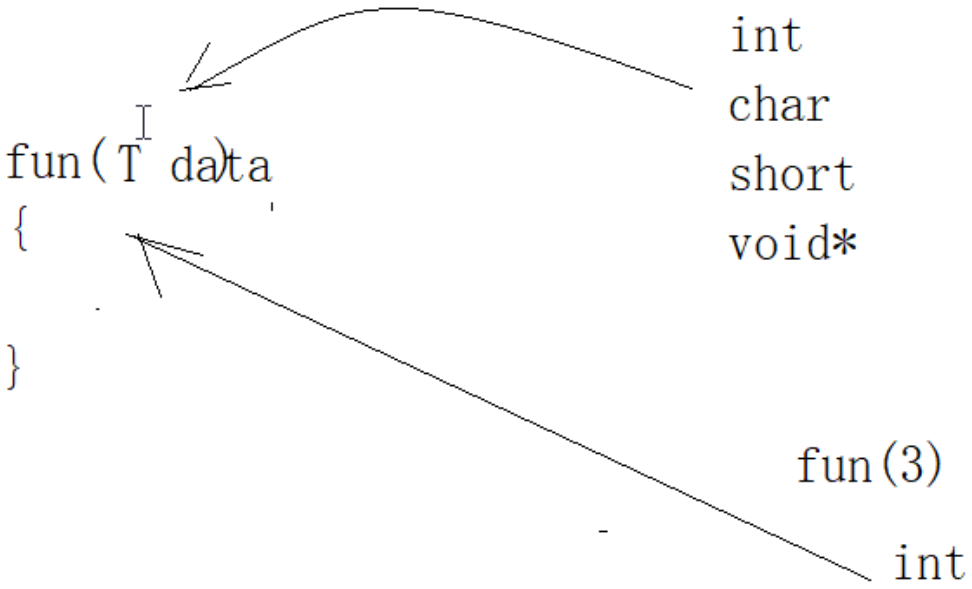
1 模板概述

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
void swap(int &x, int &y)
{
 int temp = x;
  x = y;
  y = temp;
}

void swap(char &x, char &y)
{
  char temp = x;
```

```
 x = y;
 y = temp;
}
void test01()
{
 int a = 1;
 int b = 2;
 swap(a,b);
 cout << a << " " << b << endl;
}
void test02()
{
 char a = 1;
 char b = 2;
 swap(a, b);
 cout << a << " " << b << endl;
}
//函数模板来实现
template <class T>//定义一个模板 模板的通用类型为T
//紧跟函数的定义
void swap_temp(T &a, T &b)
{
 T temp = a;
 a = b;
 b = temp;
}
void test03()
{
 char a = 1;
 char b = 2;
 int c = 3;
 int d = 4;
 swap_temp(a, b);//自动推导
 //swap_temp(a,c); 自动类型推导的结果不一致
 swap_temp(c, d);
}

int main()
{
 test01();
```

```
 return 0;
}
```

2.2 函数模板练习

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
//函数模板 用于排序
template <class T>
void array_sort( T *a,int n)
{
 for (int i = 0; i < n - 1; i++)
 {
 for (int j = i + 1; j < n; j++)
 {
 if (a[i] > a[j])
 {
 T temp = a[i];
 a[i] = a[j];
 a[j] = temp;
 }
 }

 }
}
template <class T>
void print_array(T *p, int n)
{
 for (int i = 0; i < n; i++)
 {
 cout << p[i] << " ";
 }
 cout << endl;
}
void test01()
{
 int a[10] = {1,4,3,34,5,88,17,2,69,0};
```

```
 array_sort(a,sizeof(a)/sizeof(a[0]));
 print_array(a, sizeof(a) / sizeof(a[0]));
}
void test02()
{
 double b[5] = { 3.1 ,4.5 ,2.1 ,5.6 ,1.1 };
 array_sort<double>(b,sizeof(b)/sizeof(b[0]));
 print_array<double>(b, sizeof(b) / sizeof(b[0]));
}
int main()
{
 test02();
 return 0;
}
```

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
int Myadd(int a, int b)
{
 cout << "普通函数" << endl;
 return a + b;
}
template <class T>
T Myadd(T a, T b)
{
 cout << "模板函数" << endl;
 return a + b;
}
void test01()
{
 int a = 10;
```

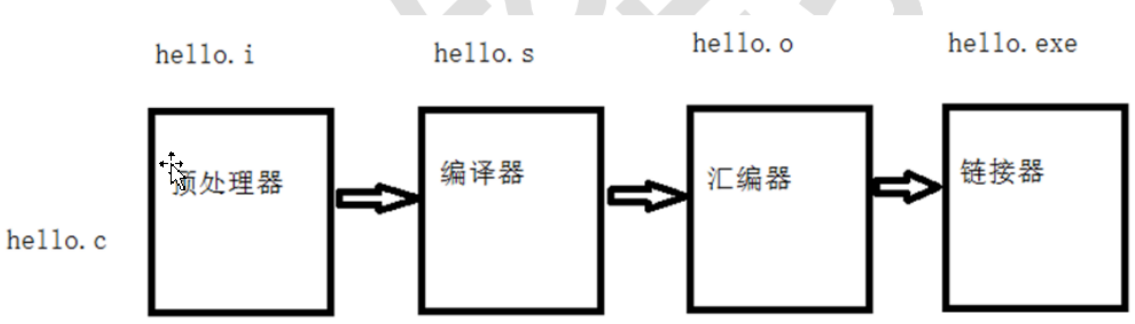
```
21  char b = 20;
22  Myadd(a,a);//调用普通函数 不用推导
23  Myadd<>(a, a);//指定调用模板函数
24  Myadd<int>(a, a);//指定调用模板函数
25  Myadd(a, b);//调用普通函数 因为普通的函数可以自动类型转换
26  //Myadd<>(a, b);//函数模板不会做自动类型转换
27
28 }
29
30 int main()
31 {
32  test01();
33  return 0;
34 }
```

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

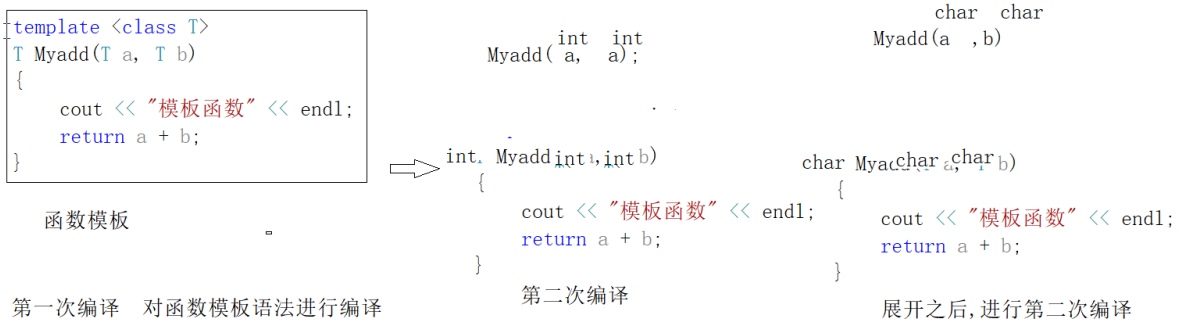


hello.c经过预处理器，将宏展开，生成的文件hello.i
hello.i经过编译器，将文件编译成汇编语言，生成文件为hello.s
hello.s经过汇编器，将文件编译成目标文件hello.o（win下为hello.obj）
hello.o经过链接器，将文件编译成可执行文件

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



6 模板具体化

```
class Person
{
public:
Person(string name, int age)
{
this->mName = name;
this->mAge = age;
}
string mName;
 int mAge;
 };

 //普通交换函数
 template <class T>
 void mySwap(T &a,T &b)
 {
 T temp = a;
 a = b;
 b = temp;
 }
 //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
 //具体化优先于常规模板
 template<>void  mySwap<Person>(Person &p1, Person &p2)
 {
 string nameTemp;
```

```
int ageTemp;

nameTemp = p1.mName;
p1.mName = p2.mName;
p2.mName = nameTemp;

ageTemp = p1.mAge;
p1.mAge = p2.mAge;
p2.mAge = ageTemp;

}

void test()
{
Person P1("Tom", 10);
Person P2("Jerry", 20);

cout << "P1 Name = " << P1.mName << " P1 Age = " << P1.mAge << endl;
cout << "P2 Name = " << P2.mName << " P2 Age = " << P2.mAge << endl;
mySwap(P1, P2);
cout << "P1 Name = " << P1.mName << " P1 Age = " << P1.mAge << endl;
cout << "P2 Name = " << P2.mName << " P2 Age = " << P2.mAge << endl;
}
```

7 类模板

7.1 类模板的实现

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
template <class T1, class T2>
class Animal
{
public:
 Animal(T1 a, T2 b)
 {
 age = a;
```

```
 data = b;
 }
 T1 age;
 T2 data;
};

void test01()
{
 //类模板不能自动类型推导
 Animal<int, int> dog(10,10);//显示指定
 Animal<int,string> cat(4,"lili");


}
int main()
{

 return 0;
}
```

7.2类模板作为函数参数

类模板作为函数的形参,该函数需要写成韩顺模板

```
template <class T1, class T2>
class Animal
{
public:
 Animal(T1 a, T2 b)
 {
 age = a;
 data = b;
 }
 T1 age;
 T2 data;
};

void show(Animal<int, int> &p)
{
 cout << p.age << " " << p.data << endl;
}

```

```
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  {}
```

```
 int x;
};

template <class T1,class T2>
class Son2 :public Base<T2>
{
public:
 Son2( T1 x1,T2 a):Base<T2>(a), x(x1)
 {]}
 T1 x;
};
void test01()
{
 Son1 p(10,20);
 Son2<int, string> p2(10,"lucy");
}
int main()
{

 return 0;
}
```

7.4 类模板的成员函数类内实现

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
template <class T1,class T2>
class Person
{
public:
 Person(T1 a, T2 b)
 {
 this->a = a;
 this->b = b;
 }
 void show()
 {
 cout << a <<" "<< b << endl;
```

```
 }
 T1 a;
 T2 b;

};
void test01()
{
 Person<int,string> p(10, "hello");
 p.show();

}
int main()
{
 test01();
 return 0;
}
```

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
template <class T1, class T2>
class Person
{
public:
 Person(T1 a, T2 b);

 void show();
 T1 a;
 T2 b;

};
//类模板的成员函数在类外实现 需要写成函数模板
template <class T1, class T2>
Person<T1,T2>::Person(T1 a, T2 b)
{
```

```
 this->a = a;
 this->b = b;
}
template <class T1, class T2>
void Person<T1, T2>::show()
{
 cout << a << " " << b << endl;
}
void test01()
{
 Person<int, string> p(10, "hello");
 p.show();

}
int main()
{
 test01();
 return 0;
}
```

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class A
{
public:
  void showA()
  {
  cout << "showA" << endl;
  }
};
class B
{
public:
```

```
 void showB()
 {
 cout << "showB" << endl;
 }
};
template <class T>
class C
{
public:
 void foo1()
 {
 obj.showA();
 }
 void foo2();
 /*{
 obj.showB();
 }*/
 T obj;
};
void test01()
{
 C<A> p;
 p.foo1();//调用foo1
 //p.foo2();
}
int main()
{
 test01();
 return 0;
}
```

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

```
#pragma once
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
```

```cpp
#include <string>
using namespace std;
template <class T1,class T2>
class person
{
public:
 person(T1 a, T2 b);
 void show();
 T1 a;
 T2 b;

};

template <class T1, class T2>
person<T1, T2>::person(T1 a, T2 b)
{
 this->a = a;
 this->b = b;
}

template <class T1, class T2>
void person<T1, T2>::show()
{
 cout << a << " " << b << endl;
}
```

main.cpp

```cpp
#include "09person.hpp"

int main()
{
 //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
 person<int, int> p(10, 20);//
 p.show();

 return 0;
}
```

7.8类模板遇到友元

```cpp
#define _CRT_SECURE_NO_WARNINGS
```

```cpp
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
template <class T1, class T2>
class Person;

template <class T1, class T2>
void showPerson1(Person<T1, T2> &p);

//类模板作为函数形参 函数需要学成 函数模板
template <class T1, class T2>
void showPerson(Person<T1, T2> &p)
{
 cout << p.a << " " << p.b << endl;
}



template <class T1,class T2>
class Person
{
 friend void showPerson1<>(Person<T1, T2> &p);
 friend void showPerson<>(Person<T1, T2> &p);
 friend void showPerson2(Person<T1, T2> &p)//定义一个全局函数并且声明为类的友元
 {
 cout << p.a << " " << p.b << endl;
 }
public:
 Person(T1 a, T2 b)
 {
 this->a = a;
 this->b = b;
 }
private:
 T1 a;
 T2 b;
};

```

```cpp
template <class T1, class T2>
void showPerson1(Person<T1, T2> &p)
{
 cout << p.a << " " << p.b << endl;
}

void test01()
{
 Person<int, string> p(10, "lucy");
 showPerson(p);
 showPerson1(p);
 showPerson2(p);
}

int main()
{
 test01();
 return 0;
}
```