

程序设计实习（实验班-2024春）

面向对象编程：泛型

授课教师：姜少峰

助教：冯施源 吴天意

Email: shaofeng.jiang@pku.edu.cn

泛型：一段代码可以用于多种类型

场景

- 很多操作对于不同变量类型事实上是相同的
 - 例如求max、排序等，对于int, double, float参数都可以用同样代码实现
- 但是直接实现可能需要针对每种类型写一个版本的（重载）函数
 - 例如可能需要写int max(int , int), double max(double, double)
- 泛型：写类/函数的时候，用一个抽象的类型，在具体调用时再替代成具体类型
 - 因此，对不同类型、但同功能实现的情况 只需写一个版本的程序

函数模板

函数模板

格式：

T是一个类型符号，指代一个类型

返回值、形参表和函数体内都可以使用T作为类型

`template<typename T> return-type func(...)`

```
template<typename T>
const T& max(const T& t1, const T& t2) {
    if (t1 < t2) return t2;
    return t1;
}
```

这里需要T类型重载了<操作符

```
template<typename T>
void swap(T& t1, T& t2) {
    T tmp = std::move(t1);
    t1 = std::move(t2);
    t2 = std::move(tmp);
}
```

等价写法（<typename T>等价于<class T>）：

`template<class T> return-type func(...)`

更一般地

类型参数可以有多个

T1, T2这些名字可以替换成任何合法的变量名, 例如Type1和Type2

`template<typename T1, ..., typename Tn> return-type func(...)`

这几个typename也都可以替换成class

允许传入两个不同类型参数

```
template<typename T1, typename T2>
T1 add(const T1& t1, const T2& t2) {
    std::cout << "T1 T2" << std::endl;
    return (T1)(t1 + t2);
}
```

如何调用？

```
template<class T1, class T2>
T1 add(const T1& t1, const T2& t2) {
    return (T1)(t1 + t2);
}
```

- 显式调用：函数名<类型参数表>(参数表)
- 隐式调用：不指定<类型参数表>，当做正常函数使用，类型由编译器匹配

具体规则下页介绍

```
int main() {
    add<int,int>(1, 1); // 显式类型匹配
    add(1.0, 1); // 隐式类型匹配
    return 0;
}
```

隐式参数匹配规则

编译时如何确定类型？

这里匹配是指
不发生类型转换

- 优先匹配具体/非模版类型，然后匹配模板类型
 - 找到匹配且同优先级无歧义则可以调用
- 否则经过类型转换找匹配的非模板函数
- 都不能找到或有歧义则报错

```
template<class T1, class T2>
T1 add(const T1& t1, const T2& t2) {
    std::cout << "T1 T2" << std::endl;
    return (T1)(t1 + t2);
}
double add(double t1, double t2) {
    std::cout << "double double" << std::endl;
    return t1 + t2;
}
int main() {
    add(1.0, 1);
    add(1.0, 1.0);
    return 0;
}
```

输出什么？

更复杂的例子

```
template<class T>
T add(const T& t1, const T& t2) {
    std::cout << "T T" << std::endl;
    return t1 + t2;
}
template<class T>
T add(int t1, const T& t2) {
    std::cout << "int T" << std::endl;
    return (T)(t1 + t2);
}
template<class T1, class T2>
T1 add(const T1& t1, const T2& t2) {
    std::cout << "T1 T2" << std::endl;
    return (T1)(t1 + t2);
}
double add(double t1, double t2) {
    std::cout << "double double" << std::endl;
    return t1 + t2;
}
```

```
int main() {
    //add(1, 1); // 编译错误: 可以匹配(T, T)和(int, T)
    add<int, int>(1, 1); // 显式指定类型匹配T1 T2版本, 输出T1 T2
    add(1.0, 1); // 输出 T1 T2
    add(1, 1.0); // 输出 int T
    add(1.0, 1.0); // 输出 double double
    add(1LL, 1LL); // 输出T T
    return 0;
}
```

调用时发生什么？

编译时先发生类型替换

- 在编译时根据具体调用时的参数类型将抽象类型 T_1, \dots, T_n 替换成对应类型
- 替换后，相当于创建了若干新的、带有具体类型的函数（理解为创建重载）
- 此时所有函数都是具体类型的正常函数，之后进行正常编译过程

将函数模板替换成正常函数后，这个函数叫做模板函数

替换过程中有歧义失败，或者替换后产生了错误，都触发编译错误

- 注：绝非运行时决定的，因此需要在编译时给编译器提供足够的信息确定类型

注：此处只能匹配编译时类型，不处理动态绑定

对特定参数类型采用特殊实现

- 场景：当T等于某些特殊类型时，需要做特殊处理
- 例：写一个print函数，对bool型替换成true/false输出，其他原样输出
- 写法：重载，但是template<>要留空，之后函数名称填上要指定的特殊类型

bool型cout默认输出0/1

```
template<typename T>
void print(const T& t) {
    std::cout << t << std::endl;
}
```

注意这种写法；template后面的<>不可省略

```
template<>
void print<bool>(const bool& t) {
    std::cout << (t ? "true" : "false") << std::endl;
}
```

可否用动态绑定实现？异同？

```
int main() {
    print(1.5); // 输出1.5
    print(true); // 输出true
    return 0;
}
```

可用于实现如下根据类型进行的switch-case

```
switch (typeof(t)) {
    case int: ...
    case string:...
```

C++并不直接有可用的typeof，所以这里介绍的方法实现更合适

类模板

类模板

- 在类的定义里也可以有泛型参数，并且类似于函数模板，在编译时进行替换
 - 尤其适合实现功能性的类，比如各种算法、数据结构
- 格式：

这里typename关键字也可以等价替换成class关键字

```
template <typename T1, ..., typename Tn>
```

```
class ClassName {
```

T1, ..., Tn类型可以在类内任意使用

这个class是定义类用的，不可用typedef替换

```
};
```

类模板的使用和实现

- 与函数模板不同，类模板使用时一定要显式指定类型

与类模板声明时给出的列表
按顺序进行对应

- 语法：定义对象时，在类名后面加<类型1, 类型2,..., 类型n>
- 例如X类需要一个类型参数T，那么声明X变量时需要X<int> x;

- 声明与定义可以分开，但定义的格式是：

类似X<int>这种对类模板X的具体化
统称为X的模板类

`template<typename T1, ..., typename Tn>`

`return-type` `ClassName<T1, ..., Tn>::func(参数表)`

除了继续需要`template<class...>`声明外，类名后面也需要加上类型列表<T1,...Tn>

例子

```
template<typename T>
class DArray {
    T* arr;
    int cap; int len;
    void enlarge();
public:
    DArray (int _cap) {
        cap = _cap; len = 0; arr = new T[cap];
    }
    DArray () : DArray(10) {}
    ~DArray() {delete[] arr;}
    void add(const T&);
    T& operator[](int i);
};
```

这里构造、析构函数比较短，为了slides排版直接就地实现
一般来说这里应该只声明不定义

定义对象时需要<类型列表>
显式指定类型

```
int main() {
    DArray<int> arr(10);
    for (int i = 0; i < 100; i++)
        arr.add(i);
    cout << arr[50] << endl;
    return 0;
}
```

```
template<typename T>
void DArray<T>::enlarge() {
    T* n_arr = new T[2 * cap];
    for (int i = 0; i < cap; i++)
        n_arr[i] = arr[i];
    delete[] arr;
    cap *= 2; arr =
}

template<typename T>
void DArray<T>::add(const T& t) {
    if (len == cap) enlarge();
    arr[len++] = t;
}

template<typename T>
T& DArray<T>::operator[](int i) {return arr[i];}
```

不要忘记类名后面需要再次
加上类型列表<T>

若这段定义放在单独的darray.cpp中会链接时错误！
声明与定义分开但放在同一个(.h)文件则无问题

为什么不能把类模板的定义放在单独cpp里？

- C++编译器先做编译再做链接

非必需不会主动寻找定义
例如，只需要有void f()的声明就允许调用f

- 编译：以文件为单元（先将include展开），将各个函数转化成机器码
- 链接：将函数的机器码链接起来，若用到的函数只有声明没有定义则报错
- 类模板不是类（函数模板也不是函数），只是对一族类和函数的描述
- 编译进行之前需要对模板先做类型替换，得到真正的类，之后才是正常编译
- 因此：若实现放在单独cpp里，编译阶段无足够信息得到该定义所应套用的类型

因为.h和.cpp是不同的编译单元
最后会触发链接时错误

显式声明

一种将类模板.h与.cpp分离的方法

- 首先：声明都放darray.h，实现都放darray.cpp

格式：template class 类名<类型>;

- 然后：在darray.cpp结尾将可能要用到的具体类型的darray声明出来

```
#ifndef _DARRAY_H
#define _DARRAY_H

template<typename T>
class DArray {
    T* arr;
    int cap; int len;
    void enlarge();
public:
    DArray (int _cap) {
        cap = _cap; len = 0; arr = new T[cap];
    }
    DArray () : DArray(10) {}
    ~DArray() {delete[] arr;}
    void add(const T&);
    T& operator[](int i);
};

#endif
```

darray.h

```
#include "darray.h"

template<typename T>
void DArray<T>::enlarge() {
    T* n_arr = new T[2 * cap];
    for (int i = 0; i < cap; i++)
        n_arr[i] = arr[i];
    delete[] arr;
    cap *= 2; arr = n_arr;
}

template<typename T>
void DArray<T>::add(const T& t) {
    if (len == cap) enlarge();
    arr[len++] = t;
}

template<typename T>
T& DArray<T>::operator[](int i) {return arr[i];}

template class DArray<int>;
template class DArray<double>;
```

darray.cpp

注意这个显式声明的写法

显式声明 (cont.)

其他优点

但不会引起程序运行效率问题

- 提升编译效率：不使用显式声明可能会让编译过程效率变低
 - 原因：不显式声明导致所有模板类对象，即使套用同类型，也需要分开解析
 - 例如 `DArray<int> a; DArray<int> b;` 这2个对象要独立运行编译解析
- 更重要：若有两个类互相依赖，则一般需要定义与实现分离（需要显式声明）

显式声明 (cont.)

相互依赖的类的实现例子

DataHandler存一组Data, 进行一些操作, 并能打印所有data的内容

- 泛型T类型的Data类和DataHandler类相互依赖

```
#ifndef _DATA_H_
#define _DATA_H_

#include <string>

template<class T> class DataHandler;

template<class T>
struct Data {
    int id;
    T content;
    DataHandler<T>* handler;
    Data(int, const T&, DataHandler<T>*),
    std::string getHandlerDescription();
};

#endif
```

data.h

Data存content和id, 也存自己从属的DataHandler, 并能得到DataHandler的描述

```
#include "data.h"
#include "data_handler.h"

template<class T>
Data<T>::Data(int _id, const T& _cont, DataHandler<T>* _h)
    : id(_id), content(_cont), handler(_h) {}

template<class T>
std::string Data<T>::getHandlerDescription() {
    return handler->description;
}

template class Data<int>;
```

data.cpp

```
#ifndef _DATA_HANDLER_H_
#define _DATA_HANDLER_H_

#include <string>

template<class T> class Data;
template<class T>
struct DataHandler {
    std::string description;
    Data<T>* arr[1000]; int len;
    DataHandler(const std::string&);
    void add(const T&);
    void printContent();
};

#endif
```

data_handler.h

```
#include "data.h"
#include "data_handler.h"
#include <iostream>

template<class T>
void DataHandler<T>::printContent() {
    for (int i = 0; i < len; i++)
        std::cout << arr[i]->content << std::endl;
}

template<class T>
void DataHandler<T>::add(const T& t) {
    arr[len] = new Data<T>(len, t, this);
    len++;
}

template<class T>
DataHandler<T>::DataHandler(const std::string& d) : description(d) {}

template class DataHandler<int>;
```

data_handler.cpp

显式声明模板类型

显式声明 (cont.)

友元

为了更好的封装性，上页的例子可以把DataHandler声明成Data的友元

```
#ifndef _DATA_H_
#define _DATA_H_

#include <string>

template<class T> class DataHandler;

template<class T>
class Data {
    int id;
    T content;
    DataHandler<T>* handler;
    friend class DataHandler<T>;
public:
    Data(int, const T&, DataHandler<T>*);
    std::string getHandlerDescription();
};

#endif
```

显式声明 (cont.)

例如这个例子中，darray.cpp只有double和int型，则main里定义DArray<bool> da变量会链接时错误

- 问题：没有显式声明的类型都不能使用

```
#include "darray.h"
/* ... */
template class DArray<int>;
template class DArray<double>;
```

- 即一旦有显式声明，则编译器不再尝试一般的类型匹配，仅从显式声明里找
- 解决方法：
 - 若darray.cpp允许改动，则可以把需要的类型都加上去
 - 若.cpp不允许改动，可以另外创建一个darray_impl.cpp，加上需要的类型

注意要先include darray.cpp

```
#include "darray.cpp"

template class DArray<bool>;
template class DArray<long long>;
```

建立darray_impl.cpp
追加需要用的类型

类模板的声明与定义

总结

与普通类不同，cpp文件总是要提供的，无法作为二进制提前编译出来

- 因为类模板不是类，所有定义需要替换成具体类型才能实施编译

声明与定义：

- 要么把定义直接放在和声明同一个.h文件中
- 要么分开并将类型的显式声明放在定义的文件末尾（或追加在XXX_impl.cpp）

无显著缺点（推荐采用）
且有时是必须采用的（互相依赖的类模板）

类模板作为友元

两种情况

- 情况一：作为友元的类模板具有指定类型

```
template<class T> class B;
```

```
template<class T>
class A {
    T data;
    friend class B<T>;
};
```

```
template<class T>
struct B {
    void f() {
        A<int> a;
        cout << a.data << endl;
        A<double> b;
        cout << b.data << endl;
    }
};
```

friend生效，
正常输出

这里是与A相同的泛型T

(当然诸如B<int>是允许的，但这样就不是类模板而是正常类)

```
int main() {
    B<int> b;
    b.f();
    return 0;
}
```

编译错误：因为main中定义的B是int型的，
friend关系只能匹配到int，匹配不到double

类模板作为友元

两种情况

- 情况二：作为友元的类模板是泛型，可以匹配任何类型

```
template<class T> class B;
```

```
template<class T>  
class A {  
    T data;
```

```
    template<class T1> friend class B;  
};
```

```
template<class T>  
struct B {  
    void f() {  
        A<int> a;  
        cout << a.data << endl;  
        A<double> b;  
        cout << b.data << endl;  
    }  
};
```

注意这种定义泛型友元的写法！

这里泛型参数不能是T，因为A的泛型已经是T了

```
int main() {  
    B<int> b;  
    b.f();  
    return 0;  
}
```

编译通过：虽然main中定义的B是int型的，但friend关系可以匹配到任何A的类型

类模板的默认类型参数

与默认参数的规则类似，只能为最后的连续若干参数提供默认值

- 可以用`template<typename T = 类型>`来指定默认类型，例如

```
template<typename T = int>
struct X {
    T data;
};
```

- 之后使用默认类型的格式：`X<> x;`

虽然<>里面是空的，但是不可以省略

类模板与静态成员

- 类模板也可以有静态成员：类型参数完全相同的模版类才共享同一份static成员
- 如X<double>和X<int>为不同类型，里面的static int是独立的
- 静态成员的定义依然必须与声明分开

```
template<class T>
struct Point {
    T x, y;
    static int memberCount;
    Point(const T& _x, const T& _y) : x(_x), y(_y){
        memberCount++;
    }
    ~Point() {memberCount--;}
};
```

```
template<class T> int Point<T>::memberCount = 0;
//template<> int Point<int>::memberCount = 0;
//template<> int Point<double>::memberCount = 0;
```

```
int main() {
    Point<int> a(0, 1), b(1, 1);
    cout << Point<int>::memberCount << endl;
    Point<double> c(0, 0);
    cout << Point<double>::memberCount << endl;
    return 0;
}
```

分别输出2 1

对static变量的定义：泛型写法

也可采用显式类型，参考这个注释中的写法；注意template<>

函数模板也可以作为类模板成员

- 下面例子中，重载了一个接受泛型T1的下标操作符[]
 - 例如可以用T1类型的hash()来作为下标（这个hash需要不造成数组越界）

```
template<class T>
class DArray {
public:
    void add(const T&);
    T& operator[](int i);
    template<class T1>
    T& operator[](const T1& t) {
        return arr[t.hash()];
    }
};
```

用T1类型的hash作为下标

模板类型声明中也可以有非泛型参数

- 这种用法是允许的：

```
template<class T, int size>
struct Array {
    int arr[size];
};
```

- 然而：定义具体对象时传入的size必须是常量
 - 不允许cin >> size后Array<int, size>这种操作
- 因此：这种用法一般没有必要，因为总可以用一个变量const int size来代替

类模板与继承

- 类模板是类的泛型描述，正常类可以进行的操作类模板也可以进行，包括继承

- 举例：

继承为类模板、父类泛型：
扩展DArray功能，使之支持删除

```
template<class T>
class DArrayDel : public DArray<T> {
};
```

继承为正常类、父类不泛型：
扩展DArray<double>功能，使之
对double型有特殊操作

```
class DoubleDArray : public DArray<double> {
};
```

引入新泛型参数：
扩展DArray功能，使数组元素
可以是T1或T2的混合出现

```
template<class T1, class T2>
class MixedDArray : public DArray<T1> {
};
```

类模板 vs 接口类

类模板本质是什么？能用接口/虚函数替代吗？

- 类模板虽然仅依赖于一个抽象类型T，但内部实现依然需要T支持某些特殊操作
 - 例如，基于比较的排序算法需要支持<操作符；DArray需要=操作符
- 概念上十分类似于接口/虚函数：要求所有子类覆盖某个函数（这里是操作符）
- 那么是否可以用接口替代类模板？

同时也达到了某种意义上的多态，
虽然是编译时就完全决定的
- 答案一般是否定的：比如<，可以定义成virtual但是子类的覆盖是无意义的

```
class A {  
    virtual bool operator<(const A&);  
};  
class B : public A {  
    bool operator<(const B&);  
};
```

因为想让动态绑定生效那么子类函数形参表必须与父类
virtual operator<完全一致
而<操作符的形参表在子类必须是子类类型而不是父类类型

不依赖于操作符、使用接口较为合适的例子

- 例：Hashable接口提供哈希值；实现对任何Hashable都能用的CountMin

```
struct Hashable {  
    virtual int hash(int seed, int m) const = 0;  
};  
  
struct Point : public Hashable {  
    int x, y;  
    Point(int _x, int _y) : x(_x), y(_y) {}  
    int hash(int seed, int m) const {  
        return ((x << 20) + y) % m;  
    }  
};
```

Hashable接口
定义了hash纯虚函数

Point类继承Hashable类/接口，实现hash

```
int main() {  
    CountMin cm(20, 3);  
    cm.add(Point(0, 0));  
    cm.add(Point(1, 0));  
    cm.add(Point(2, 1));  
    cm.add(Point(2, 1));  
    cout << cm.query(Point(2, 1))  
        << endl;  
    return 0;  
}
```

不需要任何类型转换

```
class CountMin {  
    int** counter; int m, T;  
    int* seed;  
public:  
    CountMin(int size, int rep) : m(size), T(rep) {  
        std::minstd_rand gen;  
        std::uniform_int_distribution<> dist(0, m - 1);  
        counter = new int*[T];  
        seed = new int[T];  
        for (int i = 0; i < T; i++) {  
            seed[i] = dist(gen);  
            counter[i] = new int[m];  
            for (int j = 0; j < m; j++) counter[i][j] = 0;  
        }  
    }  
    void add(const Hashable& h) {  
        for (int i = 0; i < T; i++)  
            counter[i][h.hash(seed[i], m)]++;  
    }  
    int query(const Hashable& h) {  
        int ret = counter[0][h.hash(seed[0], m)];  
        for (int i = 1; i < T; i++)  
            ret = std::min(ret, counter[i][h.hash(seed[i], m)]);  
        return ret;  
    }  
};
```

整个类只需要操作一般的Hashable对象

总结：模板 vs 接口类

没有的话（通常就是数据结构/算法）模板更适合

- 概念上：是否可能抽象成is-a关系，有强is-a的话倾向于接口
 - 当然接口不可以处理基本类型、操作符重载
- 模板的缺点：
 - 模板必须暴露cpp/实现，每次扩展新的类需要整个项目重新编译
 - 模板对于下游需要实现什么操作符/函数是没有显式定义的
- 最后提一点本质不同：动态绑定利用了运行时信息，但模板完全利用编译时

C++ STL

(Standard Template Library)

C++标准库模板类

- 主要是一些常用的算法与数据结构的类模板/函数模板
 - Algorithms, 如排序, 查找等基本算法
 - Containers, 如 (动态) 数组、链表、字典 (set/map) 等数据结构
- 以及若干广泛存在于STL的特殊用法/接口
 - functor, iterator

具体用法请注意参考：


- <https://en.cppreference.com/>
- <https://cplusplus.com/reference/>

好处

研读一些内部代码实现也是深入理解C++的好方法

- 复用性好：接口设计出色，是样板级的C++程序，且任何平台都默认支持
- 整套类库都基于统一的设计规则，使用STL的代码比较一致、可读性佳
- 实现较为高效：精心选择实际高效的算法、优化实现
- C++标准库的一部分，资料比较完备、质量较高

坏处

- 学习曲线：有时只用其中一个小功能，却需要学习整套类库的一般概念才能用
- 代码长度：定义类型的代码可能比较长
auto关键字可以一定程度上解决
- 功能强大容易造成误用
 - 很复杂的功能集于一行代码，容易让人忽略背后的时间/空间代价等
- 对于特别需要效率的地方，STL未必足够高效

Functors

function objects

```
struct Point {  
    int x, y;  
    Point(int _x, int _y) : x(_x), y(_y) {}  
    int distsq(const Point &p) const {  
        Point delta = *this - p;  
        return delta.x * delta.x + delta.y * delta.y;  
    }  
    Point operator-(const Point& p) const {  
        return Point(x - p.x, y - p.y);  
    }  
};
```

- Functors (function objects): 用重载函数调用操作符的类作为函数来使用
- 例如, sort中用来指定比较器

大体作用是想将一个重要函数传入,
但用这种方法比只传函数要灵活

根据到o点距离排序,
有构造函数和成员变量

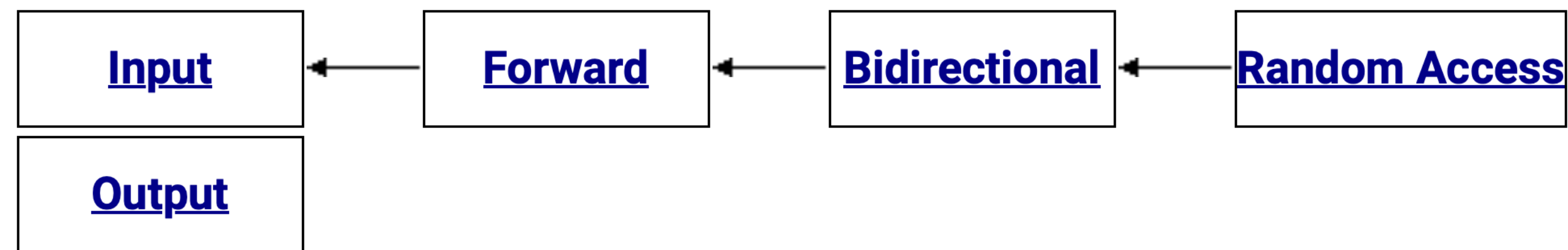
```
class PointComparatorDist {  
    Point o;  
public:  
    PointComparatorDist(const Point& p) : o(p) {}  
    bool operator()(const Point& p, const Point& q) {  
        return p.distsq(o) < q.distsq(o);  
    }  
};  
  
int main()  
{  
    Point x(5, 4), y(1, 1), u(0, 0);  
    Point arr[3] = {x, y, u};  
    std::sort(arr, arr + 3, PointComparatorDist(Point(0, 1)));  
    for (int i = 0; i < 3; i++) {  
        cout << arr[i].x << " " << arr[i].y << endl;  
    }  
    return 0;  
}
```

创建对象时传入o点坐标
(利用构造函数)

Iterator

迭代器

- 迭代器类似于指针，是指针的抽象化/推广
 - STL容器/数据结构对于所存储数据的访问都是通过迭代器进行的
- <https://cplusplus.com/reference/iterator/>
- 有5种迭代器，从左到右依次功能增强（即左边的功能右边的也有）



category				properties	valid expressions	
all categories				<u>copy-constructible</u> , <u>copy-assignable</u> and <u>destructible</u>	X b(a); b = a;	
				Can be incremented	++a a++	
<u>Random Access</u>	<u>Bidirectional</u>	<u>Forward</u>	<u>Input</u>	Supports equality/inequality comparisons	a == b a != b	
				Can be dereferenced as an rvalue	*a a->m	
			<u>Output</u>	Can be dereferenced as an lvalue (only for mutable iterator types)	*a = t *a++ = t	
			<u>default-constructible</u>	X a; X()		
			Multi-pass: neither dereferencing nor incrementing affects dereferenceability	{ b=a; *a++; *b; }		
			Can be decremented	--a a-- *a--		
	不支持forward iterator				Supports arithmetic operators + and -	a + n n + a a - n a - b
					Supports inequality comparisons (<, >, <= and >=) between iterators	a < b a > b a <= b a >= b
					Supports compound assignment operations += and -=	a += n a -= n
					Supports offset dereference operator ([])	a[n]

input 用于

区别: 比较、+/- n操作

input/output iterator主要用于单向数据流的只读/只写输入/输出

所有STL容器至少支持forward iterator

random access和bidirectional区别：支持<, >比较、+/- n操作

迭代器举例

random access

- 例如vector的begin和end分别返回头和(尾+1)的迭代器
- <https://cplusplus.com/reference/vector/vector/begin/>

An iterator to the beginning of the sequence container.

If the [vector](#) object is const-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are [random access iterator](#) types (pointing to an element and to a const element, respectively).

- 一般来说vector采用的迭代器都是random access的

主要因为vector后台是用数组维护的，天然支持random access

迭代器举例

bidirectional

- `set<int> X; X.find(val)` 返回一个迭代器，代表查找结果
- <https://cplusplus.com/reference/set/set/find/>

An iterator to the element, if **val** is found, or `set::end` otherwise.

Member types `iterator` and `const_iterator` are bidirectional iterator types pointing to elements.

- set的迭代器一般都是bidirectional的，不支持random access

后台是查找树，对random access支持不佳

迭代器举例

forward

- unordered_set的begin()函数返回一个迭代器，用于遍历
- https://cplusplus.com/reference/unordered_set/unordered_set/begin/

An iterator to the first element in the container (1) or the bucket (2).

All return types (iterator, const_iterator, local_iterator and const_local_iterator) are member types. In the [unordered_set](#) class template, these are [forward iterator](#) types.

- 为什么只是forward iterator? 有什么技术原因不能支持bidirectional吗?

无技术障碍，但是没必要支持另一个方向，因为哈希表本来就没方向可言，一般应用只是希望遍历，顺序不重要；额外增加一个反向迭代的功能可能会增加存储/运行代价

STL容器与迭代器

array和数组几乎无区别，只是包装成容器让API统一

STL容器	支持的迭代器
array/vector/dequeue	random access
list/map/multimap/set/multiset	bidirectional
forward_list/unordered_map/ unordered_multimap/unordered_set/ unordered_multiset	forward
stack/queue/priority-queue/bitset	不支持任何迭代器

注意阅读文档

- 要特别注意文档中对于复杂度、使用方式上的描述
- 例如，vector的erase函数<https://cplusplus.com/reference/vector/vector/erase/>

C++98 C++11 ?

```
iterator erase (const_iterator position); iterator erase (const_iterator first, const_iterator last);
```

Erase elements

函数重载和参数表

Removes from the [vector](#) either a single element (***position***) or a range of elements (`[first, last)`).

功能描述：

例如值得注意的是左闭右开

This effectively reduces the container [size](#) by the number of elements removed, which are destroyed.

对数据结构的影响

Because vectors use an array as their underlying storage, erasing elements in positions other than the [vector end](#) causes the container to relocate all the elements after the segment erased to their new positions. This is generally an inefficient operation compared to the one performed for the same operation by other kinds of sequence containers (such as [list](#) or [forward_list](#)).

这段针对实现细节的讨论
对于选择合适的数据结构
很有帮助

注意阅读文档

Parameters

position

Iterator pointing to a single element to be removed from the [vector](#).

Member types `iterator` and `const_iterator` are [random access iterator](#) types that point to elements.

first, last

Iterators specifying a range within the [vector](#) to be removed: `[first, last)`. i.e., the range includes all the elements between **first** and **last**, including the element pointed by **first** but not the one pointed by **last**.

Member types `iterator` and `const_iterator` are [random access iterator](#) types that point to elements.

对于两个重载版本的参数的描述
这很重要，仔细阅读防止用错

Return value

An iterator pointing to the new location of the element that followed the last element erased by the function call.

This is the [container end](#) if the operation erased the last element in the sequence.

注意这种对于特殊情况的讨论

Member type `iterator` is a [random access iterator](#) type that points to elements.

类似地，这里给出了返回值的意义

注意阅读文档

Complexity

尤其要注意这里对于复杂度的讨论：
erase在vector可以是O(n)复杂度的！

Linear on the number of elements erased (destructions) plus the number of elements after the last element deleted (moving).

Iterator validity

这段关于erase运行后对其他元素的影响的描述也很重要：例如在迭代过程中调用了erase，那么还未迭代到的元素还安全吗？是否需要重新迭代呢？

Iterators, pointers and references pointing to **position** (or **first**) and beyond are invalidated, with all iterators, pointers and references to elements before **position** (or **first**) are guaranteed to keep referring to the same elements they were referring to before the call.

```
int main() {  
    vector<int> vec;  
    for (int i = 0; i < 10; i++) vec.push_back(i);  
    for (auto it = vec.begin(); it != vec.end(); it++) {  
        if (*it % 2) vec.erase(it);  
    }  
    for (auto it = vec.begin(); it != vec.end(); it++) {  
        cout << *it << endl;  
    }  
    return 0;  
}
```

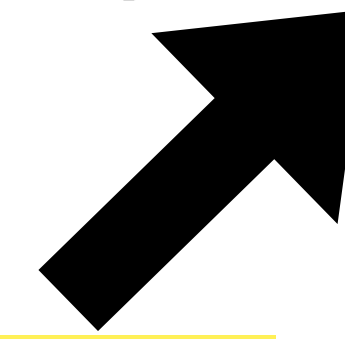
这段程序就会报错，因为文档说it后面的invalidated，it前面的才valid，而我们调用了后面的！

auto关键字

用auto缩短类型名

```
int main() {  
    set<int> s;  
    for (int i = 0; i < 10; i++)  
        s.insert(i);  
    for (set<int>::iterator it = s.begin(); it != s.end(); it++) {  
        cout << *it << endl;  
    }  
    return 0;  
}
```

An iterator to the *past-the-end* element in the container.



用iterator进行遍历

- set<int>::iterator很长，可以用auto关键字缩短

```
int main() {  
    set<int> s;  
    for (int i = 0; i < 10; i++)  
        s.insert(i);  
    for (auto it = s.begin(); it != s.end(); it++) {  
        cout << *it << endl;  
    }  
    return 0;  
}
```

auto关键字

- auto: 取代变量声明时的类型，从而不人为指定类型、让编译器自动进行推断

```
std::string func() {  
    return "abc";  
}  
  
int main() {  
    set<int> s;  
    for (int i = 0; i < 10; i++)  
        s.insert(i);  
    for (auto it = s.begin(); it != s.end(); it++) {  
        cout << *it << endl;  
    }  
    auto a = 5;  
    auto b = &a;  
    auto str = func();  
    return 0;  
}
```

```
struct Parent {  
    void f() {cout << "parent" << endl;}  
};  
  
struct Child : public Parent {  
    void f() {cout << "child" << endl;}  
};  
  
int main() {  
    auto p = new Child;  
    p->f();  
    return 0;  
}
```

输出child

auto&

- auto推断出来的类型一定不是引用，即使赋的值是引用的，例如

```
int& func(int &x) {  
    return x;  
}  
  
int main() {  
    int a = 5;  
    auto b = func(a);  
    int& c = func(a);  
    b = 10;  
    c = 20;  
    cout << a << " " << b << " " << c << endl;  
    return 0;  
}
```

输出20 10 20

- 如果要引用，则可以用auto&，例如

```
int& func(int &x) {  
    return x;  
}  
  
int main() {  
    int a = 5;  
    auto& b = func(a);  
    b = 10;  
    cout << a << " " << b << " " << endl;  
    return 0;  
}
```

输出10 10

auto应该什么时候用?

主要考虑可读性!

- 变量类型不重要/用户不关心, 且这个变量的意义即使不看变量类型也非常明确

- 简化STL中比较长的类型名是典型的好用法

仍建议慎用, 因为看代码的人只看声明
可能依然需要猜/检查对应类型

- 类型重要, 但是从变量名/所赋值的函数名称可以容易推断

```
int main() {  
    auto str = xxx;  
    auto a = toInt();  
    return 0;  
}
```

str这个名字比较明显

a的变量名虽然不明显, 但赋值toInt比较明确

- 不要用: 变量名和调用都不太能体现类型

```
struct X {  
};  
  
X func() {  
    return X();  
}
```

```
int main() {  
    auto x = func();  
    return 0;  
}
```

无法看出什么类型, 可读性极低, 读者
需要特别检查并记忆x的类型