中國人民大學
RENMIN UNIVERSITY OF CHINA

7 模板

授课教师：孙亚辉

模板基础

- 模板是一个将数据类型参数化的工具，它把“一般性的算法”和其“对数据类型的实现”区分开。
- 采用模板方式定义函数或类时，不确定某些函数参数或数据成员的类型，而将它们的数据类型作为模板的参数。
- 在使用模板时根据实参的数据类型确定模板参数的数据类型。
- 模板提高了软件的重用性。当函数参数或数据成员可以是多种类型而函数或类所实现的功能又相同时，使用C++模板在很大程度上简化了编程。

模板的用途

如果没有模板的话，针对每个所需相同行为的不同类型，重复实现相似的代码：

```
1.  int max(int a, int b) { return (a > b)?a:b;}
2.  long max(long a, long b) { return (a > b)?a:b;}
3.  double max(double a, double b) { return (a > b)?a:b;}
4.  char max(string a, string b) { return (a > b)?a:b;}
```

- 模板克服了上述缺点。
- 模板的应用非常广泛，C++标准库中，几乎所有的代码都是模板代码。

模板的分类

- 函数模板：可以产生一组功能完全相同，只是参数、返回值类型不同的函数。
- 类模板：可以为类定义一种模式，使得类中的某些数据成员、某些成员函数的参数、返回值或局部变量能取任意类型。

本章内容

1. 函数模板
2. 类模板
3. 非类型模板参数
4. 模板与多态

1 函数模板

- 定义

template <类型参数列表>
返回值类型 函数名（模板函数形参列表）{函数定义体}

- 例子

typename关键字可用class关键字替代

```
1. template <typename T>                    //含单个类型参数的函数模板
2. T max (T a, T b) { return  a < b ? b : a; }

3. template <typename T1, typename T2>  //含多个类型参数的函数模板
4. T1 max(T1 a, T2 b) { return a < b ? (T1)b : a; }
```

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

template <class T> //含单个类型参数的函数模板
T max1(T a, T b)
{
    return a < b ? b : a;
}

int main()
{
    int a = 1, b = 0;
    string c = "a", d = "b";
    cout << max1(a, b) << endl;
    cout << max1(c, d) << endl;

    return 0;
}
```

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

template <typename T> //含单个类型参数的函数模板
T max1(T a, T b)
{
    return a < b ? b : a;
}

int main()
{
    int a = 1, b = 0;
    string c = "a", d = "b";
    cout << max1(a, b) << endl;
    cout << max1(c, d) << endl;

    return 0;
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
1
b
```

函数模板的实例化

- 在代码中包含函数模板本身并不会生成函数定义
  - 它只是一个用于生成函数定义的方案
- 编译器使用模板为特定类型生成函数定义时，得到的是模板实例（instantiation）
- 模板并非函数定义，但使用int的max函数实例是函数定义

函数模板的实例化

- 隐式实例化：发生函数调用时，编译器推断模版实参，实现实例化
- 显式实例化（*explicit instantiation*）：强制某些函数实例化（无论是否有对其进行调用）
  - **显式实例化以template打头，在函数声明的函数名后加<类型名>**
  - 下例，通过查看编译器生成的汇编代码，可以看到生成了char类型对应的函数实例_Z3maxIcET_S0_S0_（在main函数中并没以char类型调用max）

```
1. template <typename T>
2. T max (T a, T b) { return  a < b ? b : a; }

3. max(7, 42);                                  //隐式实例化 int max(int, int)
4. template char max<char>(char, char);         //显式实例化 char max(char, char)
```

函数模板的调用

- 隐式指定模板参数类型：在发生函数模板的调用时，不显式给出模板参数类型，而是推演参数类型。
- 显式指定模板参数类型：在发生函数模板的调用时，显式给出模板参数类型，而不需要推演参数类型。
  - **显式指定模板参数类型是在函数名后加<类型名>**

```
1. template <typename T>
2. T max (T a, T b) { return  a < b ? b : a; }

3. max(7, 42);                                  //隐式指定模板参数int
4. max(4, 4.2);                                 //编译错误，没有匹配的函数模板
5. max(static_cast<double>(4), 4.2);            //修复方式1：强制类型转化
6. max<double>(4, 4.2)                          //修复方式2：显式指定模板参数double
```

函数模板的特化

- 在实例化函数模板时，对特定类型的模板参数进行特殊处理（实例化一个特殊的实例版本）。当以特化定义指定的参数使用函数模板时，将调用特化版本（函数内容可与通用版本不一样）。
  - “特化”也被称为“显式具体化”（explicit specialization）
  - 具体化的原型与定义应以template<>打头（刚才讲的显式实例化是以template打头）

```
1. template <typename T>
2. T max (T a, T b) { return  a < b ? b : a; }

3. template<> double max<double>(double a, double b) {
4.   return a;
5. }

6. max(7, 42);                          //使用通用函数模板生成函数实例
7. max(1.1, 4.2);                       //使用特化函数模板生成函数实例
```

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

template <typename T> //含单个类型参数的函数模板
T max1(T a, T b)
{
    return a < b ? b : a;
}

template <> double max1(double a, double b)
{
    return 1;
}

int main()
{
    cout << max1(7, 42) << endl; //隐式指定模板参数int
    cout << max1<double>(4, 4.2) << endl; // 显式指定模板参数 + 函数模板的特化

    return 0;
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
42
1
```

- 特化的原型和定义以template<>打头，而显式实例化是以template打头
  - 显式实例化只有声明，不能重新定义实现
  - **同一种类型的显式实例与特化同时存在时将编译出错**

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

template <typename T> //含单个类型参数的函数模板
T max1(T a, T b)
{
    return a < b ? b : a;
}
template double max1(double a, double b); // 显式实例化
template <> double max1(double a, double b) // 函数模板的特化
{
    return 1;
}


int main()
{
    return 0;
}
```

显式实例化后又进行特化，编译器报错

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp:11:43: error: specialization of ‘T max1(T, T) [with T = double]’
 after instantiation
   11 | template <> double max1(double a, double b) // 函数模板的特化
      |                                           ^
```

C++关于函数模板的处理

- 对于给定的函数名，可以有非模板函数、模板函数和特化模板函数以及他们的重载版本
- **非模板函数优先于特化的常规模板，特化优先于常规模板**

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

template <typename T> //含单个类型参数的函数模板
T max1(T a, T b)
{
    return a < b ? b : a;
}

template <> double max1(double a, double b)
{
    return 1;
}

double max1(double a, double b)
{
    return 2;
}

int main()
{
    cout << max1(7, 42) << endl; //隐式指定模板参数int
    cout << max1(4.1, 4.2) << endl;

    return 0;
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
42
2
```

本章内容

1. 函数模板
2. 类模板
3. 非类型模板参数
4. 模板与多态

2 类模板

- 定义

```
template ‹ 类型参数列表 ›
class 类名 {类声明体}
```

- 例子

```
1.  template<typename T>
2.  class Stack {
3.    public:
4.      void push(T value);
5.      T pop();

6.    private:
7.      T elements[100];
8.      int size;
9.  };
```

```
10. template<typename T>
11. void Stack<T>::push(T value)
12. {
13.   elements[size++] = value;
14. }
15. template<typename T>
16. T Stack<T>::pop()
17. {
18.   return elements[--size];
19. }
```

类模板中的成员函数定义

- 在类中内联定义方法

```
1.  template<typename T>
2.  class Stack {
3.    public:
4.      void push(T value) {
5.          . . . .
6.      }
7.      T pop() {
8.          . . . .
9.      }

10.   private:
11.     T elements[100];
12.     int size;
13. };
```

与普通的类成员函数内联定义一致

类模板中的成员函数定义

- 在类声明之外定义方法

```
10. template<typename T>
11. void Stack<T>::push(T value)
12. {
13.   elements[size++] = value;
14. }
15. template<typename T>
16. T Stack<T>::pop()
17. {
18.   return elements[--size];
19. }
```

- 不能省略模板前缀
- 模板参数类型名称可以与模板内中声明的类型名称不一致
  - 比如15-16行的T可以是T2
- 类名后加<模板参数>再加::
- 类声明中有几个模板参数，此处“<>”中也应该有几个

类模板中的成员函数定义

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

template <typename T>
class Stack
{
public:
    void push(T value);
    virtual T pop()
    {
        return elements[--size];
    }
    T get_element(int x)
    {
        return elements[x];
    }

private:
    T elements[100];
    int size = 0;
};
```

```cpp
template <typename T2> // 模板参数类型可以与模板内中声明的类型不一致
void Stack<T2>::push(T2 value)
{
    elements[size++] = value;
}

int main()
{
    Stack<int> intStack;
    Stack<string> dblStack;

    intStack.push(0);
    intStack.push(1);
    cout << intStack.get_element(1) << endl;

    dblStack.push("a");
    dblStack.push("b");
    cout << dblStack.get_element(1) << endl;

    return 0;
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
1
b
```

- 可以将模板成员函数定义在独立的h文件中

10_0.h

Miscellaneous Files

```
1   template <typename T>
2   class Stack
3   {
4   public:
5       void push(T value);
6       T pop();
7       T get_element(int x)
8       {
9           return elements[x];
10      }
11
12  private:
13      T elements[100];
14      int size = 0;
15  };
```

10_1.h

Miscellaneous Files

```
1   #include "10_0.h"
2
3   template <typename T>
4   void Stack<T>::push(T value)
5   {
6       elements[size++] = value;
7   }
8
9   template <typename T>
10  T Stack<T>::pop()
11  {
12      return elements[--size];
13  }
```

test.cpp

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   #include "10_1.h"
6
7   int main()
8   {
9       Stack<int> a;
10      a.push(1);
11      a.push(2);
12      cout << a.get_element(1) << endl;
13
14      return 0;
15  }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
2
```

- 不能将模板成员函数放在独立的没有模板实例化请求的cpp文件中（存在模板实例化请求时，编译器才实现模板函数）
  - **模板必须与特定的模板实例化请求一起使用**

m.h

```
template <typename T>
class Stack
{
public:
    void push(T value);
    T pop();
    T get_element(int x)
    {
        return elements[x];
    }

private:
    T elements[100];
    int size;
};
```

m.cpp

```
#include "m.h"
template<typename T>
void Stack<T>::push(T value)
{
 elements[size++] = value;
}
template<typename T>
T Stack<T>::pop()
{
 return elements[--size];
}
```

test.cpp

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

#include "m.h"

int main()
{
    Stack<int> a;
    a.push(1);
    a.push(2);
    cout << a.get_element(1) << endl;

    return 0;
}
```

```
yahui@Yahui:/media/sf_VM$ g++ m.cpp test.cpp
/usr/bin/ld: /tmp/ccjd859W.o: in function `main':
test.cpp:(.text+0x2e): undefined reference to `Stack<int>::push(int)'
/usr/bin/ld: test.cpp:(.text+0x42): undefined reference to `Stack<int>:
:push(int)'
collect2: error: ld returned 1 exit status
```

类模板的实例化

- 隐式实例化：在使用模板类生成对象时，将类模板实例化
- 显式实例化：不管是否生成一个模板类的对象，都可以直接通过显式实例化声明将类模板实例化

```
1. template<typename T>
2. class Stack { … }

3. Stack<int> intStack;                    //隐式实例化
4. template class Stack<char>;             //显式实例化
```

类模板的特化

- 通过特化类模板，可以优化基于某种特定类型的实现。特化类模板时要特化其中的所有成员函数。

```
1.  template<> class Stack<std::string> {
2.    public:
3.      void push(std::string value);
4.      std::string  pop();
5.  private:
6.      std::vector<std::string> elements;
7.  };
8.  void Stack<std::string>::push(std::string val) {
9.    elements.push_back(val);
10. }
11. std::string Stack<std::string>::pop() {
12.   …
13. }
```

局部特化

- 类模板可以被局部特化（*partial specialization*），可以在特定的环境下指定类模板的特定实现

```
1.  //普通类模板
2.  template<typename T1, typename T2>
3.  class MyClass { … };

4.  //局部特化1：两个模板参数具有相同的类型
5.  template<typename T> class MyClass<T, T> {…};
6.  //局部特化2：第2个模板参数的类型是int
7.  template<typename T> class MyClass<T, int> {…};
8.  //局部特化3：两个模板都是指针类型
9.  template<typename T1, typename T2>
10. class MyClass<T1*, T2*> {…}

11. MyClass<int, float> mif;          //使用普通类模板版本
12. MyClass<float, float> mff;        //使用局部特化1版本
13. MyClass<float, int> mfi;          //使用局部特化2版本
14. MyClass<int*, float*> mp;         //使用局部特化3版本
```

将第二个模板参数特化为int，所以template<>的<>中只需声明未被特化的类型参数

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

//普通类模板
template <typename T1, typename T2>
class MyClass
{
public:
    void print() { cout << "MyClass" << endl; }
};
//局部特化1：两个模板参数具有相同的类型
template <typename T>
class MyClass<T, T>
{
public:
    void print() { cout << "MyClass<T, T>" << endl; }
};
//局部特化2：第2个模板参数的类型是int
template <typename T>
class MyClass<T, int>
{
public:
    void print() { cout << "MyClass<T, int>" << endl; }
};
//局部特化3：两个模板都是指针类型
template <typename T1, typename T2>
class MyClass<T1 *, T2 *>
{
public:
    void print() { cout << "MyClass<T1 *, T2 *>" << endl; }
};
```

```cpp
int main()
{
    MyClass<int, float> mif; //使用普通类模板版本
    mif.print();
    MyClass<float, float> mff; //使用局部特化1版本
    mff.print();
    MyClass<float, int> mfi; //使用局部特化2版本
    mfi.print();
    MyClass<int *, float *> mp; //使用局部特化3版本
    mp.print();

    return 0;
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
MyClass
MyClass<T, T>
MyClass<T, int>
MyClass<T1 *, T2 *>
```

缺省（默认）类模板参数

- 可以为类模板参数定义缺省值，在实例化类模板时，如果没有给带缺省值的模板参数指定值，则该模板参数使用缺省值。

```
1.  template<typename T, typename CONT = std::vector<T>>
2.  class Stack {
3.    public:
4.      void push(T value);
5.      T pop();

6.    private:
7.      CONT elements;
8.      int size;
9.  };

10. Stack<int> intStack;                                  //默认使用std::vector
11. Stack<double, std::deque<double>> dblStack;    //指定使用std::deque
```

成员模板

- 前述的示例中，类模板的成员函数只用到了类模板所声明的模板参数类型。事实上，可以在类模板的成员（如成员函数、嵌套类）中引入新的模板参数（模板的嵌套）。

```
1.  template<typename T>
2.  class Stack {
3.    public:
4.      void push(T value);
5.      T pop();
6.      template<typename X, typename Y>
7.      void print(X x, Y y) {
8.          cout << x << " " << y << endl;
9.      }

10.   private:
11.     T elements[100];
12.     int size;
13. };
```

```
1.  int main() {
2.      Stack<int> stack;
3.      stack.print(1.23, 'A');
4.  }
```

输出结果：
1.23 A

- 如果print方法在类声明之外进行定义，则应使用如下形式

```
1.  template<typename T>
2.    template<typename X, typename Y>
3.  void Stack<T>::print(X x, Y y) {
4.      cout << x << “ “ << y << endl;
5.  }
```

- 而不能使用

```
1.  template<typename T, typename X, typename Y>
2.  void Stack<T>::print(X x, Y y) {
3.      cout << x << “ “ << y << endl;
4.  }
```

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

template <typename T, typename CONT = vector<T>> // 为类模板参数定义缺省值
class Stack
{
public:
  void push(T value);
  virtual T pop(); // 非模板的成员函数可以是虚函数

  template <typename X, typename Y> //可以在类模板的成员（如成员函数、 嵌套类） 中引入新的模板参数
  void print(X x, Y y); // print 不能为virtual:  成员函数模板不能声明为虚函数

private:
  T elements[100];
  int size = 0;
};

template <typename T, typename CONT>
void Stack<T, CONT>::push(T value)
{
  elements[size++] = value;
}

template <typename T, typename CONT>
T Stack<T, CONT>::pop()
{
  return elements[--size];
}
```

```cpp
template <typename T, typename CONT>
template <typename X, typename Y> // 不能将该行与上一行合并
void Stack<T, CONT>::print(X x, Y y)
{
  cout << x << " " << y << endl;
}

int main()
{
  Stack<int> intStack; //默认使用std::vector

  Stack<double, deque<double>> dblStack; //指定使用std::deque

  return 0;
}
```

- 注意，类模板的普通成员函数可以声明为虚函数，但**成员函数模板不能声明为虚函数。**（编译器在编译一个类的时候，需要确定这个类的虚函数表的大小，即类内部虚函数的数量；如果允许一个成员模板函数为虚函数的话，因为我们可以为该成员模板函数实例化出很多不同的版本，也就是可以实例化出很多不同版本的虚函数，此时编译器难以确定类内部虚函数的数量，也难以确定虚函数表的大小）

```
1.  template <typename T>
2.  class D {
3.  public:
4.      virtual ~D();
5.      virtual void copy(T const & t);
6.  };
```

```
1.  template <typename T>
2.  class D {
3.  public:
4.      virtual ~D();
5.      template <typename T2>
6.      virtual void copy(T2 const & t);
7.  };
```

智能指针类模板shared_ptr

如下代码

```
void leaky() { SomeClass * sc = new SomeClass;  sc->doSomething(); }
```

存在明显的内存泄露：在函数内部通过new操作分配的内存，既没有作为返回值供调用者使用或赋予某个全局变量供他处使用，也没有在生命周期结束时销毁。C++提供了多种智能指针解决上述代码中存在的问题，开发者无需显式地在leaky中销毁sc，而是通过智能指针自动完成该操作。智能指针类模板shared_ptr允许多个智能指针对象共享同一个对象指针（如sc），而且所有智能指针对象生命周期结束的时候，能够自动销毁对象指针。如：

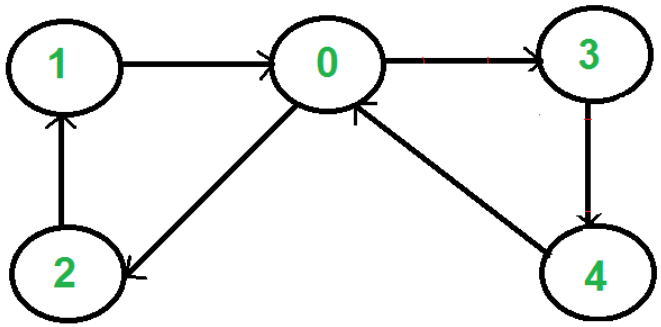
```
shared_ptr<SomeClass> sp1(sc);  shared_ptr<SomeClass> sp2 = sp1;
sp1->doSomething();   (*sp2).doSomething();
```

上述代码中的sc只会被销毁一次。C++中的shared_ptr通过一个引用计数来判别所持有的对象指针是否应该销毁：对象指针被一个shared_ptr引用，引用计数为1；被两个shared_ptr引用，引用计数为2；若引用计数为0，则对象指针可被释放掉。

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

template <typename T>
class MyShared_ptr
{
private:
  int *m_count;
  T *m_ptr;

public:
  MyShared_ptr() : m_ptr(nullptr), m_count(new int) {}
  MyShared_ptr(T *ptr) : m_ptr(ptr), m_count(new int)
  {
    cout << "new space: " << ptr << endl;
    *m_count = 1;
  }
  MyShared_ptr(const MyShared_ptr &ptr) //拷贝构造函数
  {
    m_count = ptr.m_count;
    m_ptr = ptr.m_ptr;
    ++(*m_count);
  }
  ~MyShared_ptr()
  {
    --(*m_count);
    if (*m_count == 0)
    {
      cout << "release space: " << m_ptr << endl;
      delete m_ptr;
      delete m_count;
      m_ptr = nullptr;
      m_count = nullptr;
    }
  }
  T &operator*() { return *m_ptr; }
  T *operator->() { return m_ptr; }
};

class A
{
public:
  int x;
  void doSomething() {}
  ~A() { cout << "~A()" << endl; }
};

void leaky()
{
  A *sc = new A;
  sc->x = 1;
}

void not_leaky()
{
  MyShared_ptr<A> sc = new A;
  sc->x = 1;
  MyShared_ptr<A> sp1(sc);
  cout << "sp1->x: " << sp1->x << endl;
  MyShared_ptr<A> sp2 = sp1;
  cout << "(*sp2).x: " << (*sp2).x << endl;
  MyShared_ptr<string> sp3(new string("hello world"));
  cout << "(*sp3).c_str(): " << (*sp3).c_str() << endl;
}

int main()
{
  leaky();
  cout << "--------" << endl;
  not_leaky();
  return 0;
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
--------
new space: 0x55c28a51e2e0
sp1->x: 1
(*sp2).x: 1
new space: 0x55c28a51e320
(*sp3).c_str(): hello world
release space: 0x55c28a51e320
release space: 0x55c28a51e2e0
~A()
```

有向图directed graph



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

/*a directed graph with edge weights*/
template <typename weight_type> // weight_type may be int, float, double...
class dgraph_v_of_v {
public:
    /*
    INs.size() == OUTs.size() is the number of vertices, IDs from 0 to n-1
    INs: adj_lists: set of adjacency vertices that can reach v (with arc/edge weights)
    OUTs: adj_lists: set of adjacency vertices that can be reached by v (with arc/edge weights)
    if v1 is in INs of v2, then v2 must be in OUTs of v1, and both records should be associated with the same weight.
    */
    std::vector<std::vector<std::pair<int, weight_type>>> INs, OUTs;

    /*constructors*/
    dgraph_v_of_v() {}
    dgraph_v_of_v(int n) {
        INs.resize(n); // initialize n vertices
        OUTs.resize(n);
    }

    /*class member functions*/
    inline void add_edge(int, int, weight_type); // this function can change edge weights
    inline void remove_edge(int, int);
    inline weight_type edge_weight(int, int);
    inline bool contain_edge(int, int); // whether there is an edge
    inline long long int edge_number(); // the total number of edges
    inline void clear();
    inline int degree(int);
};
```

```cpp
/*binary operations*/
template <typename T>
bool graph_hash_of_mixed_weighted_binary_operations_search(std::vector<std::pair<int, T>>& input_vector, int key) {

    /*return true if key is in vector; time complexity O(log n)*/
    int left = 0, right = input_vector.size() - 1;
    while (left <= right) {
        int mid = left + ((right - left) / 2); // mid is between left and right (may be equal);
        if (input_vector[mid].first == key) {
            return true;
        }
        else if (input_vector[mid].first > key) {
            right = mid - 1;
        }
        else {
            left = mid + 1;
        }
    }
    return false;
}
template <typename T>
T graph_hash_of_mixed_weighted_binary_operations_search_weight(std::vector<std::pair<int, T>>& input_vector, int key) {

    /*return std::numeric_limits<T>::max() if key is not in vector; time complexity O(log n)*/
    int left = 0, right = input_vector.size() - 1;
    while (left <= right) {
        int mid = left + ((right - left) / 2); // mid is between left and right (may be equal);
        if (input_vector[mid].first == key) {
            return input_vector[mid].second;
        }
        else if (input_vector[mid].first > key) {
            right = mid - 1;
        }
        else {
            left = mid + 1;
        }
    }
    return std::numeric_limits<T>::max();
}
```

```cpp
template <typename T>
void graph_hash_of_mixed_weighted_binary_operations_erase(std::vector<std::pair<int, T>>& input_vector, int key) {

    /*erase key from vector; time complexity O(log n + size()-position ), which is O(n) in the worst case, as
    the time complexity of erasing an element from a vector is the number of elements behind this element*/
    if (input_vector.size() > 0) {
        int left = 0, right = input_vector.size() - 1;
        while (left <= right) {
            int mid = left + ((right - left) / 2);
            if (input_vector[mid].first == key) {
                input_vector.erase(input_vector.begin() + mid);
                break;
            }
            else if (input_vector[mid].first > key) {
                right = mid - 1;
            }
            else {
                left = mid + 1;
            }
        }
    }
}
```

```cpp
template <typename T>
int graph_hash_of_mixed_weighted_binary_operations_insert(std::vector<std::pair<int, T>>& input_vector, int key, T load) {

    /*return the inserted position;
    insert <key, load> into vector, if key is already inside, then load is updated; time complexity O(log n + size()-positi
    the time complexity of inserting an element into a vector is the number of elements behind this element*/

    int left = 0, right = input_vector.size() - 1;
    while (left <= right) { // it will be skept when input_vector.size() == 0
        int mid = left + ((right - left) / 2); // mid is between left and right (may be equal);
        if (input_vector[mid].first == key) {
            input_vector[mid].second = load;
            return mid;
        }
        else if (input_vector[mid].first > key) {
            right = mid - 1; // the elements after right are always either empty, or have larger keys than input key
        }
        else {
            left = mid + 1; // the elements before left are always either empty, or have smaller keys than input key
        }
    }

    /*the following code is used when key is not in vector, i.e., left > right, specifically, left = right + 1;
    the elements before left are always either empty, or have smaller keys than input key;
    the elements after right are always either empty, or have larger keys than input key;
    so, the input key should be insert between right and left at this moment*/
    input_vector.insert(input_vector.begin() + left, { key,load });
    return left;
}
```

```cpp
/*class member functions*/
template <typename weight_type>
void dgraph_v_of_v<weight_type>::add_edge(int v1, int v2, weight_type weight) {

    /*
    edge direction: v1 to v2;
    this function adds v1 into INs of v2, and also adds v2 into OUTs of v1;
    this function can change edge weights;
    */
    graph_hash_of_mixed_weighted_binary_operations_insert(OUTs[v1], v2, weight);
    graph_hash_of_mixed_weighted_binary_operations_insert(INs[v2], v1, weight);
}
template <typename weight_type>
void dgraph_v_of_v<weight_type>::remove_edge(int v1, int v2) {

    /*edge direction: v1 to v2*/
    graph_hash_of_mixed_weighted_binary_operations_erase(OUTs[v1], v2);
    graph_hash_of_mixed_weighted_binary_operations_erase(INs[v2], v1);
}
template <typename weight_type>
weight_type dgraph_v_of_v<weight_type>::edge_weight(int v1, int v2) {

    /*edge direction: v1 to v2*/
    return graph_hash_of_mixed_weighted_binary_operations_search_weight(OUTs[v1], v2);
}
template <typename weight_type>
bool dgraph_v_of_v<weight_type>::contain_edge(int v1, int v2) {

    /*this function checks two conditions: v1 is in INs of v2; v2 is in OUTs of v1*/
    return graph_hash_of_mixed_weighted_binary_operations_search(OUTs[v1], v2);
}
template <typename weight_type>
long long int dgraph_v_of_v<weight_type>::edge_number() {

    /*only check INs*/
    long long int num = 0;
    for (int i = INs.size() - 1; i >= 0; i--) {
        num += INs[i].size();
    }
    return num;
}
```

```cpp
template <typename weight_type>
void dgraph_v_of_v<weight_type>::clear() {
    std::vector<std::vector<std::pair<int, weight_type>>>().swap(INs);
    std::vector<std::vector<std::pair<int, weight_type>>>().swap(OUTs);
}
template <typename weight_type>
int dgraph_v_of_v<weight_type>::degree(int v) {
    return OUTs[v].size() + INs[v].size();
};

int main() {
    dgraph_v_of_v<double> g(10); // initialize a graph with 10 vertices
    g.add_edge(1, 5, 1.02);
    g.add_edge(5, 1, 1.42);
    g.add_edge(5, 2, 122);
    g.remove_edge(5, 1);
    std::cout << g.edge_weight(1, 5) << std::endl;
    std::cout << g.contain_edge(1, 5) << std::endl;
    std::cout << g.contain_edge(5, 1) << std::endl;
    std::cout << g.edge_number() << std::endl;
    std::cout << g.INs.size() << std::endl;
}
```

本章内容

1. 函数模板
2. 类模板
3. 非类型模板参数
4. 模板与多态

3 非类型模板参数

- 模板参数并不局限于类型，普通值也可以作为模板参数
- 当要使用基于值的模板时，必须显式地指定这些值，才能对模板进行实例化
- 典型应用场景
  - 用来定义数组长度或者容器大小
  - 定义参与函数运算的常量
- 限制
  - 非类型模板参数可以是常整数
  - **浮点数和类对象不允许作为非类型模板参数**

非类型参数是在编译期或链接期可以确定的常值

- 以下函数模板定义了一组用于增加特定值的函数

```
1.  template <typename T, int VAL>
2.  T addValue (T const& x) {
3.    return x + VAL;
4.  }
```

- 以下类模板指定了成员数组的大小

```
1. template <typename T, int MAXSIZE>
2. class Stack {
3.   public:
4.     void push(T value);
5.     T pop();

6.   private:
7.     T elements[MAXSIZE];
8.     int size;
9. };

10.Stack<int, 20> int20Stack;
```

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

template <typename T, int MAXSIZE> //普通值也可以作为模板参数
class Stack
{
public:
  void push(T value);
  T pop();

private:
  T elements[MAXSIZE];
  int size = 0;
};

template <typename T, int MAXSIZE> // 类声明中有几个模板参数， 此 处“<>” 中也应该有几个
void Stack<T, MAXSIZE>::push(T value)
{
  elements[size++] = value;
}

template <typename T, int MAXSIZE>
T Stack<T, MAXSIZE>::pop()
{
  return elements[--size];
}

int main()
{
  Stack<int, 100> intStack; // 当要使用基于值的模板时， 必须显式地指定这些值， 才能对模板进行实例化
  // 非类型参数是在编译期或链接期可以确定的常值

  intStack.push(1);
  intStack.push(2);
  cout << intStack.pop() << endl;

  return 0;
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
2
```

本章内容

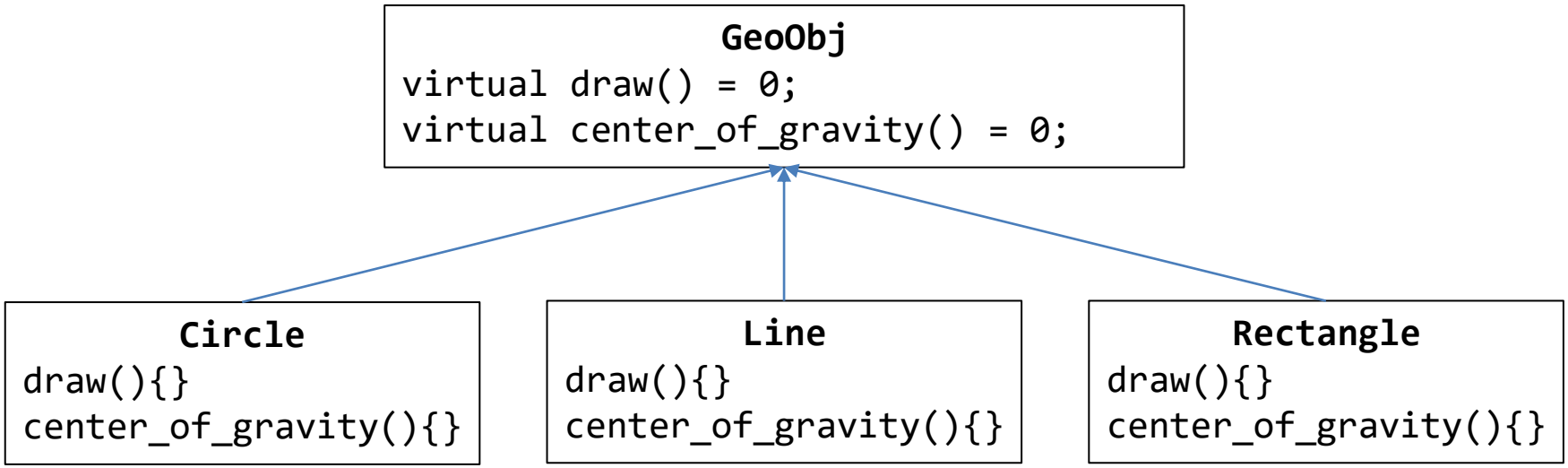
1. 函数模板
2. 类模板
3. 非类型模板参数
4. 模板与多态

4 模板与多态

- 动态多态（运行时多态）：通过继承和虚函数相结合的方式实现。对于几个相关对象的类型，确定它们之间的一个共同功能集，然后在基类中，把这些共同的功能声明为虚函数接口。

- 静态多态（编译期多态）：通过模板的方式实现。不依赖于基类中包含公共行为的因素；但依然存在一种隐式的公共性：相同的操作用的函数名字相同。

动态多态




```
GeoObj *obj1 = new Circle();
GeoObj *obj2 = new Line();
obj1->draw();  //Circle::draw
obj2->draw();   //Line::draw
```

动态多态本质上就是面向对象设计中的继承、多态的概念

静态多态

Circle
draw() center_of_gravity()

Line
draw() center_of_gravity()

Rectangle
draw() center_of_gravity()

```
Circle circle;
Line line;

template<typename T>
void draw(T x) { …… }

draw(circle);
draw(line);
```

静态多态本质上就是模板的实例化

对比动态多态和静态多态

- 通过继承和虚函数实现的多态是绑定的和动态的
  - 绑定的：对于参与多态行为的类型，它们的接口是在公共基类的设计中就预先确定的
  - 动态的：接口的绑定是在运行时动态完成的

- 通过模板实现的多态是非绑定的和静态的
  - 非绑定的：对于参与多态行为的类型，它们的接口是没有预先确定的
  - 静态的：接口的绑定时在编译阶段完成的

对比动态多态和静态多态

- 动态多态的优点
  - 能够优雅地处理同一继承体系下异类对象集合（一个基类指针可以指向不同的派生类对象）
  - 面向对象设计，是对客观世界的直觉认识
  - 可以对代码进行完全编译，不需要发布实现源码

- 静态多态的优点
  - 可以容易地实现内建类型的集合，不需要通过公共基类来表达接口的共同性
  - 所生成的代码效率通常都比较高（编译期完成，编译器可以进行优化）
  - 通过模板编程为C++带来了泛型设计的概念，比如强大的STL库（泛型程序设计，generic programming，是算法在实现时不指定具体要操作的数据类型的程序设计方法。所谓“泛型”，指的是算法只要实现一遍，就能适用于多种数据类型。泛型程序设计方法的优势在于能够减少重复代码的编写。）

作业

- YOJ-399 三维数组类模板
- YOJ-622 C++模板（1）
- YOJ-623 C++模板（2）
- YOJ-624 C++模板（3）
- YOJ-625 C++模板（4）