聚合类型使用列表初始化，相当于对其中的元素逐一初始化，而对非聚合类型使用列表初始化，则需要调用匹配的构造函数。

```cpp
// initlist3.cpp

// 聚合类型与非聚合类型的列表初始化

#include <iostream>

using namespace std;

// 非聚合类型基类

class Base {
public:
    Base(int n) : n(n) { // 自定义构造函数
        cout << "Base构造: " << this << endl;
    }

    virtual ostream& operator>>(ostream& os) const { // 虚函数
        return os << n;
    }

private:
    int n; // 私有的非静态成员变量
};

// 非聚合类型子类

class Nonagg : public Base { // 有基类
public:
    Nonagg(int n, double d) : Base(n), d(d) { // 自定义构造函数
        cout << "Nonagg构造: " << this << endl;
    }

    ostream& operator>>(ostream& os) const { // 虚函数
        return Base::operator>>(os) << ' ' << d << ' ' << s;
    }

    char const* s = "ABC"; // 通过“=”在声明的同时初始化的非静态成员变量
```

```
38
39  protected:
40      double d; // 保护的非静态成员变量
41  };
42
43  // 聚合类型
44  class Aggreg {
45  public:
46      ostream& operator>>(ostream& os) const {
47          return nonagg >> os << ' ' << n << ' ' << d << ' ' << s;
48      }
49
50      Nonagg nonagg; // 聚合类型可以包含非聚合类型的元素
51      int n;
52      double d;
53      char const* s;
54  };
55
56  int main(void) {
57      Nonagg nonagg{12, 3.4};
58
59      Base& base = nonagg;
60      base >> cout << endl;
61
62      Aggreg aggreg{{12, 3.4}, 56, 7.8, "DEF"};
63      aggreg >> cout << endl;
64
65      return 0;
66  }
```

10.1.3.4 变长初始化列表

```
1   // initlist4.cpp
2
3   // 变长初始化列表
4
5   #include <iostream>
6   #include <algorithm>
7   #include <iterator>
8   #include <vector>
9   #include <map>
10
11  using namespace std;
12
13  class Students {
14      using vs = vector<char const*>;
15      using sn = map<char const*, int>;
16
17  public:
18      // 令构造函数接受initializer_list参数，以支持变长初始化列表
19
20      Students(initializer_list<vs::value_type> names) {
21          for (auto it = names.begin(); it != names.end(); ++it)
22              this->names.push_back(*it);
23      }
```

```
24
25      Students(initializer_list<sn::value_type> scores) {
26          for (auto it = scores.begin(); it != scores.end(); ++it)
27              this->scores.insert(*it);
28      }
29
30      void pnames(void) const {
31          copy(names.begin(), names.end(),
ostream_iterator<decltype(*names.begin())>(cout, " "));
32          cout << endl;
33      }
34
35      void pscores(void) const {
36          for (auto it = scores.begin(); it != scores.end(); ++it)
37              cout << it->first << ':' << it->second << endl;
38      }
39
40  private:
41      vs names;
42      sn scores;
43  };
44
45  // initializer_list不仅用于构造函数，任何需要传递变长同类型数据集的场合都可以使用
46
47  int average(initializer_list<int> scores) {
48      int ave = 0;
49
50      for (auto it = scores.begin(); it != scores.end(); ++it)
51          ave += *it;
52
53      if (scores.size())
54          ave /= scores.size();
55
56      return ave;
57  }
58
59  int main (void) {
60      // 用于数组的初始化列表，其长度可以是不固定的
61
62      char const* a[]{"张飞", "赵云", "关羽", "黄忠", "马超"};
63      copy(a, a + sizeof(a) / sizeof(a[0]), ostream_iterator<decltype(a[0])>
(cout, " "));
64      cout << endl;
65
66      // 用于标准容器的初始化列表，其长度同样可以是不固定的
67
68      vector<char const*> b{"张飞", "赵云", "关羽", "黄忠", "马超"};
69      copy(b.begin(), b.end(), ostream_iterator<decltype(b[0])>(cout, " "));
70      cout << endl;
71
72      map<char const*, int> c{
73          {"张飞", 50}, {"赵云", 60}, {"关羽", 70}, {"黄忠", 80}, {"马超", 90}};
74      for (auto it = c.begin(); it != c.end(); ++it)
75          cout << it->first << ':' << it->second << endl;
76
77      // 支持变长初始化列表的自定义类
```

```
78
79      Students d{"张飞", "赵云", "关羽", "黄忠", "马超"};
80      d.pnames();
81
82      Students e{
83          {"张飞", 50}, {"赵云", 60}, {"关羽", 70}, {"黄忠", 80}, {"马超", 90}};
84      e.pscores();
85
86      // 支持变长参数列表的自定义函数
87
88      cout << average({}) << endl;
89
90      int f = 60, g = 70, h = 80;
91      cout << average({50, f, g, h, 90}) << endl;
92
93      return 0;
94  }
```

10.1.3.5 initializer_list轻量级类模板容器

```
1   // initlist5.cpp
2
3   // initializer_list轻量级类模板容器
4
5   #include <iostream>
6   #include <algorithm>
7   #include <iterator>
8   #include <list>
9
10  using namespace std;
11
12  // 轻量级容器内部存放初始化列表元素的引用而非其拷贝
13
14  initializer_list<int> light(void) {
15      int a = 1000, b = 2000, c = 3000;
16      return {a, b, c}; // 所返回局部变量的引用将在函数返回后失效
17  }
18
19  // 重量级容器内部存放初始化列表元素的拷贝而非其引用
20
21  list<int> heavy(void) {
22      int a = 1000, b = 2000, c = 3000;
23      return {a, b, c}; // 所返回局部变量的拷贝在函数返回后依然有效
24  }
25
26  int main(void) {
27      // 可以接受任意长度的初始化列表（{...}），但列表中元素的类型必须相同
28
29      initializer_list<int> in{10, 20, 30, 40, 50};
30
31      // 只有三个公有成员函数：begin()，end()和size()
32
33      copy(in.begin(), in.end(), ostream_iterator<decltype(*in.begin())>(cout,
    " "));
34      cout << '[' << in.size() << ']' << endl;
```

```
35
36      // 迭代器为只读类型，其目标元素不可修改
37
38      // for (auto it = in.begin(); it != in.end(); ++it)
39      //     *it *= 100;
40
41      // 但可对容器整体赋值
42
43      in = {1000, 2000, 3000};
44      copy(in.begin(), in.end(), ostream_iterator<decltype(*in.begin())>(cout,
    " "));
45      cout << '[' << in.size() << ']' << endl;
46
47      // 提供缺省构造函数，用于实例化空容器
48
49      in = {};
50      copy(in.begin(), in.end(), ostream_iterator<decltype(*in.begin())>(cout,
    " "));
51      cout << '[' << in.size() << ']' << endl;
52
53      // 轻量级容器内部存放初始化列表元素的引用而非其拷贝
54
55      in = light();
56      copy(in.begin(), in.end(), ostream_iterator<decltype(*in.begin())>(cout,
    " "));
57      cout << '[' << in.size() << ']' << endl;
58
59      // 重量级容器内部存放初始化列表元素的拷贝而非其引用
60
61      list<int> ln = heavy();
62      copy(ln.begin(), ln.end(), ostream_iterator<decltype(*ln.begin())>(cout,
    " "));
63      cout << '[' << ln.size()<< ']' << endl;
64
65      return 0;
66  }
```

10.1.3.6 借助列表初始化防止类型收窄

```
1   // initlist6.cpp
2
3   // 利用列表初始化防止类型收窄
4
5   #include <iostream>
6
7   using namespace std;
8
9   int main(void) {
10      // 常量及带有常限定的变量，根据值判定是否收窄，有信息损失则报错
11
12      int a = 1.2;
13      int b{1.2}; // 错误：从double到int需要收窄转换
14
15      float c = 1e40;
16      float d{1e40}; // 错误：从double到float需要收窄转换
```

```
    float e = 1.2;
    float f{1.2};

    float g = (unsigned long long)-1;
    float h{(unsigned long long)-1}; // 错误：从unsigned long long到float需要收窄转换

    float i = (unsigned long long)1;
    float j{(unsigned long long)1};

    int const k = 128, l = 1;

    char m = k;
    char n{k}; // 错误：从int转换到char需要收窄转换

    char o = l;
    char p{l};

    // 变量，根据类型判定是否收窄，有信息损失的风险即给出警告

    int q = 128, r = 1;

    char s = q;
    char t{q}; // 警告：从int到char需要收窄转换

    char u = r;
    char v{r}; // 警告：从int到char需要收窄转换

    return 0;
}
```

10.1.4 范围循环

10.1.4.1 基于范围的for循环

```
// for1.cpp

// 基于范围的for循环

#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

void print(int i) {
    cout << i << ' ';
}

int main(void) {
    int an[]{65, 66, 67, 68, 69};
    size_t size = sizeof(an) / sizeof(an[0]);
    vector<int> vn(an, an + size);

```

```
    // 基于下标运算的for循环，不是所有的容器都支持下标运算，不通用

    for (size_t i = 0; i < size; ++i)
        cout << an[i] << ' ';
    cout << endl;

    for (size_t i = 0; i < size; ++i)
        cout << vn[i] << ' ';
    cout << endl;

    // 基于迭代器的for循环，显式对迭代器做自增，需要理解迭代器运算

    for (int* it = an; it != an + size; ++it)
        cout << *it << ' ';
    cout << endl;

    for (vector<int>::const_iterator it = vn.begin(); it != vn.end(); ++it)
        cout << *it << ' ';
    cout << endl;

    // 基于泛型算法的for循环，需指明容器两端，对全遍历而言过于繁冗

    for_each(an, an + size, print);
    cout << endl;

    for_each(vn.begin(), vn.end(), print);
    cout << endl;

    // 基于范围的for循环，适用于包括数组在内的任何容器，既无需
    // 使用迭代器，亦无需指明容器两端，通用，透明，且代码简洁

    for (int n : an)
        cout << n << ' ';
    cout << endl;

    for (int n : vn)
        cout << n << ' ';
    cout << endl;

    // 撷取变量的类型，只要能从容器元素的类型做隐式转换即可，未必严格一致

    for (char c : an)
        cout << c << ' ';
    cout << endl;

    for (char c : vn)
        cout << c << ' ';
    cout << endl;

    // 也可以使用auto关键字，让编译器根据容器元素的类型，自动推导出撷取变量的类型

    for (auto n : an)
        cout << n << ' ';
    cout << endl;

    for (auto n : vn)
```

```
        cout << n << ' ';
    cout << endl;

    // 对于包含修改操作的循环，可以使用引用型撷取变量，引用容器中的每个元素

    for (auto& n : an)
        ++n;

    for (auto& n : vn)
        ++n;

    // 即便是非修改循环，使用引用型撷取变量也有助于提高运行性能，特别是当容
    // 器元素是复杂的类类型对象时。使用常引用还能防止对容器中元素的意外修改

    for (auto const& n : an)
        cout << n << ' ';
    cout << endl;

    for (auto const& n : vn)
        cout << n << ' ';
    cout << endl;

    return 0;
}
```

10.1.4.2 范围循环的注意事项

```
// for2.cpp

// 范围循环的注意事项

#include <iostream>
#include <list>
#include <map>
#include <set>

using namespace std;

list<int> scores(void) {
    cout << __FUNCTION__ << endl;

    return {70, 75, 80, 85, 90, 95};
}

int main(void) {
    // 对map和multimap容器使用范围循环，每次拿到的
    // 元素既不是键也不是值，而是由键和值组成的pair

    multimap<string, int> sn{
        {"张飞", 70}, {"赵云", 75}, {"关羽", 80},
        {"张飞", 85}, {"赵云", 90}, {"关羽", 95}};

    for (auto kv : sn)
        cout << kv.first << ':' << kv.second << endl;

```

```
    // 对范围循环中的撷取变量使用auto关键字，其自动推导出的类型
    // 是容器中的value_type，而非iterator，注意“.”和“->”的区别

    for (auto kv = sn.begin(); kv != sn.end(); ++kv)
        cout << kv->first << ':' << kv->second << endl;

    // 在使用基于范围的for循环时，还需要注意容器本身的约束
    /*
    for (auto& kv : sn)
    if (kv.first == "张飞")
        kv.first = "黄忠"; // 编译错误，map/multimap中的键是只读的

    set<int> numbers{10, 20, 30, 40, 50, 60};

    for (auto& number : numbers)
        ++number; // 编译错误，set/multiset中的元素是只读的
    */
    // 基于范围的for循环，无论循环体执行多少次，冒号后面的表达式永远只执行一次

    for (auto score : scores())
        cout << score << ' ';
    cout << endl;

    // 基于范围的for循环，其底层实现依然要借助于容器的迭代器，因
    // 此任何可能导致迭代器失效的结构性改变，都将引发未定义的后果

    auto ln = scores();
    for (auto n : ln) {
        cout << n << ' ';
        // ln.pop_front(); // 因结构性改变而致迭代器失效
    }
    cout << endl;

    return 0;
}
```

10.1.4.3 使自定义类型支持基于范围的for循环

```
// for3.cpp

// 使自定义类型支持基于范围的for循环

#include <iostream>

using namespace std;

// 一个类只要提供了分别用于获取起始和终止迭代器的
// begin和end方法，就可以支持基于范围的for循环

template<typename T, size_t S>
class Array {
public:
    T& operator[](size_t i) {
        return arr[i];
    }
```

```
18
19      T const& operator[](size_t i) const {
20          return const_cast<Array&>(*this)[i];
21      }
22
23      // 获取起始迭代器
24
25      T* begin(void) {
26          return arr;
27      }
28
29      T const* begin(void) const {
30          return const_cast<Array&>(*this).begin();
31      }
32
33      // 获取终止迭代器
34
35      T* end(void) {
36          return arr + S;
37      }
38
39      T const* end(void) const {
40          return const_cast<Array&>(*this).end();
41      }
42
43  private:
44      T arr[S];
45  };
46
47  int main(void) {
48      int n = 0;
49      Array<decltype(n), 5> a1;
50
51      for (auto& elem : a1) // elem:int&
52          elem = ++n * 10;
53
54      auto const& a2 = a1;
55
56      for (auto& elem : a2) // elem:int const&
57          cout << elem << ' ';
58      cout << endl;
59
60      return 0;
61  }
```

10.1.5 函数绑定

10.1.5.1 可调用对象与函数对象

```
1  // function1.cpp
2
3  // 可调用对象与函数对象
4
5  #include <iostream>
6  #include <functional>
```

```cpp

using namespace std;

// 函数

int fun(int x) {
    cout << __FUNCTION__ << '(' << x << ")->" << flush;

    return x;
}

// 实现了函数操作符的类

class Foo {
public:
    int operator()(int x, int y) const {
        cout << __FUNCTION__ << '(' << x << ',' << y << ")->" << flush;

        return x + y;
    }
};

// 可被转换为函数指针的类

class Bar {
    using hum_t = int (*)(int,int,int);

public:
    operator hum_t(void) const {
        return hum;
    }

private:
    static int hum(int x, int y, int z) {
        cout << __FUNCTION__ << '(' << x << ',' << y << ',' << z << ")->"
<< flush;

        return x + y + z;
    }
};

// 用函数对象实现回调

int run(function<int (int)> const& callback, int x) {
    return callback(x);
}

int run(function<int (int,int)> const& callback, int x, int y) {
    return callback(x, y);
}

int run(function<int (int,int,int)> const& callback, int x, int y, int z) {
    return callback(x, y, z);
}

int main(void) {
```

```
    // 能够被当做函数调用的不一定就是函数，它们也可能是：

    // 1.存放函数入口地址的函数指针

    int (*pfun)(int) = fun;
    cout << pfun(10) << endl;

    // 2.实现了函数操作符的类对象，亦称仿函数

    Foo foo;
    cout << foo(20, 30) << endl;

    // 3.可被转换为函数指针的类对象

    Bar bar;
    cout << bar(40, 50, 60) << endl;

    // 象pfun、foo、bar这样的对象被称为可调用对象，而它们
    // 的类型：int(*)(int)、Foo、Bar，则被称为可调用类型

    // function是可调用对象的包装器。它是一个类模板，可容纳
    // 以上三种中的任何一种可调用对象，并按照函数语法调用它们

    function<int (int)> ffun = fun;
    cout << ffun(10) << endl;

    function<int (int,int)> ffoo = foo;
    cout << ffoo(20, 30) << endl;

    function<int (int,int,int)> fbar = bar;
    cout << fbar(40, 50, 60) << endl;

    // 用函数对象实现回调

    cout << run(fun, 10) << endl;
    cout << run(foo, 20, 30) << endl;
    cout << run(bar, 40, 50, 60) << endl;

    return 0;
}
```

10.1.5.2 函数绑定器

```
// bind1.cpp

// 函数绑定器

#include <iostream>
#include <functional>

using namespace std;

// 函数

int fun(int x) {
```

```
    cout << __FUNCTION__ << '(' << x << ")->" << flush;

    return x;
}

// 实现了函数操作符的类

class Foo {
public:
    int operator()(int x, int y) const {
        cout << __FUNCTION__ << '(' << x << ',' << y << ")->" << flush;

        return x + y;
    }
};

// 可被转换为函数指针的类

class Bar {
    using hum_t = int (*)(int,int,int);

public:
    operator hum_t(void) const {
        return hum;
    }

private:
    static int hum(int x, int y, int z) {
        cout << __FUNCTION__ << '(' << x << ',' << y << ',' << z << ")->" <<
flush;

        return x + y + z;
    }
};

int main (void) {
    // 绑定器可将任何可调用对象和需要传递给它们的参数绑定为一个函数对象。该函
    // 数对象负责调用被绑定的可调用对象，依次传入被绑定的参数，并返回其返回值

    function<int (void)> f1 = bind(fun, 10);
    cout << f1() << endl;

    // placeholders::_n是一个占位符，其值将由传给函数对象的第n个参数取代

    Foo foo;
    auto f2 = bind(foo, 20, placeholders::_1);
    cout << f2(30) << endl;

    // 第1和第4个参数，即80和20，在调用时被吞掉。第2和第3个
    // 参数，即60和40，则作为第3和第1个参数被传给可调用对象

    cout << bind(Bar(), placeholders::_3, 50, placeholders::_2)(80, 60, 40,
20) << endl;

    return 0;
}
```

10.1.5.3 绑定类的成员函数

```cpp
// bind2.cpp

// 绑定类的成员函数

#include <iostream>
#include <functional>

using namespace std;

class A {
public:
    // 普通成员函数

    int foo(/* A const* this, */int arg) const {
        return /* this-> */var + arg;
    }

    // 静态成员函数

    static int bar(int arg) {
        return sta + arg;
    }

private:
    int var = 10;
    static int const sta = 20;
};

int main(void) {
    A a;

    // 绑定普通成员函数，注意作为this指针实参的调用对象地址

    function<int (int)> ffoo = bind(&A::foo, &a, placeholders::_1);
    cout << ffoo(1) << endl;

    // 绑定静态成员函数，除多了作用域限定与全局函数没有区别

    function<int (int)> fbar = bind(A::bar, placeholders::_1);
    cout << fbar(2) << endl;

    return 0;
}
```

10.1.5.4 新版本的bind简化和强化了老版本的bind1st和bind2nd

```cpp
// bind3.cpp

// 新版本的bind简化和强化了老版本的bind1st和bind2nd

#include <iostream>
```

```cpp
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

int main(void) {
    vector<int> scores{70, 40, 80, 50, 70, 80, 60, 60, 90, 50};
    auto begin = scores.begin(), end = scores.end();

    // 老版本的bind1st和bind2nd只能对带两个参数的函数绑定其第
    // 一或第二个参数，其目的就是将一个二元函数降级为一元函数

    // 60 <= score
    cout << "及格人数: " << count_if(begin, end, bind1st(less_equal<int>(), 60)) << endl;

    // score < 60
    cout << "不及格人数: " << count_if(begin, end, bind2nd(less<int>(), 60)) << endl;

    // 新版本的bind对被绑定函数的参数个数和具体绑定哪些参数都不做任何限制，更加灵活、通用且形式统一

    using placeholders::_1;

    // 60 <= score
    cout << "及格人数: " << count_if(begin, end, bind(less_equal<int>(), 60, _1)) << endl;

    // score < 60
    cout << "不及格人数: " << count_if(begin, end, bind(less<int>(), _1, 60)) << endl;

    // 通过bind还可以组合多个函数构成复合闭包

    // 70 <= score < 90
    cout << "良好人数: " << count_if(begin, end, bind(logical_and<bool>(),
        bind(less_equal<int>(), 70, _1), bind(less<int>(), _1, 90))) << endl;

    return 0;
}
```

10.1.6 匿名函数

10.1.6.1 lambda表达式

lambda表达式定义了一个匿名函数，其本质是一个匿名的仿函数对象，亦可被视为第4种形式的可调用对象。

```cpp
// lambda1.cpp

// lambda表达式

```

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main(void) {
    // [捕获表](参数表) 选项 -> 返回类型 { 函数体 }

    auto f1 = [](int x) -> int { return x * x; };

    // lambda表达式的值是一个实现了函数操作符的类对象，即仿函数。该类的成员变量来自
    // 于捕获表，其函数操作符函数的参数、返回类型和函数体均来自于lambda表达式的定义

    cout << f1(10) << endl; // 100

    // 返回类型通常可以省略，编译器可以根据return语句自动推导出返回类型

    auto f2 = [](int x) { return x * x; };

    cout << f2(11) << endl; // 121

    // 列表初始化不能用于返回类型的自动推导

    // auto f3 = [](void) { return {10, 20, 30, 40, 50}; };
    auto f3 = [](void) -> vector<int> { return {10, 20, 30, 40, 50}; };

    for (auto n : f3())
        cout << n << ' ';
    cout << endl; // 10 20 30 40 50

    // 空参数表和返回类型可以省略

    [](void) -> void { cout << "hello world" << endl; }(); // hello world
    []() { cout << "hello world" << endl; }(); // hello world
    [] { cout << "hello world" << endl; }(); // hello world

    return 0;
}
```

10.1.6.2 lambda表达式的捕获表

```cpp
// lambda2.cpp

// lambda表达式的捕获表

#include <cmath>
#include <iostream>

using namespace std;

int a = 10;

class Base {
protected:
    int b = 20;
```

```
};

class Derived : public Base {
public:
    void foo(int d) {
        // 不捕获任何外部变量

        [](int e) {
            ++a;
            cout << "a=" << a << endl;
            // ++b;
            // cout << "b=" << b << endl;
            // ++c;
            // cout << "c=" << c << endl;
            // ++d;
            // cout << "d=" << d << endl;
            ++e;
            cout << "e=" << e << endl << endl; }(50);

        // 按值捕获指定的外部变量

        [d](int e) {
            ++a;
            cout << "a=" << a << endl;
            // ++b;
            // cout << "b=" << b << endl;
            // ++c;
            // cout << "c=" << c << endl;
            // ++d; // 不可修改按值捕获的外部变量
            cout << "d=" << d << endl;
            ++e;
            cout << "e=" << e << endl << endl; }(50);

        // 按引用捕获指定的外部变量

        [&d](int e) {
            ++a;
            cout << "a=" << a << endl;
            // ++b;
            // cout << "b=" << b << endl;
            // ++c;
            // cout << "c=" << c << endl;
            ++d; // 可修改按引用捕获的外部变量
            cout << "d=" << d << endl;
            ++e;
            cout << "e=" << e << endl << endl; }(50);

        // 捕获this指针

        [this](int e) {
            ++a;
            cout << "a=" << a << endl;
            ++b;
            cout << "b=" << b << endl;
            ++c;
            cout << "c=" << c << endl;
```

```
            // ++d
            // cout << "d=" << d << endl;
            ++e;
            cout << "e=" << e << endl << endl; }(50);

        // 按值捕获所有外部变量（包括this指针）

        [=](int e) {
            ++a;
            cout << "a=" << a << endl;
            ++b;
            cout << "b=" << b << endl;
            ++c;
            cout << "c=" << c << endl;
            // ++d; // 不可修改按值捕获的外部变量
            cout << "d=" << d << endl;
            ++e;
            cout << "e=" << e << endl << endl; }(50);

        // 按引用捕获所有外部变量（包括this指针）

        [&](int e) {
            ++a;
            cout << "a=" << a << endl;
            ++b;
            cout << "b=" << b << endl;
            ++c;
            cout << "c=" << c << endl;
            ++d; // 可修改按引用捕获的外部变量
            cout << "d=" << d << endl;
            ++e;
            cout << "e=" << e << endl << endl; }(50);

        // 按值捕获所有外部变量（包括this指针），按引用捕获指定的外部变量

        [=, &d](int e) {
            ++a;
            cout << "a=" << a << endl;
            ++b;
            cout << "b=" << b << endl;
            ++c;
            cout << "c=" << c << endl;
            ++d; // 可修改按引用捕获的外部变量
            cout << "d=" << d << endl;
            ++e;
            cout << "e=" << e << endl << endl; }(50);
    }

private:
    int c = 30;
};

int main (void) {
    Derived d;
    d.foo(40);

```

```
    // 按值捕获，保存在f1中的是x自增之前的值60

    int x = 60;
    auto f1 = [x] { cout << "x=" << x << endl << endl; };
    ++x;
    f1();

    // 按引用捕获，保存在f2中的是y的引用，自增后变成71

    int y = 70;
    auto f2 = [&y]{ cout << "y=" << y << endl << endl; };
    ++y;
    f2();

    // 非可变lambda表达式不能修改按值捕获的外部变量

    int z = 80;
    [=] { cout << "z=" << /*++*/z << endl << endl; }();

    // 带有mutable关键字的可变lambda表达式可以修改按值捕获的外部变量，但实际被修改的
    // 只是该变量在lambda表达式中的拷贝。可变lambda表达式不可省略参数表，即使其为空

    [=]() mutable { cout << "z=" << ++z << endl; }();
    cout << "z=" << z << endl << endl;

    // 这与按引用捕获外部变量并在lambda表达式中修改的情况不一样

    [&] { cout << "z=" << ++z << endl; }();
    cout << "z=" << z << endl << endl;

    // 不捕获任何外部变量的lambda表达式可被隐式转换为相应类型的函数指针

    double (*pyth)(double, double) =
        [](double a, double b) -> double { return sqrt(a * a + b * b); };
    cout << "c=" << pyth(3, 4) << endl;

    return 0;
}
```

10.1.6.3 借助lambda表达式简化对标准库算法的调用

```
// lambda3.cpp

// 利用lambda表达式简化标准库算法的调用

#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <functional>

using namespace std;

class Student {
public:
```

```
    Student(string const& name, int age, int score) : name(name), age(age), score(score) {}

    friend ostream& operator<<(ostream& os, Student const& student) {
        return os << '(' << student.name << ',' << student.age << ',' << student.score << ')';
    }

    string name;  // 姓名
    int    age;   // 年龄
    int    score; // 成绩
};

int main (void) {
    vector<Student> vs{
        {"张飞", 40, 80},
        {"赵云", 30, 70},
        {"关羽", 30, 80},
        {"黄忠", 30, 90},
        {"马超", 20, 80}};

    copy(vs.begin(), vs.end(), ostream_iterator<Student>(cout, ""));
    cout << endl;

    // 用仿函数做排序比较器
    /*
    class StudentComparator {
    public:
        // 按成绩降序排列，成绩相同者按年龄升序排列
        bool operator()(Student const& a, Student const& b) const {
            if (a.score == b.score)
                return a.age < b.age;
            else
                return a.score > b.score;
        }
    };

    sort(vs.begin(), vs.end(), StudentComparator());
    */
    // 用lambda表达式做排序比较器

    sort(vs.begin(), vs.end(),
        [](Student const& a, Student const& b) {
            if (a.score == b.score)
                return a.age < b.age;
            else
                return a.score > b.score;
        });

    copy(vs.begin(), vs.end(), ostream_iterator<Student>(cout, ""));
    cout << endl;

    vector<int> scores{70, 40, 80, 50, 70, 80, 60, 60, 90, 50};
    auto begin = scores.begin(), end = scores.end();

    // 通过bind组合多个函数构成复合闭包
```

```
    /*
    using placeholders::_1;

    // 70 <= score < 90
    cout << "良好人数: " << count_if(begin, end, bind(logical_and<bool>(),
        bind(less_equal<int>(), 70, _1), bind(less<int>(), _1, 90))) << endl;
    */
    // 使用lambda表达式构造闭包更加简洁

    // 70 <= score < 90
    cout << "良好人数: " << count_if(begin, end,
        [](int score) { return 70 <= score && score < 90; }) << endl;

    return 0;
}
```

10.1.7 泛型元组

tuple可以理解为是对老版本pair的增强和扩展，其中的元素个数不再限于两个，功能也更加丰富。

```
// tuple1.cpp

// 泛型元组

#include <iostream>
#include <tuple>

using namespace std;

int main (void) {
    char const* name   = "张飞";
    int         age    = 25;
    double      height = 1.75;

    // 直接构造tuple对象

    tuple<char const*, int, double> t1(name, age, height);

    // 获取tuple元素的值

    cout << '(' << get<0>(t1) << ',' << get<1>(t1) << ',' << get<2>(t1) <<
')' << endl;

    // 通过make_tuple构造tuple对象

    auto t2 = make_tuple(name, age, height);
    cout << '(' << get<0>(t2) << ',' << get<1>(t2) << ',' << get<2>(t2) <<
')' << endl;

    // 按以上两种方式构造的tuple对象所保存的只是构造实参的拷贝

    get<0>(t1) = "赵云";
    get<1>(t2) = 20;
    cout << '(' << name << ',' << age << ',' << height << ')' << endl;

```

```
    // 通过tie构造tuple对象，保存构造实参的左值引用

    auto t3 = tie(name, age, height);
    get<2>(t3) = 1.65;
    cout << '(' << name << ',' << age << ',' << height << ')' << endl;

    // 事实上tie的本意是用于解析而非构造tuple对象

    auto t4 = make_tuple("关羽", 30, 1.85);
    tie(name, age, height) = t4;
    cout << '(' << name << ',' << age << ',' << height << ')' << endl;

    // 可以用ignore占位符忽略掉不感兴趣的元素

    auto t5 = make_tuple("马超", 22, 1.65);
    tie(name, ignore, height) = t5;
    cout << '(' << name << ',' << age << ',' << height << ')' << endl;

    // 通过forward_as_tuple构造tuple对象，保存构造实参的右值引用

    auto t6 = forward_as_tuple("黄忠", 40, 1.95);
    cout << '(' << get<0>(t6) << ',' << get<1>(t6) << ',' << get<2>(t6) <<
')' << endl;

    // 用tuple_cat连接多个tuple对象

    int x = 10, y = 20;
    auto t7 = tuple_cat(make_tuple(x, y), tie(x, y), forward_as_tuple(x,
y));
    ++x; ++y;
    cout << "((" << get<0>(t7) << ',' << get<1>(t7) << "),(" <<
        get<2>(t7)++ << ',' << get<3>(t7)++ << "),(" <<
        get<4>(t7) << ',' << get<5>(t7) << "))" << endl;

    return 0;
}
```

10.2 提高性能

10.2.1 右值引用

10.2.1.1 左值和右值

```
// rvr1.cpp

// 左值和右值

#include <iostream>

using namespace std;

int& foo(int& a) {
    return a;
}

```

```cpp
int bar(int& a) {
    return a;
}

string hum(void) {
    return "hello world";
}

int main(void) {
    // 区分左右值，关键看是否可对该值取地址

    int a = 1, b = 2;

    // 可以取地址的值就是左值，左值通常具名

    cout << &a << endl;        // 有名变量
    cout << &foo(a) << endl;   // 引用形式的函数返回值
    cout << &++a << endl;      // 前缀自增减表达式的值
    cout << &(a = b) << endl; // 赋值类表达式的值

    // 不可取地址的值就是右值，右值通常匿名

    // cout << &10 << endl;         // 字面值常量
    // cout << &bar(a) << endl;     // 值形式的函数返回值
    // cout << &a++ << endl;        // 后缀自增减表达式的值
    // cout << &(a + b) << endl;    // 运算表达式的值
    // cout << &(double)a << endl; // 类型转换表达式的值

    // 以上右值也被称为纯右值，即临时值，天然具有只读属性，但
    // 类对象形式的临时值，虽为右值，在某些编译器上亦可取地址

    // cout << &hum() << endl;
    // cout << &(string)"Hello, World !" << endl;
    // cout << &[=] { return a + b; } << endl;

    return 0;
}
```

10.2.1.2 左值引用和右值引用

```cpp
// rvr2.cpp

// 左值引用和右值引用

#include <iostream>

using namespace std;

class X {
public:
    X(void) {
        cout << "构造: " << this << endl;
    }

    X(X const& x) {
```

```
        cout << "拷贝：" << &x << "->" << this << endl;
    }

    ~X(void) {
        cout << "析构：" << this << endl;
    }
};

X foo(void) {
    X x; // 局部值

    return x;
}

int main(void) {
    int a = 1, b = 2;

    // 左值引用只能引用左值，不能引用右值

    int& lvr1 = a;
    // int& lvr2 = a + b;

    // 右值引用只能引用右值，不能引用左值

    int&& rvr1 = a + b;
    // int&& rvr2 = a;

    // 常左值引用既能引用左值，也能引用右值，故称万能引用

    int const& clvr1 = a;
    int const& clvr2 = a + b;

    // 临时值仅具语句级生命期

    foo(); // 构造（局部值）->拷贝（临时值）->析构（局部值）->析构（临时值）
    cout << "--------------------" << endl;

    // 引用可将临时值的寿命延长至与引用变量本身一样

    X&& rvr3 = foo(); // 构造（局部值）->拷贝（临时值）->析构（局部值）
    cout << "--------------------" << endl;

    return 0;
}   // 析构（临时值）

// g++ -fno-elide-constructors rvr2.cpp
```

10.2.1.3 通用引用

```
// rvr3.cpp

// 通用引用

#include <iostream>

```

```cpp
using namespace std;

// 在函数模板隐式推断过程中，若实参为左值，则T&&被推断为左值引用，
// 若实参为右值，则T&&被推断为右值引用，这样的引用被称为通用引用

template<typename T>
void foo(T&& r) {
    cout << r << endl;
}

// 只有T&&才是通用引用，加了const就不是了

template<typename T>
void bar(T const&& r) {
    cout << r << endl;
}

int main(void) {
    int a = 1, b = 2;

    foo(a); // r:int&，左值引用可以引用左值
    foo(a + b); // r:int&&，右值引用可以引用右值

    // 不做隐式推断，没有通用引用

    // foo<int>(a); // r:int&&，右值引用无法引用左值
    foo<int>(a + b); // r:int&&，右值引用可以引用右值

    // T const&&只能被推断为常右值引用

    // bar(a); // r:int const&&，右值引用无法引用左值
    bar(a + b); // r:int const&&，右值引用可以引用右值

    // 类似的规则也同样适用于基于auto关键字的类型推导

    auto&& c = a; // c:int&，左值引用可以引用左值
    auto&& d = a + b; // d:int&&，右值引用可以引用右值

    // auto const&& e = a; // e:int const&&，右值引用无法引用左值
    auto const&& f = a + b; // f:int const&&，右值引用可以引用右值

    return 0;
}
```

10.2.1.4 引用折叠

C++98/03标准不允许声明引用的引用，但C++11允许这么声明，并根据引用折叠规则将其处理为简单引用的形式：只有右值引用的右值引用还是右值引用，其它情况下的叠加都是左值引用。

```cpp
// rvr4.cpp

// 引用折叠

#include <iostream>

```

```cpp
using namespace std;

template<typename T>
void what(void) {
    if (is_lvalue_reference<T>::value)
        cout << "左值引用" << endl;
    else
    if (is_rvalue_reference<T>::value)
        cout << "右值引用" << endl;
    else
        cout << "不是引用" << endl;
}

int main(void) {
    using lvr_t = int&;
    using rvr_t = int&&;

    what<lvr_t>(); // 左值引用
    what<rvr_t>(); // 右值引用

    what<lvr_t&>(); // 左值引用的左值引用是左值引用
    what<rvr_t&>(); // 右值引用的左值引用是左值引用

    what<lvr_t&&>(); // 左值引用的右值引用是左值引用
    what<rvr_t&&>(); // 右值引用的右值引用是右值引用

    return 0;
}
```

10.2.1.5 move和forward

```cpp
// rvr5.cpp

// move和forward

#include <iostream>

using namespace std;

void foo(int& lvr) {
    cout << "foo(int&)" << endl;
}

void foo(int&& rvr) {
    cout << "foo(int&&)" << endl;
}

template<typename T>
void bar(T&& r) {
    foo(r); // 无论r是左值引用还是右值引用，其实都是左值
}

template<typename T>
void hum(T&& r) {
    foo(move(r)); // 无论r是左值引用还是右值引用，一律被move处理为右值
```

```
}

template<typename T>
void fun(T&& r) {
    foo(forward<T>(r)); // r是左值或右值引用，被forward处理为左值或右值
}

int main(void) {
    int a = 1, b = 2;

    // 右值引用只能引用右值，但其本身是左值，可取地址，可被
    // 修改，可被左值引用引用

    int&& c = a + b;
    cout << &c << endl;
    cout << ++c << endl;
    int& d = c;

    // 左值引用只能引用左值，其本身也是左值，可取地址，可被
    // 修改，不能被右值引用引用，除非通过move将其转换为右值

    int& e = a;
    cout << &e << endl;
    cout << ++e << endl;
    // int&& f = e;
    int&& g = move(e);
    int&& h = move(a);

    foo(a);     // ->foo(int&)
    foo(a + b); // ->foo(int&&)

    bar(a);     // ->foo(int&)
    bar(a + b); // ->foo(int&)

    hum(a);     // ->foo(int&&)
    hum(a + b); // ->foo(int&&)

    fun(a);     // ->foo(int&)
    fun(a + b); // ->foo(int&&)

    return 0;
}
```

10.2.2 移动语义

10.2.2.1 支持浅拷贝的构造函数

```
// move1.cpp

// 支持浅拷贝的构造函数

#include <string.h>

#include <iostream>

```

```
using namespace std;

class String {
public:
    String(char const* s)
        : s(strcpy(new char[strlen(s ? s : "")+1], s ? s : "")) {
        cout << "构造函数: " << this << endl;
    }

    // 支持浅拷贝的构造函数

    String(String const& str) : s(str.s) {
        cout << "浅拷构造: " << &str << "->" << this << endl;
    }

    ~String(void) {
        cout << "析构函数: " << this << endl;

        delete[] s;
    }

    friend ostream& operator<< (ostream& os, String const& str) {
        return os << str.s;
    }

private:
    char* s;
};

String foo(void) {
    String str = "hello world";

    cout << "foo: " << str << endl;

    return str;
}

int main(void) {
    String str = foo();

    cout << "main: " << str << endl; // 运行错误

    return 0;
} // 崩溃

// g++ -fno-elide-constructors move1.cpp
```

10.2.2.2 支持深拷贝的构造函数

```
// move2.cpp

// 支持深拷贝的构造函数

#include <string.h>

```

```cpp
#include <iostream>

using namespace std;

class String {
public:
    String(char const* s)
        : s(strcpy(new char[strlen(s ? s : "")+1], s ? s : "")) {
        cout << "构造函数: " << this << endl;
    }

    // 支持深拷贝的构造函数

    String(String const& str)
        : s(strcpy(new char[strlen(str.s)+1], str.s)) {
        cout << "深拷构造: " << &str << "->" << this << endl;
    }

    ~String(void) {
        cout << "析构函数: " << this << endl;

        delete[] s;
    }

    friend ostream& operator<< (ostream& os, String const& str) {
        return os << str.s;
    }

private:
    char* s;
};

String foo(void) {
    String str = "hello world";

    cout << "foo: " << str << endl;

    return str;
}

int main(void) {
    String str = foo();

    cout << "main: " << str << endl;

    return 0;
}

// g++ -fno-elide-constructors move2.cpp
```

10.2.2.3 支持移动语义的构造函数

```cpp
// move3.cpp

// 支持移动语义的构造函数

#include <string.h>

#include <iostream>

using namespace std;

class String {
public:
    String(char const* s)
        : s(strcpy(new char[strlen(s ? s : "")+1], s ? s : "")) {
        cout << "构造函数: " << this << endl;
    }

    // 支持移动语义的构造函数

    String(String&& str) : s(str.s) {
        cout << "移动构造: " << &str << "->" << this << endl;

        str.s = nullptr;
    }

    ~String(void) {
        cout << "析构函数: " << this << endl;

        delete[] s;
    }

    friend ostream& operator<< (ostream& os, String const& str) {
        return os << str.s;
    }

private:
    char* s;
};

String foo(void) {
    String str = "hello world";

    cout << "foo: " << str << endl;

    return str;
}

int main(void) {
    String str = foo(); // 等号右边的对象是一个将亡右值，其资源即将被移动到左侧对象中

    cout << "main: " << str << endl;

    return 0;
```

```
}

// g++ -fno-elide-constructors move3.cpp
```

10.2.2.4 同时提供支持深拷贝和移动语义的构造函数和赋值操作符函数

同时提供支持深拷贝和移动语义的构造函数和赋值操作符函数，编译器会在适当的时候选择适当的版本，在保证功能的同时提高性能。

```cpp
// move4.cpp

// 同时提供支持深拷贝和移动语义的构造函数和赋值操作符函数

#include <string.h>

#include <iostream>

using namespace std;

class String {
public:
    String(char const* s)
        : s(strcpy(new char[strlen(s ? s : "")+1], s ? s : "")) {
        cout << "构造函数：" << this << endl;
    }

    // 支持深拷贝的构造函数

    String(String const& str)
        : s(strcpy(new char[strlen(str.s)+1], str.s)) {
        cout << "深拷构造：" << &str << "->" << this << endl;
    }

    // 支持深拷贝的赋值操作符函数

    String& operator=(String const& str) {
        cout << "深拷赋值：" << &str << "->" << this << endl;

        if (&str != this) {
            String tmp = str; // 深拷构造
            swap(s, tmp.s);
        }

        return *this;
    }

    // 支持移动语义的构造函数

    String(String&& str) : s(str.s) {
        cout << "移动构造：" << &str << "->" << this << endl;

        str.s = nullptr;
    }

    // 支持移动语义的赋值操作符函数
```

```
47
48      String& operator=(String&& str) {
49          cout << "移动赋值: " << &str << "->" << this << endl;
50
51          if (&str != this) {
52              String tmp = move(str); // 移动构造
53              swap(s, tmp.s);
54          }
55
56          return *this;
57      }
58
59      ~String(void) {
60          cout << "析构函数: " << this << endl;
61
62          delete[] s;
63      }
64
65      friend ostream& operator<< (ostream& os, String const& str) {
66          return os << str.s;
67      }
68
69  private:
70      char* s;
71  };
72
73  String foo(void) {
74      String str = "hello world";
75
76      cout << "foo: " << str << endl;
77
78      return str;
79  }
80
81  int main(void) {
82      String a = "hello tarena", b = "hello C++";
83
84      cout << "---------------------------------------- 1" << endl;
85
86      String c = a; // 深拷构造
87      cout << "main: " << c << endl;
88
89      cout << "---------------------------------------- 2" << endl;
90
91      c = b; // 深拷赋值
92      cout << "main: " << c << endl;
93
94      cout << "---------------------------------------- 3" << endl;
95
96      String d = foo(); // 移动构造
97      cout << "main: " << d << endl;
98
99      cout << "---------------------------------------- 4" << endl;
100
101     c = foo();        // 移动赋值
102     cout << "main: " << c << endl;
```

```
103
104     cout << "---------------------------------------- 5" << endl;
105
106     return 0;
107 }
108
109 // g++ -fno-elide-constructors move4.cpp
```

10.2.3 就地构造

几乎所有标准库容器都提供了类似emplace_back、emplace_front和emplace这样的接口，以作为传统的push_back、push_front和insert接口的替代方案。相较于后者，前者采用在容器内部就地构造对象的方式，以尽可能避免过多的内存复制和移动，具有更好的运行时性能。

```
1  // emplace1.cpp
2
3  // 就地构造
4
5  #include <iostream>
6  #include <vector>
7  #include <algorithm>
8  #include <iterator>
9
10 using namespace std;
11
12 class Point2D {
13 public:
14     Point2D(int x, int y) : x(x), y(y) {
15         cout << "构造函数: " << this << endl;
16     }
17
18     Point2D(Point2D const& p) : x(p.x), y(p.y) {
19         cout << "拷贝构造: " << &p << "->" << this << endl;
20     }
21
22     Point2D& operator=(Point2D const& p) {
23         cout << "拷贝赋值: " << &p << "->" << this << endl;
24
25         if (&p != this) {
26             x = p.x;
27             y = p.y;
28         }
29
30         return *this;
31     }
32
33     ~Point2D(void) {
34         cout << "析构函数: " << this << endl;
35     }
36
37     friend ostream& operator<<(ostream& os, Point2D const& p) {
38         return os << '(' << p.x << ',' << p.y << ')';
39     }
40
41 private:
```

```
    int x, y;
};

int main (void) {
    vector<Point2D> v1;

    v1.push_back({1, 2});

    cout << "---------------------------------- 1" << endl;

    v1.push_back({5, 6});

    cout << "---------------------------------- 2" << endl;

    v1.insert(v1.begin() + 1, {3, 4});

    cout << "---------------------------------- 3" << endl;

    copy(v1.begin(), v1.end(), ostream_iterator<Point2D>(cout, ""));
    cout << endl;

    cout << "---------------------------------- 4" << endl;

    vector<Point2D> v2;

    v2.emplace_back(1, 2);

    cout << "---------------------------------- 5" << endl;

    v2.emplace_back(5, 6);

    cout << "---------------------------------- 6" << endl;

    v2.emplace(v2.begin() + 1, 3, 4);

    cout << "---------------------------------- 7" << endl;

    copy(v2.begin(), v2.end(), ostream_iterator<Point2D>(cout, ""));
    cout << endl;

    cout << "---------------------------------- 8" << endl;

    return 0;
}
```

10.2.4 无序容器

C++11在C++98/03所提供的四个基于红黑树的有序关联容器，map、multimap、set和multiset的基础上，又增加了四个基于散列表的无序关联容器，unordered_map、unordered_multimap、unordered_set和unordered_multiset。与前者相比，后者通过哈希而非排序组织其中的元素，因而效率更高。

```
// unordered1.cpp

// 无序容器
```

```cpp

#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

// 姓名类

class Name {
public:
    Name(string const& firstName, string const& familyName)
        : firstName(firstName), familyName(familyName) {}

    string firstName, familyName;
};

// 哈希器

class NameHash {
public:
    size_t operator()(Name const& name) const {
        return hash<string>()(name.firstName) ^ (hash<string>()
(name.familyName) << 1);
    }
};

// 判等器

class NameEqual {
public:
    bool operator()(Name const& a, Name const& b) const {
        return a.firstName == b.firstName && a.familyName == b.familyName;
    }
};

int main (void) {
    // 缺省构造

    unordered_map<string, int> a;

    // 列表构造

    unordered_map<string, int> b {{"张飞", 70}, {"赵云", 80}, {"关羽", 90}};

    for (auto const& kv : b)
        cout << kv.first << ':' << kv.second << endl;

    // 拷贝构造

    unordered_map<string, int> c = b;

    for (auto const& kv : c)
        cout << kv.first << ':' << kv.second << endl;

    // 移动构造
```

```
59
60      unordered_map<string, int> d = move(c);
61
62      for (auto const& kv : d)
63          cout << kv.first << ':' << kv.second << endl;
64
65      // 范围构造
66
67      vector<pair<string, int>> e {{"张飞", 70}, {"赵云", 80}, {"关羽", 90}};
68      unordered_map<string, int> f(e.begin(), e.end());
69
70      for (auto const& kv : f)
71          cout << kv.first << ':' << kv.second << endl;
72
73      // 如果键的类型不是基本类型，那么必须在构造容
74      // 器的同时指定与键类型相适应的哈希器和判等器
75
76      unordered_map<Name, int, NameHash, NameEqual> g {
77          {{"飞", "张"}, 70}, {{"云", "赵"}, 80}, {{"羽", "关"}, 90}};
78
79      for (auto const& kv : g)
80          cout << kv.first.familyName << kv.first.firstName << ':' <<
    kv.second << endl;
81
82      return 0;
83  }
```

10.3 避免重复

10.3.1 类型萃取

10.3.1.1 类的编译期常量

10.3.1.1.1 C++98/03标准为类定义编译期常量的方法

```
1   // constant1.cpp
2
3   // C++98/03标准为类定义编译期常量的方法
4
5   #include <iostream>
6
7   using namespace std;
8
9   class One {
10  public:
11      static int const value = 1;
12  };
13
14  class Two {
15  public:
16      enum { value = 2 };
17  };
18
19  int main(void) {
20      cout << One::value << endl; // 1
```

```
    cout << Two::value << endl; // 2

    return 0;
}
```

10.3.1.1.2 C++11标准为类定义编译期常量的方法

在C++11中，为一个类定义编译期常量，既无需自己定义常静态成员变量，亦无需提供枚举成员类型，只要从integral_constant基类继承即可。

```
// constant2.cpp

// C++11标准为类定义编译期常量的方法

#include <iostream>

using namespace std;

class One : public integral_constant<int, 1> {};

class Two : public integral_constant<int, 2> {};

int main(void) {
    cout << One::value << endl; // 1
    cout << Two::value << endl; // 2

    return 0;
}
```

10.3.1.1.3 integral_constant类模板的简化实现

```
// constant3.cpp

// integral_constant类模板的简化实现

#include <iostream>

using namespace std;

template<typename T, T v>
class IntegralConstant {
public:
    typedef T value_type;
    typedef IntegralConstant<value_type, v> type;

    static value_type const value = v;

    operator value_type(void) const {
        return value;
    }
};

class One : public IntegralConstant<int, 1> {};

class Two : public IntegralConstant<int, 2> {};
```

```
25
26  int main (void) {
27      cout << One::value << endl; // 1
28      cout << Two::value << endl; // 2
29
30      One one;
31      One::value_type a = one;
32      cout << a << endl; // 1
33
34      Two two;
35      Two::value_type b = two;
36      cout << b << endl; // 2
37
38      return 0;
39  }
```

10.3.1.1.4 编译期布尔类型

```
1   // constant4.cpp
2
3   // 编译期布尔类型
4
5   #include <iostream>
6
7   using namespace std;
8   /*
9   typedef integral_constant<bool, false> false_type;
10  typedef integral_constant<bool, true> true_type;
11  */
12  template<typename T> class IsPointer : public false_type {};
13  template<typename T> class IsPointer<T*> : public true_type {};
14
15  template<typename T> class IsReference : public false_type {};
16  template<typename T> class IsReference<T&> : public true_type {};
17
18  template<typename T> class IsArray : public false_type {};
19  template<typename T> class IsArray<T[]> : public true_type {};
20
21  int main(void) {
22      cout << boolalpha << false_type::value << endl; // false
23      cout << true_type::value << endl;               // true
24
25      cout << IsPointer<int>::value << endl;  // false
26      cout << IsPointer<int*>::value << endl; // true
27
28      cout << IsReference<int>::value << endl;  // false
29      cout << IsReference<int&>::value << endl; // true
30
31      cout << IsArray<int>::value << endl;   // false
32      cout << IsArray<int[]>::value << endl; // true
33
34      return 0;
35  }
```

10.3.1.2 编译期类型和类型关系判断

10.3.1.2.1 编译期类型判断

当且仅当实例化模板的类型实参为...	...模板实例化类中的布尔型静态成员value的值为true
空类型	is_void
整数类型	is_integral
浮点数类型	is_floating_point
指针类型（不包括成员指针）	is_pointer
成员指针类型	is_member_pointer
引用类型	is_reference
数组类型	is_array
枚举类型	is_enum
联合类型	is_union
类类型（包括结构体）	is_class
多态类型（含虚函数）	is_polymorphic
抽象类型（含纯虚函数）	is_abstract
函数类型	is_function
算术类型（整数、浮点数）	is_arithmetic
基本类型（空、整数、浮点数、指针）	is_fundamental
复合类型	is_compound
对象类型（除空、引用、函数外）	is_object
标量类型（空、指针、成员指针、枚举、算术）	is_scalar
有符号类型	is_signed
无符号类型	is_unsigned
常类型	is_const
.....	.....

```cpp
// is1.cpp

// 编译期类型判断

#include <iostream>

using namespace std;
```

```cpp

int main(void) {
    cout << boolalpha;

    cout << is_function<int (int,int)>::value << endl; // true

    cout << is_const<int>::value << endl;              // false
    cout << is_const<const int>::value << endl;        // true
    cout << is_const<int const>::value << endl;        // true

    cout << is_const<int*>::value << endl;             // false
    cout << is_const<const int*>::value << endl;       // false
    cout << is_const<int const*>::value << endl;       // false
    cout << is_const<int* const>::value << endl;       // true
    cout << is_const<int const* const>::value << endl; // true

    cout << is_const<int&>::value << endl;             // false
    cout << is_const<const int&>::value << endl;       // false
    cout << is_const<int const&>::value << endl;       // false

    // 禁止针对引用本身而非其目标使用常限定

    // cout << is_const<int& const>::value << endl;
    // cout << is_const<int const& const>::value << endl;

    return 0;
}
```

10.3.1.2.2 编译期类型关系判断

当且仅当实例化模板的两个类型实参满足...	...模板实例化类中的布尔型静态成员value的值为true
相同类型	is_same
第一个是第二个的基类	is_base_of
第一个能被隐式转换为第二个	is_convertible
.....	.....

```cpp
// is2.cpp

// 编译期类型关系判断

#include <iostream>

using namespace std;

class A {};                       //   A
class B : public A {};            //  /
                                  // B   C
class C {};                       //  \ /
class D : public B, public C {}; //   D

```

```cpp
class Integer {
public:
    // int -显式转换-> Integer

    explicit Integer(int n = 0) : n(n) {}

    // Integer -隐式转换-> int

    operator int(void) const {
        return n;
    }

private:
    int n;
};

int main (void) {
    cout << boolalpha;

    cout << is_same<signed int, int>::value << endl;      // true
    cout << is_same<int, unsigned int>::value << endl;    // false
    cout << is_same<unsigned long, size_t>::value << endl; // true

    cout << is_base_of<A, B>::value << endl; // true
    cout << is_base_of<A, C>::value << endl; // false
    cout << is_base_of<A, D>::value << endl; // true

    cout << is_convertible<double, int>::value << endl;   // true
    cout << is_convertible<double*, int*>::value << endl; // false
    cout << is_convertible<int*, void*>::value << endl;   // true

    cout << is_convertible<A, B>::value << endl;   // false
    cout << is_convertible<B, A>::value << endl;   // true
    cout << is_convertible<A*, B*>::value << endl; // false
    cout << is_convertible<B*, A*>::value << endl; // true
    cout << is_convertible<A&, B&>::value << endl; // false
    cout << is_convertible<B&, A&>::value << endl; // true

    cout << is_convertible<int, Integer>::value << endl; // false
    cout << is_convertible<Integer, int>::value << endl; // true

    return 0;
}
```

10.3.1.3 编译期类型转换及其应用

10.3.1.3.1 编译期类型转换

用特定类型实参实例化模板...	模板实例化类中的成员类型type为...
remove_const	去除类型实参中的常量限定
add_const	在类型实参上添加常量限定

用特定类型实参实例化模板...	模板实例化类中的成员类型type为...
remove_pointer	去除类型实参中的指针属性
add_pointer	在类型实参上添加指针属性
remove_reference	去除类型实参中的引用属性
add_lvalue_reference	在类型实参上添加左值引用
add_rvalue_reference	在类型实参上添加右值引用
remove_extent	去除数组型类型实参的顶层维度
remove_all_extents	去除数组型类型实参的所有维度
decay	衰减类型实参：去除CV限定、去除引用属性、数组变指针、函数变函数指针
common_type	所有类型实参都可被隐式转换到的公共类型
.....	.....

```cpp
// convert1.cpp

// 编译期类型转换

#include <iostream>

using namespace std;

int main(void) {
    cout << boolalpha;

    cout << is_same<remove_const<int const>::type, int>::value << endl; // true
    cout << is_same<add_const<int>::type, int const>::value << endl;    // true

    cout << is_same<remove_pointer<int*>::type, int>::value << endl; // true
    cout << is_same<add_pointer<int>::type, int*>::value << endl;    // true

    cout << is_same<remove_reference<int&>::type, int>::value << endl; // true
    cout << is_same<add_lvalue_reference<int>::type, int&>::value << endl; // true
    cout << is_same<add_rvalue_reference<int>::type, int&&>::value << endl; // true

    cout << is_same<remove_extent<int[3]>::type, int>::value << endl; // true
    cout << is_same<remove_extent<int[3][4]>::type, int[4]>::value << endl; // true
    cout << is_same<remove_all_extents<int[3][4]>::type, int>::value << endl; // true
```

```
25
26      cout << is_same<decay<int const volatile>::type, int>::value << endl;
          // true
27      cout << is_same<decay<int&>::type, int>::value << endl;
          // true
28      cout << is_same<decay<int[3]>::type, int*>::value << endl;
         // true
29      cout << is_same<decay<int[3][4]>::type, int (*)[4]>::value << endl;
          // true
30      cout << is_same<decay<int (int,int)>::type, int (*)(int,int)>::value <<
    endl; // true
31
32      cout << is_same<common_type<bool, char, short>::type, int>::value <<
    endl; // true
33
34      return 0;
35  }
```

10.3.1.3.2 去除模板类型参数中的引用属性

根据模板的类型参数创建对象时，需要去除其引用属性。

```
1   // convert2.cpp
2
3   // 去除模板类型参数中的引用属性
4
5   #include <iostream>
6
7   using namespace std;
8
9   class Add {
10  public:
11      Add(int x = 0, int y = 0) : x(x), y(y) {}
12
13      int calc(void) const {
14          return x + y;
15      }
16
17  private:
18      int x, y;
19  };
20
21  class Sub {
22  public:
23      Sub(int x = 0, int y = 0) : x(x), y(y) {}
24
25      int calc(void) const {
26          return x - y;
27      }
28
29  private:
30      int x, y;
31  };
32
33  // 模板的类型参数可能会是引用类型，而创建对象所需要的是原始
```

```cpp
// 类型而非引用类型，且函数的返回值亦不可能是指向引用的指针
/*
template<typename T>
T* create(int x, int y) {
    return new T(x, y);
}
*/
template<typename T>
typename add_pointer<typename remove_reference<T>::type>::type create (int x, int y) {
    return new typename remove_reference<T>::type(x, y);
}

template<typename T>
int calc(T const& c, int x, int y) {
    return create<decltype(c)>(x, y)->calc();
}

int main(void) {
    cout << create<Add>(123, 456)->calc() << endl; // 579
    cout << create<Sub>(456, 123)->calc() << endl; // 333

    Add add;
    Sub sub;

    cout << calc(add, 123, 456) << endl; // 579
    cout << calc(sub, 456, 123) << endl; // 333

    return 0;
}
```

10.3.1.3.3 在模板类型参数上添加引用属性

从模板中返回类型参数对象的引用时，需要为其添加引用属性。

```cpp
// convert3.cpp

// 在模板类型参数上添加引用属性

#include <iostream>

using namespace std;

class Add {
public:
    Add(int x = 0, int y = 0) : x(x), y(y) {}

    int calc(void) const {
        return x + y;
    }

private:
    int x, y;
};

```

```
class Sub {
public:
    Sub(int x = 0, int y = 0) : x(x), y(y) {}

    int calc(void) const {
        return x - y;
    }

private:
    int x, y;
};

// 模板的类型参数可能会是引用类型，而创建对象所需要的是原始
// 类型而非引用类型，且函数的返回值亦不可能是引用引用的引用
/*
template<typename T>
T& create(int x, int y) {
    return *new T(x, y);
}
*/
template<typename T>
typename add_lvalue_reference<typename remove_reference<T>::type>::type
create(int x, int y) {
    return *new typename remove_reference<T>::type (x, y);
}

template<typename T>
int calc(T const& c, int x, int y) {
    return create<decltype(c)>(x, y).calc();
}

int main(void) {
    cout << create<Add>(123, 456).calc() << endl; // 579
    cout << create<Sub>(456, 123).calc() << endl; // 333

    Add add;
    Sub sub;

    cout << calc(add, 123, 456) << endl; // 579
    cout << calc(sub, 456, 123) << endl; // 333

    return 0;
}
```

10.3.1.3.4 去除模板类型参数中的常量限定

```
// convert4.cpp

// 去除模板类型参数中的常量限定

#include <iostream>

using namespace std;

class Integer {
```

```
public:
    Integer(int n = 0) : n(n) {}

    Integer& operator++(void) {
        ++n;

        return *this;
    }

    friend ostream& operator<<(ostream& os, Integer const& i) {
        return os << i.n;
    }

private:
    int n;
};

// 普通

template<typename T>
T& create1(int n) {
    return *new T(n);
}

// 去除引用属性

template<typename T>
typename add_lvalue_reference<typename remove_reference<T>::type>::type
create2(int n) {
    return *new typename remove_reference<T>::type(n);
}

// 去除引用属性和常量限定

template<typename T>
typename add_lvalue_reference<typename remove_const<typename
remove_reference<T>::type>::type>::type create3(int n) {
    return *new typename remove_const<typename
remove_reference<T>::type>::type(n);
}

// 去除引用属性和CV限定

template<typename T>
typename add_lvalue_reference<typename remove_cv<typename
remove_reference<T>::type>::type>::type create4(int n) {
    return *new typename remove_cv<typename
remove_reference<T>::type>::type(n);
}

// 去除引用属性和CV限定--衰减

template<typename T>
typename add_lvalue_reference<typename decay<T>::type>::type create5(int n)
{
    return *new typename decay<T>::type(n);
```

```
}

int main(void) {
    cout << ++create1<Integer>(100) << endl;
    cout << ++create2<Integer&>(200) << endl;
    cout << ++create3<Integer const&>(300) << endl;
    cout << ++create4<Integer const volatile&>(400) << endl;

    cout << ++create5<Integer>(100) << endl;
    cout << ++create5<Integer&>(200) << endl;
    cout << ++create5<Integer const&>(300) << endl;
    cout << ++create5<Integer const volatile&>(400) << endl;

    return 0;
}
```

10.3.1.3.5 将函数类型衰减为函数指针类型

函数类型不能直接声明变量。要想用它声明变量，以便后续调用，就需要将其升级为函数对象类型，或降级为函数指针类型。

```
// convert5.cpp

// 将函数类型衰减为函数指针类型

#include <iostream>
#include <functional>

using namespace std;

template<typename Notify>
class Processor {
public:
    Processor(Notify& notify) : fobj(notify), fptr(notify) {}

    void run(void) {
        fobj(100);

        fptr(200);
    }

private:
    // Notify notify; // 函数类型不能直接声明变量
    function<Notify> fobj; // 要么升级为函数对象类型
    typename decay<Notify>::type fptr; // 要么降级为函数指针类型
};

void notify(int eventId) {
    cout << eventId << endl;
}

int main(void) {
    Processor<void (int)> processor(notify);
    processor.run();

```

```
    return 0;
}
```

10.3.1.4 根据条件使能类型

template<bool B, typename T = void> struct enable_if;

传递给模板布尔型形参B的值为...	模板实例化类中的成员类型type为...
true	传递给模板类型形参T的类型实参，缺省为void
false	无效，引发编译错误

10.3.1.4.1 enable_if的基本用法

```
// enable1.cpp

// enable_if的基本用法

#include <iostream>
#include <typeinfo>

using namespace std;

// 函数的参数必须是整数类型

template<typename T>
void foo(typename enable_if<is_integral<T>::value, T>::type arg) {
    cout << typeid(arg).name() << endl;
}

// 函数的返回值必须是整数类型

template<typename T>
typename enable_if<is_integral<T>::value, T>::type bar(void) {
    return T();
}

// 模板的参数必须是整数类型

template<typename T, typename = typename
enable_if<is_integral<T>::value>::type>
class A {
public:
    static void hum(void) {
        cout << typeid(T).name() << endl;
    }
};

// 针对类型参数取任意类型的通用版本

template<typename T, typename = void>
class B {
public:
    static void fun(void) {
```

```
        cout << "通用版本: " << typeid(T).name() << endl;
    }
};

// 针对类型参数取整数类型的特化版本

template<typename T>
class B<T, typename enable_if<is_integral<T>::value>::type> {
public:
    static void fun(void) {
        cout << "特化版本: " << typeid(T).name() << endl;
    }
};

int main(void) {
    foo<char>(1);
    // foo<double>(1.); // 编译错误

    cout << typeid(bar<short>()).name() << endl;
    // cout << typeid(bar<double>()).name() << endl; // 编译错误

    A<int>::hum();
    // A<double>::hum();

    B<long>::fun();      // ->特化版本
    B<long long>::fun(); // ->特化版本
    B<double>::fun();    // ->通用版本

    return 0;
}
```

10.3.1.4.2 基于enable_if的函数重载

一般而言，（模板）函数间的重载，需要有差异化的参数表，而enable_if则打破了这一限制。

```
// enable2.cpp

// 基于enable_if的函数重载

#include <iostream>

using namespace std;

// 整数版本

template<typename T>
typename enable_if<is_integral<T>::value, int>::type foo(T const& arg) {
    return 1;
}

// 指针版本

template<typename T>
typename enable_if<is_pointer<T>::value, int>::type foo(T const& arg) {
    return 2;
```

```
}

// 数组版本

template<typename T>
typename enable_if<is_array<T>::value, int>::type foo(T const& arg) {
    return 3;
}

// 字符串版本

template<typename T>
typename enable_if<is_same<T, string>::value, int>::type foo(T const& arg)
{
    return 4;
}

// enable_if模板第二个参数的缺省值是void，对于没有返回值的函数，该参数可省略不写

template<typename T>
typename enable_if<is_integral<T>::value>::type bar(T const& arg) {
    cout << "整数版本" << endl;
}

template<typename T>
typename enable_if<is_pointer<T>::value>::type bar(T const& arg) {
    cout << "指针版本" << endl;
}

template<typename T>
typename enable_if<is_array<T>::value>::type bar(T const& arg) {
    cout << "数组版本" << endl;
}

template<typename T>
typename enable_if<is_same<T, string>::value>::type bar(T const& arg) {
    cout << "字符串版本" << endl;
}

// 基于enable_if的函数重载，有助于降低与类型判断有关的复杂度

class A {};
class B {};
class C {};
class D {};

template<typename T>
void hum1(void) {
    if (typeid(T) == typeid(A))
        cout << 'A' << endl;
    else
    if (typeid(T) == typeid(B))
        cout << 'B' << endl;
    else
    if (typeid(T) == typeid(C))
        cout << 'C' << endl;
```

```
    else
        cout << 'X' << endl;
}

template<typename T>
typename enable_if<is_same<T, A>::value>::type hum(void) {
    cout << 'A' << endl;
}

template<typename T>
typename enable_if<is_same<T, B>::value>::type hum(void) {
    cout << 'B' << endl;
}

template<typename T>
typename enable_if<is_same<T, C>::value>::type hum(void) {
    cout << 'C' << endl;
}

template<typename T>
typename enable_if<!is_same<T, A>::value && !is_same<T, B>::value &&
!is_same<T, C>::value>::type hum(void) {
    cout << 'X' << endl;
}

int main(void) {
    char c = 'A';
    char const* p = "A";
    char a[] = "A";
    string s = "A";

    cout << foo(c) << endl;
    cout << foo(p) << endl;
    cout << foo(a) << endl;
    cout << foo(s) << endl;

    bar(c);
    bar(p);
    bar(a);
    bar(s);

    hum<A>();
    hum<B>();
    hum<C>();
    hum<D>();

    return 0;
}
```

10.3.1.5 根据条件选择类型

template<bool B, typename T, typename F> struct conditional;

传递给模板布尔型形参B的值为...	模板实例化类中的成员类型type为...
true	传递给模板第一个类型形参T的类型实参
false	传递给模板第二个类型形参F的类型实参

```cpp
// condition1.cpp

// 根据条件选择类型

#include <iostream>
#include <typeinfo>

using namespace std;

int main(void) {
    using A = conditional<true, short, float>::type; // short
    using B = conditional<false, short, float>::type; // float

    cout << typeid(A).name() << endl;
    cout << typeid(B).name() << endl;

    using C = conditional<is_integral<A>::value, long, double>::type; // long
    using D = conditional<is_integral<B>::value, long, double>::type; // double

    cout << typeid(C).name() << endl;
    cout << typeid(D).name() << endl;

    using E = conditional<sizeof(A) < sizeof(C), A, C>::type; // short
    using F = conditional<sizeof(D) < sizeof(B), B, D>::type; // double

    cout << typeid(E).name() << endl;
    cout << typeid(F).name() << endl;

    return 0;
}
```

10.3.1.6 获取返回值的类型

10.3.1.6.1 借助decltype获取返回值的类型

```cpp
// result1.cpp

// 借助decltype获取返回值的类型

#include <iostream>
#include <sstream>

using namespace std;

// 可调用对象fun的类型Fun是函数模板foo的类型参数，而foo
// 的返回类型又由fun决定，因此无法直接写出foo的返回类型
```

```
/*
template<typename Fun, typename Arg>
? foo(Fun fun, Arg arg) {
    return fun(arg);
}
*/
// 借助decltype固然可以获取函数的返回类型，但代码晦涩难懂，可读性差
/*
template<typename Fun, typename Arg>
decltype((*(Fun*)0)(*(Arg*)0)) foo(Fun fun, Arg arg) {
    return fun(arg);
}
*/
// 利用返回类型后置可以简化其书写形式

template<typename Fun, typename Arg>
auto foo(Fun fun, Arg arg) -> decltype(fun(arg)) {
    return fun(arg);
}

int s2i(string const& s) {
    return atoi(s.c_str());
}

string i2s(int i) {
    ostringstream oss;
    oss << i;
    return oss.str();
}

int main(void) {
    cout << foo(s2i, "123") << endl;
    cout << foo(i2s, 456) << endl;

    return 0;
}
```

10.3.1.6.2 借助declval和decltype获取返回值的类型

```
// result2.cpp

// 借助declval和decltype获取返回值的类型

#include <iostream>
#include <typeinfo>

using namespace std;

class A {
public:
    int operator()(int arg) {
        return arg;
    }

private:
```

```
    A(void); // 私有缺省构造函数
};

int main(void) {
    // 私有缺省构造函数导致以下代码无法通过编译

    // decltype(A()(0)) n = 0; // 编译错误

    // declval可以获取任何类型的临时变量而无需调用构造函
    // 数，这样的临时变量可以用于类型推导，但不能用于求值

    decltype(declval<A>()(declval<int>())) n = 0;
    cout << typeid(n).name() << endl;

    return 0;
}
```

10.3.1.6.3 借助result_of获取返回值的类型

假设Fun为某种可调用对象的类型，Arg1，Arg2，...，为传递给该可调用对象的各个参数的类型，则类模板result_of用Fun(Arg1,Arg2,...)实例化所得类中的成员类型type，即为该可调用对象返回值的类型。

```
// result3.cpp

// 借助result_of获取返回值的类型

#include <iostream>
#include <sstream>
#include <typeinfo>

using namespace std;

template<typename Fun, typename Arg>
typename result_of<Fun(Arg)>::type foo(Fun fun, Arg arg) {
    return fun(arg);
}

int s2i(string const& s) {
    return atoi(s.c_str());
}

string i2s(int i) {
    ostringstream oss;
    oss << i;
    return oss.str();
}

class A {
public:
    int operator()(int arg) {
        return arg;
    }

private:
    A(void); // 私有缺省构造函数
```

```
};

int main(void) {
    cout << foo(s2i, "123") << endl;
    cout << foo(i2s, 456) << endl;

    result_of<A(int)>::type n = 0;
    cout << typeid(n).name() << endl;

    return 0;
}
```

10.3.1.6.4 result_of的基本用法

```
// result4.cpp

// result_of的基本用法

#include <iostream>
#include <typeinfo>

using namespace std;

// 函数

int fun(int x) {
    cout << __FUNCTION__ << '(' << x << ")->" << flush;

    return x;
}

// 函数指针

using fun_ptr = int (*)(int);

// 函数引用

using fun_ref = int (&)(int);

class Dummy {
public:
    int fun(int x) const {
        cout << __FUNCTION__ << '(' << x << ")->" << flush;

        return x;
    }
};

// 成员函数指针

using mem_ptr = int (Dummy::*)(int) const;

// 实现了函数操作符的类

class Foo {
```

```
public:
    int operator()(int x) const {
        cout << __FUNCTION__ << '(' << x << ")->" << flush;

        return x;
    }

    double operator()(double x) const {
        cout << __FUNCTION__ << '(' << x << ")->" << flush;

        return x;
    }
};

// 可被转换为函数指针的类

class Bar {
    using hum_n = int (*)(int);
    using hum_d = double (*)(double);

public:
    operator hum_n(void) const {
        return hum;
    }

    operator hum_d(void) const {
        return hum;
    }

private:
    static int hum(int x) {
        cout << __FUNCTION__ << '(' << x << ")->" << flush;

        return x;
    }

    static double hum(double x) {
        cout << __FUNCTION__ << '(' << x << ")->" << flush;

        return x;
    }
};

int main (void) {
    cout << fun(100) << endl;

    fun_ptr pfun = &fun;
    cout << pfun(200) << endl;

    fun_ref rfun = fun;
    cout << rfun(300) << endl;

    mem_ptr pmem = &Dummy::fun;
    cout << (Dummy().*pmem)(400) << endl;

    Foo foo;
```

```
    cout << foo(500) << endl;
    cout << foo(0.5) << endl;

    Bar bar;
    cout << bar(600) << endl;
    cout << bar(0.6) << endl;

    // result_of模板的类型参数必须是一个能被实例化为可调用对象的类型
    // 及传递给该可调用对象的各个参数的类型。可调用对象包括函数指针、
    // 函数引用、实现了函数操作符的类对象和可被转换为函数指针的类对
    // 象，但函数类型不能被实例化为对象，因此下面的写法无法通过编译

    // cout << typeid(result_of<decltype(fun)(int)>::type).name() << endl;

    // 只有将其转换为函数指针或引用，才能作为参数传给result_of模板

    cout << typeid(result_of<decltype(fun)*(int)>::type).name() << endl;
    cout << typeid(result_of<decay<decltype(fun)>::type(int)>::type).name()
<< endl;
    cout << typeid(result_of<decltype(fun)&(int)>::type).name() << endl;

    cout << typeid(result_of<fun_ptr(int)>::type).name() << endl;
    cout << typeid(result_of<fun_ref(int)>::type).name() << endl;
    cout << typeid(result_of<mem_ptr(Dummy*,int)>::type).name() << endl;
    cout << typeid(result_of<Foo(int)>::type).name() << endl;
    cout << typeid(result_of<Foo(double)>::type).name() << endl;
    cout << typeid(result_of<Bar(int)>::type).name() << endl;
    cout << typeid(result_of<Bar(double)>::type).name() << endl;

    return 0;
}
```

10.3.1.6.5 result_of的应用案例

```
// result5.cpp

// result_of的应用案例

#include <iostream>
#include <vector>
#include <map>

using namespace std;

class Student {
public:
    Student(string const& name, int age, float score)
        : name(name), age(age), score(score) {}

    string getName(void) const {
        return name;
    }

    int getAge(void) const {
        return age;
```

```
    }

    float getScore(void) const {
        return score;
    }

    friend ostream& operator<<(ostream& os, Student const& s) {
        return os << '(' << s.name << ',' << s.age << ',' << s.score << ')';
    }

private:
    string name;
    int age;
    float score;
};

// 函数orderBy所返回的多重映射以参数向量students中每个Student类型元素的某个成员变
// 量作为键，而将该元素作为与之相对应的值。究竟选取哪个成员变量则由（成员）函数指
// 针参数key决定。该指针目标函数的返回类型即为orderBy函数所返回多重映射中键的类型

template<typename Key>
multimap<typename result_of<Key(Student*)>::type, Student> orderBy (
    vector<Student> const& students, Key key) {
    multimap<typename result_of<Key(Student*)>::type, Student> mm;

    for (auto const& student : students)
        mm.insert(make_pair((student.*key)(), student));

    return mm;
}

int main (void) {
    vector<Student> students {
        {"zhangfei", 25, 90},
        {"zhaoyun", 22, 80},
        {"guanyu", 30, 90},
        {"huangzhong", 25, 80},
        {"machao", 22, 70}};

    for (auto const& kv : orderBy(students, &Student::getName))
        cout << kv.second;
    cout << endl;

    for (auto const& kv : orderBy(students, &Student::getAge))
        cout << kv.second;
    cout << endl;

    for (auto const& kv : orderBy(students, &Student::getScore))
        cout << kv.second;
    cout << endl;

    return 0;
}
```

10.3.2 变参模板

10.3.2.1 函数模板

10.3.2.1.1 带有可变参数表的函数模板

在C++98/03中，函数模板只能带有固定数量的模板参数，C++11放开了这一限制，允许为函数模板声明0到任意多个模板参数，其形式就是在typename或class关键字后面写上省略号“...”。形如：

```
template<typename... TYPES>
返回类型 函数模板名(调用形参表) {
    函数体
}
```

```
// vaftmpl1.cpp

// 带有可变参数表的函数模板

#include <iostream>

using namespace std;

template<typename... TYPES>
void foo(TYPES... args) {
    // args是由0到任意多个不同类型的调用参数组成的参数
    // 包。打印函数模板调用参数（亦即模板参数）的个数

    cout << sizeof...(args) << endl;
}

int main(void) {
    foo();
    foo(123);
    foo(123, 4.56);
    foo(123, 4.56, "789");

    return 0;
}
```

10.3.2.1.2 借助递归调用展开参数包

借助递归调用展开带有可变参数表的函数模板的参数包，需要提供一个参数包展开函数，和一个递归终止函数，后者是用来终止递归的。

```
// vaftmpl2.cpp

// 借助递归调用展开参数包

#include <iostream>
#include <typeinfo>

using namespace std;

// 递归终止函数
```

```cpp

void foo(void) {
    cout << "展开终止" << endl;
}

// 参数包展开函数

template<typename HEAD, typename... REST>
void foo(HEAD head, REST... rest) {
    cout << head << ':' << typeid(head).name() << endl;

    // 参数包rest在展开过程中递归调用自身，每调用一次，参数包中的参数就减少一个，直到所
    // 有的参数都被展开为止。当参数包为空时，调用非模板形式的递归终止函数，结束递归过程

    foo(rest...);
}

int main(void) {
    foo();
    foo(123);
    foo(123, 4.56);
    foo(123, 4.56, "789");

    return 0;
}
```

借助泛型元组和类型萃取也可以达到同样的效果。

```cpp
// vaftmpl3.cpp

// 借助递归调用展开参数包

#include <iostream>
#include <typeinfo>
#include <tuple>

using namespace std;

// 递归终止函数

template<size_t I = 0, typename TUPLE>
typename enable_if<I == tuple_size<TUPLE>::value>::type bar(TUPLE tuple) {
    // 当参数索引等于参数个数时，递归终止

    cout << "展开终止" << endl;
}

// 参数包展开函数

template<size_t I = 0, typename TUPLE>
typename enable_if<I < tuple_size<TUPLE>::value>::type bar(TUPLE tuple) {
    // 当参数索引小于参数个数时，取当前索引位置的参数

    cout << get<I>(tuple) << ':' << typeid(get<I>(tuple)).name() << endl;

```

```
    // 然后通过递增参数索引来选择不同的函数版本

    bar<I+1>(tuple);
}

template<typename... TYPES>
void foo(TYPES... args) {
    // 先将由不同类型不定个数的调用参数组成的参数包转换为泛型元组

    bar(make_tuple(args...));
}

int main(void) {
    foo();
    foo(123);
    foo(123, 4.56);
    foo(123, 4.56, "789");

    return 0;
}
```

10.3.2.1.3 借助逗号表达式和列表初始化展开参数包

所谓逗号表达式，就是形如“表达式1, 表达式2, 表达式3 ... 表达式n”的表达式。逗号表达式的求值过程是，从左到右依次计算每个表达式的值，以最后一个表达式的值作为整个逗号表达式的值。

```
// vaftmpl4.cpp

// 借助逗号表达式和列表初始化展开参数包

#include <iostream>
#include <typeinfo>

using namespace std;

template<typename T>
void bar(T arg) {
    cout << arg << ':' << typeid(arg).name() << endl;
}

template<typename... TYPES>
void foo(TYPES... args) {
    int arr[]{(bar(args), 0)...};

    // 以上列表初始化将被编译器展开成如下形式:
    //
    // int arr[]{(bar(args里的第1个参数), 0), (bar(args里的第2个参数), 0), ...};
    //
    // 其中的每一个初值都是一个被圆括号括起来的逗号表达式。为了计
    // 算该逗号表达式的值，会首先用参数包args里的第n个参数调用bar
    // 函数，并取逗号后面的0作为该逗号表达式的值。最后，在arr被初
    // 始化为一个全零数组的同时，参数包中的每一个参数都得到了处理

    cout << "展开终止" << endl;
}
```

```
30
31  int main(void) {
32      foo();
33      foo(123);
34      foo(123, 4.56);
35      foo(123, 4.56, "789");
36
37      return 0;
38  }
```

也可以用初始化列表代替数组。

```
1   // vaftmpl5.cpp
2
3   // 借助逗号表达式和列表初始化展开参数包
4
5   #include <iostream>
6   #include <typeinfo>
7
8   using namespace std;
9
10  template<typename T>
11  void bar(T arg) {
12      cout << arg << ':' << typeid(arg).name() << endl;
13  }
14
15  template<typename... TYPES>
16  void foo(TYPES... args) {
17      initializer_list<int>{(bar(args), 0)...};
18
19      cout << "展开终止" << endl;
20  }
21
22  int main(void) {
23      foo();
24      foo(123);
25      foo(123, 4.56);
26      foo(123, 4.56, "789");
27
28      return 0;
29  }
```

甚至可以用lambda表达式代替有名函数bar。

```
1   // vaftmpl6.cpp
2
3   // 借助逗号表达式和列表初始化展开参数包
4
5   #include <iostream>
6   #include <typeinfo>
7
8   using namespace std;
9
10  template<typename... TYPES>
```

```
void foo(TYPES... args) {
    initializer_list<int>{([&] {
        cout << args << ':' << typeid(args).name() << endl; }(), 0)...};

    cout << "展开终止" << endl;
}

int main(void) {
    foo();
    foo(123);
    foo(123, 4.56);
    foo(123, 4.56, "789");

    return 0;
}

// 本程序代码无法使用GNU C++编译器编译通过，建议使用Visual Studio 2013以上版本编译器
```

10.3.2.2 类模板

10.3.2.2.1 带有可变参数表的类模板

在C++98/03中，类模板只能带有固定数量的模板参数，C++11放开了这一限制，允许为类模板声明0到任意多个模板参数，其形式就是在typename或class关键字后面写上省略号“...”。形如：

```
template<typename... TYPES>
class 类模板名 [: 继承表] {
    成员类型
    成员函数
    成员变量
};
```

```
// vactmpl1.cpp

// 带有可变参数表的类模板

#include <iostream>

using namespace std;

template<typename... TYPES>
class Foo {
public:
    Foo(TYPES... args) {
        // args是由0到任意多个不同类型的调用参数组成的参数
        // 包。打印构造函数调用参数（亦即模板参数）的个数

        cout << sizeof...(args) << endl;
    }
};

int main(void) {
    Foo<> a;
    Foo<int> b(123);
    Foo<int, double> c(123, 4.56);
```

```
    Foo<int, double, char const*> d(123, 4.56, "789");

    return 0;
}
```

10.3.2.2.2 借助模板特化展开参数包

借助模板特化展开带有可变参数表的类模板的参数包，需要提供一个接受任意多个模板参数的通用版本、一个接受至少一个模板参数的特化版本，和一个不接受任何模板参数的特化版本。

```
// vactmpl2.cpp

// 借助模板特化展开参数包

#include <iostream>
#include <typeinfo>

using namespace std;

// 接受任意多个模板参数的通用版本

template<typename... TYPES> class Foo; // 只需声明，无需定义

// 接受至少一个模板参数的特化版本

template<typename HEAD, typename... REST>
class Foo<HEAD, REST...> {
public:
    Foo(HEAD head, REST... rest) : head(head), rest(rest...) {
        cout << head << ':' << typeid(head).name() << endl;
    }

private:
    HEAD head;

    // 参数包REST在展开过程中实例化类模板自身，每实例化一次，
    // 参数包中的参数就减少一个，直到所有的参数都被展开为止。
    // 当参数包为空时，将会选择不接受任何类型参数的特化版本

    Foo<REST...> rest;
};

// 不接受任何模板参数的特化版本

template<> class Foo<> {
public:
    Foo(void) {
        cout << "展开终止" << endl;
    }
};

int main(void) {
    Foo<> a;
    Foo<int> b(123);
    Foo<int, double> c(123, 4.56);
```

```
    Foo<int, double, char const*> d(123, 4.56, "789");

    return 0;
}
```

使用继承语法也可以达到同样的效果。

```
// vactmpl3.cpp

// 借助模板特化展开参数包

#include <iostream>
#include <typeinfo>

using namespace std;

// 接受任意多个模板参数的通用版本

template<typename... TYPES> class Foo; // 只需声明，无需定义

// 接受至少一个模板参数的特化版本

template<typename HEAD, typename... REST>
class Foo<HEAD, REST...> : public Foo<REST...> {
    // 参数包REST在展开过程中实例化类模板自身，每实例化一次，
    // 参数包中的参数就减少一个，直到所有的参数都被展开为止。
    // 当参数包为空时，将会选择不接受任何类型参数的特化版本

public:
    Foo(HEAD head, REST... rest) : head(head), Foo<REST...>(rest...) {
        cout << head << ':' << typeid(head).name() << endl;
    }

private:
    HEAD head;
};

// 不接受任何模板参数的特化版本

template<> class Foo<> {
public:
    Foo(void) {
        cout << "展开终止" << endl;
    }
};

int main(void) {
    Foo<> a;
    Foo<int> b(123);
    Foo<int, double> c(123, 4.56);
    Foo<int, double, char const*> d(123, 4.56, "789");

    return 0;
}
```

10.3.2.2.3 借助泛型元组和索引生成器展开参数包

```cpp
// vactmpl4.cpp

// 借助泛型元组和索引生成器展开参数包

#include <iostream>
#include <typeinfo>
#include <tuple>

using namespace std;

// 索引序列

template<int...> class IndexSequence {};

// 索引生成器通用版本

template<int N, int... INDEXES>
class IndexMaker {
public:
    using Indexes = typename IndexMaker<N - 1, N - 1, INDEXES...>::Indexes;
};

// 索引生成器N取0的特化版本

template<int... INDEXES>
class IndexMaker<0, INDEXES...> {
public:
    using Indexes = IndexSequence<INDEXES...>;
};

template<typename... TYPES>
class Foo {
public:
    Foo(TYPES... args) {
        printArgs(typename IndexMaker<sizeof...(args)>::Indexes(),
make_tuple(args...));
    }

    // 打印参数元组

    template<int... INDEXES>
    void printArgs(IndexSequence<INDEXES...>, tuple<TYPES...>&& args) {
        initializer_list<int>{(printArg(get<INDEXES>(args)), 0)...};

        cout << "展开终止" << endl;
    }

    // 打印单个参数

    template<typename T>
    void printArg(T arg) {
        cout << arg << ':' << typeid(arg).name() << endl;
    }
```

```
};

int main(void) {
    Foo<> a;
    Foo<int> b(123);
    Foo<int, double> c(123, 4.56);
    Foo<int, double, char const*> d(123, 4.56, "789");

    return 0;
}
```

10.3.2.3 通用打印函数

10.3.2.3.1 C++98/03中的通用打印函数

```
// print1.cpp

// C++98/03中的通用打印函数

#include <iostream>

using namespace std;

// 几乎完全相同的模板定义代码大量重复

template<typename TYPE>
void print(TYPE arg) {
    cout << arg << endl;
}

template<typename TYPE1, typename TYPE2>
void print(TYPE1 arg1, TYPE2 arg2) {
    cout << arg1 << ' ' << arg2 << endl;
}

template<typename TYPE1, typename TYPE2, typename TYPE3>
void print(TYPE1 arg1, TYPE2 arg2, TYPE3 arg3) {
    cout << arg1 << ' ' << arg2 << ' ' << arg3 << endl;
}

int main(void) {
    print(123);
    print(123, 4.56);
    print(123, 4.56, "789");
    // print(123, 4.56, "789", '0'); // 最多只能支持三个任意类型数据的打印

    return 0;
}
```

10.3.2.3.2 C++11中的通用打印函数

```
// print2.cpp

// C++11中的通用打印函数

```

```cpp
#include <iostream>

using namespace std;

// 代码简洁优雅，完全消除重复

void print(void) {
    cout << endl;
}

template<typename HEAD, typename... REST>
void print(HEAD head, REST... rest) {
    cout << head << ' ';

    print(rest...);
}

int main(void) {
    print(123);
    print(123, 4.56);
    print(123, 4.56, "789");
    print(123, 4.56, "789", '0'); // 支持任意多个任意类型数据的打印

    return 0;
}
```

10.3.2.4 通用工厂

10.3.2.4.1 C++98/03中的通用工厂

```cpp
// factory1.cpp

// C++98/03中的通用工厂

#include <iostream>

using namespace std;

// 几乎完全相同的模板定义代码大量重复

template<typename CLASS, typename TYPE>
CLASS* create(TYPE arg) {
    return new CLASS(arg);
}

template<typename CLASS, typename TYPE1, typename TYPE2>
CLASS* create(TYPE1 arg1, TYPE2 arg2) {
    return new CLASS(arg1, arg2);
}

template<typename CLASS, typename TYPE1, typename TYPE2, typename TYPE3>
CLASS* create(TYPE1 arg1, TYPE2 arg2, TYPE3 arg3) {
    return new CLASS(arg1, arg2, arg3);
}

```

```
class One {
public:
    One(int arg) : var(arg) {}

    friend ostream& operator<<(ostream& os, One const& one) {
        return os << one.var;
    }

private:
    int var;
};

class Two {
public:
    Two(int arg1, double arg2) : var1(arg1), var2(arg2) {}

    friend ostream& operator<<(ostream& os, Two const& two) {
        return os << two.var1 << ' ' << two.var2;
    }

private:
    int var1;
    double var2;
};

class Three {
public:
    Three(int arg1, double arg2, char const* arg3)
        : var1(arg1), var2(arg2), var3(arg3) {}

    friend ostream& operator<<(ostream& os, Three const& three) {
        return os << three.var1 << ' ' << three.var2 << ' ' << three.var3;
    }

private:
    int var1;
    double var2;
    char const* var3;
};

class Four {
public:
    Four(int arg1, double arg2, char const* arg3, char arg4)
        : var1(arg1), var2(arg2), var3(arg3), var4(arg4) {}

    friend ostream& operator<<(ostream& os, Four const& four) {
        return os << four.var1 << ' ' << four.var2 << ' ' <<
            four.var3 << ' ' << four.var4;
    }

private:
    int var1;
    double var2;
    char const* var3;
    char var4;
};
```

```
82
83  int main(void) {
84      One* one = create<One>(123);
85      cout << *one << endl;
86
87      Two* two = create<Two>(123, 4.56);
88      cout << *two << endl;
89
90      Three* three = create<Three>(123, 4.56, "789");
91      cout << *three << endl;
92
93      // Four* four = create<Four>(123, 4.56, "789", '0'); // 最多支持三个构造实参
94      // cout << *four << endl;
95
96      return 0;
97  }
```

10.3.2.4.2 C++11中的通用工厂

```
1   // factory2.cpp
2
3   // C++11中的通用工厂
4
5   #include <iostream>
6
7   using namespace std;
8
9   // 代码简洁优雅，完全消除重复
10  /*
11  template<typename CLASS, typename... TYPES>
12  CLASS* create(TYPES... args) {
13      return new CLASS(args...);
14  }
15  */
16  // 更优化的版本
17
18  template<typename CLASS, typename... TYPES>
19  CLASS* create(TYPES&&... args) { // 引用传参，避免复制，提高性能
20      return new CLASS(forward<TYPES>(args)...); // 通过完美转发保持参数左右值引用不
    变
21  }
22
23  class One {
24  public:
25      One(int arg) : var(arg) {}
26
27      friend ostream& operator<<(ostream& os, One const& one) {
28          return os << one.var;
29      }
30
31  private:
32      int var;
33  };
34
35  class Two {
```

```
public:
    Two(int arg1, double arg2) : var1(arg1), var2(arg2) {}

    friend ostream& operator<<(ostream& os, Two const& two) {
        return os << two.var1 << ' ' << two.var2;
    }

private:
    int var1;
    double var2;
};

class Three {
public:
    Three(int arg1, double arg2, char const* arg3)
        : var1(arg1), var2(arg2), var3(arg3) {}

    friend ostream& operator<<(ostream& os, Three const& three) {
        return os << three.var1 << ' ' << three.var2 << ' ' << three.var3;
    }

private:
    int var1;
    double var2;
    char const* var3;
};

class Four {
public:
    Four(int arg1, double arg2, char const* arg3, char arg4)
        : var1(arg1), var2(arg2), var3(arg3), var4(arg4) {}

    friend ostream& operator<<(ostream& os, Four const& four) {
        return os << four.var1 << ' ' << four.var2 << ' ' <<
            four.var3 << ' ' << four.var4;
    }

private:
    int var1;
    double var2;
    char const* var3;
    char var4;
};

int main (void) {
    One* one = create<One>(123);
    cout << *one << endl;

    Two* two = create<Two>(123, 4.56);
    cout << *two << endl;

    Three* three = create<Three>(123, 4.56, "789");
    cout << *three << endl;

    Four* four = create<Four>(123, 4.56, "789", '0'); // 支持任意多个构造实参
    cout << *four << endl;
```

```
92
93      return 0;
94  }
```

10.3.3 综合应用

10.4 杜绝泄漏

10.4.1 共享指针

10.4.2 独占指针

10.4.3 虚弱指针

10.4.4 第三方库

10.5 简化线程

10.5.1 线程模型

10.5.2 互斥量

10.5.3 条件变量

10.5.4 原子变量

10.5.5 只调一次

10.5.6 异步操作

10.5.7 异步任务

10.6 实用工具

10.6.1 日期时间

10.6.2 数串互转

10.6.3 宽窄字符

10.7 其它特性

10.7.1 委托构造

10.7.2 继承构造

10.7.3 原始字符

10.7.4 阻断继承

10.7.5 显式覆盖

10.7.6 内存对齐

10.7.7 泛型算法