

1 模板概述

2 函数模板

2.1 什么是函数模板

2.2 函数模板练习

3 函数模板和普通函数的区别

4 函数模板和普通函数在一起的调用规则

5 函数模板剖析

5.1 编译过程

5.2 函数模板的本质

6 模板具体化

7 类模板

7.1 类模板的实现

7.2 类模板作为函数参数

7.3 类模板遇到继承

7.4 类模板的成员函数类内实现

7.5 类模板的成员函数类外实现

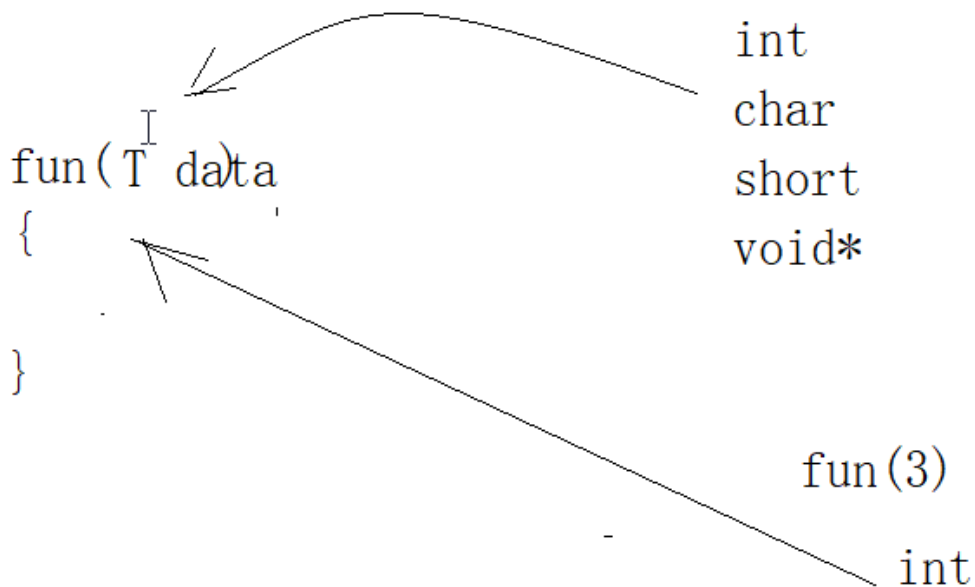
7.6 类模板成员函数的创建时机

7.7 类模板的分文件问题

7.8 类模板遇到友元

1 模板概述

函数模板: 形参的类型不具体指定,用通用类型代替,在调用时,编译器会根据实参的类型推导出形参的类型(类型参数化)



c++模板：
最重要的技术：类型参数化

2函数模板

2.1 什么是函数模板

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 void swap(int &x, int &y)
8 {
9     int temp = x;
10    x = y;
11    y = temp;
12 }
13
14 void swap(char &x, char &y)
15 {
16     char temp = x;
```

```
17  x = y;
18  y = temp;
19  }
20  void test01()
21  {
22      int a = 1;
23      int b = 2;
24      swap(a,b);
25      cout << a << " " << b << endl;
26  }
27  void test02()
28  {
29      char a = 1;
30      char b = 2;
31      swap(a, b);
32      cout << a << " " << b << endl;
33  }
34  //函数模板来实现
35  template <class T> //定义一个模板 模板的通用类型为T
36  //紧跟函数的定义
37  void swap_temp(T &a, T &b)
38  {
39      T temp = a;
40      a = b;
41      b = temp;
42  }
43  void test03()
44  {
45      char a = 1;
46      char b = 2;
47      int c = 3;
48      int d = 4;
49      swap_temp(a, b); //自动推导
50      //swap_temp(a,c); 自动类型推导的结果不一致
51      swap_temp(c, d);
52  }
53
54  int main()
55  {
56      test01();
```

```
57     return 0;
58 }
```

2.2 函数模板练习

```
1  #define _CRT_SECURE_NO_WARNINGS
2  #include <iostream>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <string>
6  using namespace std;
7  //函数模板 用于排序
8  template <class T>
9  void array_sort( T *a,int n)
10 {
11     for (int i = 0; i < n - 1; i++)
12     {
13         for (int j = i + 1; j < n; j++)
14         {
15             if (a[i] > a[j])
16             {
17                 T temp = a[i];
18                 a[i] = a[j];
19                 a[j] = temp;
20             }
21         }
22     }
23 }
24
25 template <class T>
26 void print_array(T *p, int n)
27 {
28     for (int i = 0; i < n; i++)
29     {
30         cout << p[i] << " ";
31     }
32     cout << endl;
33 }
34 void test01()
35 {
36     int a[10] = {1,4,3,34,5,88,17,2,69,0};
```

```

37  array_sort(a, sizeof(a)/sizeof(a[0]));
38  print_array(a, sizeof(a) / sizeof(a[0]));
39  }
40  void test02()
41  {
42      double b[5] = { 3.1 ,4.5 ,2.1 ,5.6 ,1.1 };
43      array_sort<double>(b, sizeof(b)/sizeof(b[0]));
44      print_array<double>(b, sizeof(b) / sizeof(b[0]));
45  }
46  int main()
47  {
48      test02();
49      return 0;
50  }

```

3 函数模板和普通函数的区别

- 普通函数可以自动进行类型转换
- 函数模板不能自动进行类型转换

```

1  #define _CRT_SECURE_NO_WARNINGS
2  #include <iostream>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <string>
6  using namespace std;
7  int Myadd(int a, int b)
8  {
9      cout << "普通函数" << endl;
10     return a + b;
11 }
12 template <class T>
13 T Myadd(T a, T b)
14 {
15     cout << "模板函数" << endl;
16     return a + b;
17 }
18 void test01()
19 {
20     int a = 10;

```

```

21 char b = 20;
22 Myadd(a,a); //调用普通函数 不用推导
23 Myadd<>(a, a); //指定调用模板函数
24 Myadd<int>(a, a); //指定调用模板函数
25 Myadd(a, b); //调用普通函数 因为普通的函数可以自动类型转换
26 //Myadd<>(a, b); //函数模板不会做自动类型转换
27
28 }
29
30 int main()
31 {
32     test01();
33     return 0;
34 }

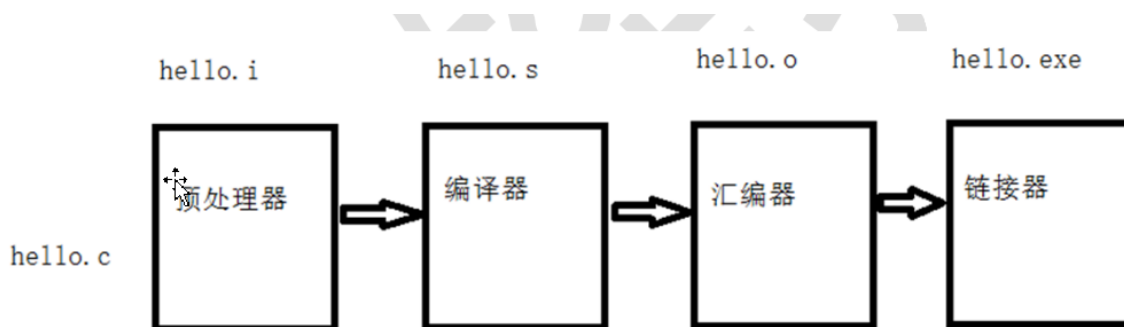
```

4 函数模板和普通函数在一起的调用规则

- C++编译器优先考虑普通函数
- 可以通过空模板实参列表的语法限定编译器只能通过模板匹配
- 函数模板可以像普通函数那样可以被重载
- 函数模板如果有更好的匹配, 优先使用函数模板

5 函数模板剖析

5.1 编译过程



hello.c 经过预处理器, 将宏展开, 生成的文件 hello.i
 hello.i 经过编译器, 将文件编译成汇编语言, 生成文件为 hello.s
 hello.s 经过汇编器, 将文件编译成目标文件 hello.o (win下为 hello.obj)
 hello.o 经过链接器, 将文件编译成可执行文件

5.2 函数模板的本质

就是进行二次编译

第一次对函数模板进行编译,第二次在调用处对函数模板展开,进行二次编译

```
template <class T>
T Myadd(T a, T b)
{
    cout << "模板函数" << endl;
    return a + b;
}
```

函数模板

第一次编译 对函数模板语法进行编译

```
Myadd(int a, int a);
```



```
int Myadd(int a, int b)
{
    cout << "模板函数" << endl;
    return a + b;
}
```

第二次编译

```
char Myadd(char a, char b)
```

```
char Myadd(char a, char b)
```

```
{
    cout << "模板函数" << endl;
    return a + b;
}
```

展开之后, 进行第二次编译

6 模板具体化

```
1  class Person
2  {
3  public:
4  Person(string name, int age)
5  {
6  this->mName = name;
7  this->mAge = age;
8  }
9  string mName;
10 int mAge;
11 };
12
13 //普通交换函数
14 template <class T>
15 void mySwap(T &a,T &b)
16 {
17 T temp = a;
18 a = b;
19 b = temp;
20 }
21 //第三代具体化,显示具体化的原型和定意思以template<>开头,并通过名称来指出类型
22 //具体化优先于常规模板
23 template<>void mySwap<Person>(Person &p1, Person &p2)
24 {
25 string nameTemp;
```

```

26  int ageTemp;
27
28  nameTemp = p1.mName;
29  p1.mName = p2.mName;
30  p2.mName = nameTemp;
31
32  ageTemp = p1.mAge;
33  p1.mAge = p2.mAge;
34  p2.mAge = ageTemp;
35
36  }
37
38  void test()
39  {
40  Person P1("Tom", 10);
41  Person P2("Jerry", 20);
42
43  cout << "P1 Name = " << P1.mName << " P1 Age = " << P1.mAge << endl;
44  cout << "P2 Name = " << P2.mName << " P2 Age = " << P2.mAge << endl;
45  mySwap(P1, P2);
46  cout << "P1 Name = " << P1.mName << " P1 Age = " << P1.mAge << endl;
47  cout << "P2 Name = " << P2.mName << " P2 Age = " << P2.mAge << endl;
48  }

```

7 类模板

7.1 类模板的实现

```

1  #define _CRT_SECURE_NO_WARNINGS
2  #include <iostream>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <string>
6  using namespace std;
7  template <class T1, class T2>
8  class Animal
9  {
10 public:
11     Animal(T1 a, T2 b)
12     {
13         age = a;

```



```

14  data = b;
15  }
16  T1 age;
17  T2 data;
18  };
19
20 void test01()
21 {
22  //类模板不能自动类型推导
23  Animal<int, int> dog(10,10);//显示指定
24  Animal<int,string> cat(4,"lili");
25
26
27 }
28 int main()
29 {
30
31  return 0;
32 }

```

7.2类模板作为函数参数

类模板作为函数的形参,该函数需要写成模板

```

1  template <class T1, class T2>
2  class Animal
3  {
4  public:
5      Animal(T1 a, T2 b)
6      {
7          age = a;
8          data = b;
9      }
10     T1 age;
11     T2 data;
12 };
13
14 void show(Animal<int, int> &p)
15 {
16     cout << p.age << " " << p.data << endl;
17 }
18

```

```

19
20 template <class T1,class T2>
21 void show(Animal<T1, T2> &p)
22 {
23     cout << p.age << " " << p.data << endl;
24 }
25
26 template <class T1>
27 void show1(T1 &p)
28 {
29     cout << p.age << " " << p.data << endl;
30 }

```

7.3类模板遇到继承

- 类模板遇到继承 在继承时,继承的类必须是一个模板类<>
- 类模板遇到继承,可以将子类写成类模板,

```

1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7
8 template <class T>
9 class Base
10 {
11 public:
12     Base(T a)
13     {
14         this->a = a;
15     }
16     T a;
17 };
18
19 class Son1 :public Base<int>
20 {
21 public:
22     Son1(int x1, int a) :Base<int>(a), x(x1)
23     {}

```

```

24  int x;
25  };
26
27  template <class T1,class T2>
28  class Son2 :public Base<T2>
29  {
30  public:
31      Son2( T1 x1,T2 a):Base<T2>(a), x(x1)
32      {}
33      T1 x;
34  };
35  void test01()
36  {
37      Son1 p(10,20);
38      Son2<int, string> p2(10,"lucy");
39  }
40  int main()
41  {
42
43      return 0;
44  }

```

7.4 类模板的成员函数类内实现

```

1  #define _CRT_SECURE_NO_WARNINGS
2  #include <iostream>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <string>
6  using namespace std;
7  template <class T1,class T2>
8  class Person
9  {
10 public:
11     Person(T1 a, T2 b)
12     {
13         this->a = a;
14         this->b = b;
15     }
16     void show()
17     {
18         cout << a <<" "<< b << endl;

```

```

19  }
20  T1 a;
21  T2 b;
22
23  };
24  void test01()
25  {
26      Person<int,string> p(10, "hello");
27      p.show();
28
29  }
30  int main()
31  {
32      test01();
33      return 0;
34  }

```

7.5 类模板的成员函数类外实现

类模板的成员函数放在类外实现需要写成函数模板

```

1  #define _CRT_SECURE_NO_WARNINGS
2  #include <iostream>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <string>
6  using namespace std;
7  template <class T1, class T2>
8  class Person
9  {
10 public:
11     Person(T1 a, T2 b);
12
13     void show();
14     T1 a;
15     T2 b;
16
17 };
18 //类模板的成员函数在类外实现 需要写成函数模板
19 template <class T1, class T2>
20 Person<T1,T2>::Person(T1 a, T2 b)
21 {

```

```

22  this->a = a;
23  this->b = b;
24  }
25  template <class T1, class T2>
26  void Person<T1, T2>::show()
27  {
28      cout << a << " " << b << endl;
29  }
30  void test01()
31  {
32      Person<int, string> p(10, "hello");
33      p.show();
34
35  }
36  int main()
37  {
38      test01();
39      return 0;
40  }

```

7.6 类模板成员函数的创建时机

类模板成员函数的创建时机是在调用时,没有调用,编译器不会创建这个函数,只有函数的声明

```

1  #define _CRT_SECURE_NO_WARNINGS
2  #include <iostream>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <string>
6  using namespace std;
7  class A
8  {
9  public:
10     void showA()
11     {
12         cout << "showA" << endl;
13     }
14 };
15 class B
16 {
17 public:

```

```

18 void showB()
19 {
20     cout << "showB" << endl;
21 }
22 };
23 template <class T>
24 class C
25 {
26 public:
27     void foo1()
28     {
29         obj.showA();
30     }
31     void foo2();
32     /*{
33         obj.showB();
34     }*/
35     T obj;
36 };
37 void test01()
38 {
39     C<A> p;
40     p.foo1();//调用foo1
41     //p.foo2();
42 }
43 int main()
44 {
45     test01();
46     return 0;
47 }

```

7.7 类模板的分文件问题

注意: 类模板的分文件,必须将函数的定义和类的声明写到一个文件

09person.h

```

1 #pragma once
2 #define _CRT_SECURE_NO_WARNINGS
3 #include <iostream>
4 #include <string.h>
5 #include <stdlib.h>

```

```

6 #include <string>
7 using namespace std;
8 template <class T1, class T2>
9 class person
10 {
11 public:
12     person(T1 a, T2 b);
13     void show();
14     T1 a;
15     T2 b;
16
17 };
18
19 template <class T1, class T2>
20 person<T1, T2>::person(T1 a, T2 b)
21 {
22     this->a = a;
23     this->b = b;
24 }
25
26 template <class T1, class T2>
27 void person<T1, T2>::show()
28 {
29     cout << a << " " << b << endl;
30 }

```

main.cpp

```

1 #include "09person.hpp"
2
3 int main()
4 {
5     //调用构造函数和show函数需要创建,但是没有这两个函数的定义,不能创建
6     person<int, int> p(10, 20);//
7     p.show();
8
9     return 0;
10 }

```

7.8类模板遇到友元

```

1 #define _CRT_SECURE_NO_WARNINGS

```

```

2  #include <iostream>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <string>
6  using namespace std;
7  template <class T1, class T2>
8  class Person;
9
10 template <class T1, class T2>
11 void showPerson1(Person<T1, T2> &p);
12
13 //类模板作为函数形参 函数需要学成 函数模板
14 template <class T1, class T2>
15 void showPerson(Person<T1, T2> &p)
16 {
17     cout << p.a << " " << p.b << endl;
18 }
19
20
21
22 template <class T1, class T2>
23 class Person
24 {
25     friend void showPerson1<>(Person<T1, T2> &p);
26     friend void showPerson<>(Person<T1, T2> &p);
27     friend void showPerson2(Person<T1, T2> &p); //定义一个全局函数并且声明为类的友元
28 {
29     cout << p.a << " " << p.b << endl;
30 }
31 public:
32     Person(T1 a, T2 b)
33     {
34         this->a = a;
35         this->b = b;
36     }
37 private:
38     T1 a;
39     T2 b;
40 };
41

```



```
42 template <class T1, class T2>
43 void showPerson1(Person<T1, T2> &p)
44 {
45     cout << p.a << " " << p.b << endl;
46 }
47
48 void test01()
49 {
50     Person<int, string> p(10, "lucy");
51     showPerson(p);
52     showPerson1(p);
53     showPerson2(p);
54 }
55
56 int main()
57 {
58     test01();
59     return 0;
60 }
```