8 模板

8.1 模板起源

8.1.1 针对具体类型的实现

C和C++语言的静态类型系统，在满足效率与安全性要求的同时，很大程度上也成为阻碍程序员编写通用代码的桎梏。它迫使人们不得不为每一种数据类型编写完全或几乎完全相同的实现，虽然它们在抽象层面上是一致的。

```cpp
// typed.cpp

// 针对不同的类型定义不同的函数

#include <iostream>

using namespace std;

// 整型版本

int min_int(int x, int y) {
    return x < y ? x : y;
}

int max_int(int x, int y) {
    return x < y ? y : x;
}

// 双精度型版本

double min_double(double x, double y) {
    return x < y ? x : y;
}

double max_double(double x, double y) {
    return x < y ? y : x;
}

// 字符串型版本

string min_string(string x, string y) {
    return x < y ? x : y;
}

string max_string(string x, string y) {
    return x < y ? y : x;
}

// 测试用例

int main(void) {
    cout << "Enter two integer: " << flush;
    int nx, ny;
```

```
    cin >> nx >> ny;
    cout << "min=" << min_int(nx, ny) << endl;
    cout << "max=" << max_int(nx, ny) << endl;

    cout << "Enter two double: " << flush;
    double dx, dy;
    cin >> dx >> dy;
    cout << "min=" << min_double(dx, dy) << endl;
    cout << "max=" << max_double(dx, dy) << endl;

    cout << "Enter two string: " << flush;
    string sx, sy;
    cin >> sx >> sy;
    cout << "min=" << min_string(sx, sy) << endl;
    cout << "max=" << max_string(sx, sy) << endl;

    cout << "Enter two string: " << flush;
    char cx[256], cy[256];
    cin >> cx >> cy;
    cout << "min=" << min_string(cx, cy) << endl;
    cout << "max=" << max_string(cx, cy) << endl;

    return 0;
}
```

8.1.2 借助参数宏摆脱类型的限制

宏定义只是在预处理器的作用下，针对源代码的文本替换，其本身不具备函数语义。因此借助于参数宏（又名宏函数）可以在某种程度上使程序的编写者摆脱那些源于类型的约束和限制，但同时也丧失了类型的安全性。

```
// untyped.cpp

// 借助参数宏摆脱了类型限制，同时也因为失去类型检查而引入风险

#include <iostream>

using namespace std;

// 宏----去类型化

#define min(x, y) ((x) < (y) ? (x) : (y))
#define max(x, y) ((x) < (y) ? (y) : (x))

// 测试用例

int main(void) {
    cout << "Enter two integer: " << flush;
    int nx, ny;
    cin >> nx >> ny;
    cout << "min=" << min(nx, ny) << endl;
    cout << "max=" << max(nx, ny) << endl;

    cout << "Enter two double: " << flush;
    double dx, dy;
```

```
    cin >> dx >> dy;
    cout << "min=" << min(dx, dy) << endl;
    cout << "max=" << max(dx, dy) << endl;

    cout << "Enter two string: " << flush;
    string sx, sy;
    cin >> sx >> sy;
    cout << "min=" << min(sx, sy) << endl;
    cout << "max=" << max(sx, sy) << endl;

    cout << "Enter two string: " << flush;
    char cx[256], cy[256];
    cin >> cx >> cy;
    cout << "min=" << min(cx, cy) << endl; // ?
    cout << "max=" << max(cx, cy) << endl; // ?

    return 0;
}
```

8.1.3 让预编译器写代码

利用宏定义构建通用代码的框架，让预处理器将其扩展为针对不同类型的具体版本。将宏的一般性和函数的类型安全性完美地结合起来。

```
// macro.cpp

// 借助预处理器产生类似模板的功能

#include <iostream>

using namespace std;

// 模板宏

#define MIN(T) \
T min_##T(T x, T y) { \
    return x < y ? x : y; \
}

#define MAX(T) \
T max_##T(T x, T y) { \
    return x < y ? y : x; \
}

// 将模板宏扩展为具体函数

MIN(int)
MAX(int)

MIN(double)
MAX(double)

MIN(string)
MAX(string)

```

```cpp
// 函数名宏

#define min(T) min_##T
#define max(T) max_##T

// 测试用例

int main(void) {
    cout << "Enter two integer: " << flush;
    int nx, ny;
    cin >> nx >> ny;
    cout << "min=" << min(int)(nx, ny) << endl;
    cout << "max=" << max(int)(nx, ny) << endl;

    cout << "Enter two double: " << flush;
    double dx, dy;
    cin >> dx >> dy;
    cout << "min=" << min(double)(dx, dy) << endl;
    cout << "max=" << max(double)(dx, dy) << endl;

    cout << "Enter two string: " << flush;
    string sx, sy;
    cin >> sx >> sy;
    cout << "min=" << min(string)(sx, sy) << endl;
    cout << "max=" << max(string)(sx, sy) << endl;

    cout << "Enter two string: " << flush;
    char cx[256], cy[256];
    cin >> cx >> cy;
    cout << "min=" << min(string)(cx, cy) << endl;
    cout << "max=" << max(string)(cx, cy) << endl;

    return 0;
}
```

8.2 函数模板

```cpp
// functmpl.cpp

// 函数模板

#include <iostream>

using namespace std;

// 定义函数模板

template<class T>
T min(T x, T y) {
    return x < y ? x : y;
}

template<typename T>
T max(T x, T y) {
    return x < y ? y : x;
```

```
}

// 使用函数模板

int main (void) {
    cout << "Enter two integer: " << flush;
    int nx, ny;
    cin >> nx >> ny;
    cout << "min=" << ::min<int>(nx, ny) << endl;
    cout << "max=" << ::max<int>(nx, ny) << endl;

    cout << "Enter two double: " << flush;
    double dx, dy;
    cin >> dx >> dy;
    cout << "min=" << ::min<double>(dx, dy) << endl;
    cout << "max=" << ::max<double>(dx, dy) << endl;

    cout << "Enter two string: " << flush;
    string sx, sy;
    cin >> sx >> sy;
    cout << "min=" << ::min<string>(sx, sy) << endl;
    cout << "max=" << ::max<string>(sx, sy) << endl;

    cout << "Enter two string: " << flush;
    char cx[256], cy[256];
    cin >> cx >> cy;
    cout << "min=" << ::min<string>(cx, cy) << endl;
    cout << "max=" << ::max<string>(cx, cy) << endl;

    return 0;
}
```

8.2.1 函数模板的定义

8.2.1.1 模板参数的语法形式

模板的类型参数必须用如下形式的语法来声明：

```
template<typename 类型形参1, typename 类型形参2, ...>
```

例如：

```
template<typename A, typename b, typename _C>
A function(b arg) {
    _C var;
    ...
}
```

这里的typename关键字也可以换成class，但建议使用typename。

8.2.1.2 类型参数

可以使用任何标识符作为类型参数的名称，但使用“T”已经成为了一种惯例。类型参数“T”表示的是，调用者调用这个函数时所指定的任意类型。

可以将任何具体的类型实参（基本类型、自定义类型、类类型等）传递给模板的类型形参，前提是所使用的类型实参必须能够满足该模板所需要的操作。比如将一个不支持小于操作符（<）的类型实参，传递给min或max模板的类型形参，将引发编译错误。

```
template<class T>
T min(T x, T y) {
    return x < y ? x : y; // 传递给函数模板min的类型形参T的类型实参，必须支持小于操作符（<）
}

template<typename T>
T max(T x, T y) {
    return x < y ? y : x; // 传递给函数模板max的类型形参T的类型实参，必须支持小于操作符（<）
}
```

8.2.2 函数模板的使用

8.2.2.1 模板实例化

通常而言，并不是把模板编译成一个可以处理任何类型的单一实体，而是针对传递给模板类型形参的每一种类型实参，都从模板产生出一个以具体类型实现的独立实体。这种以具体类型代替类型参数的过程称为模板的实例化。每个以具体类型实现的独立实体都是该模板的一个实例。

函数模板min的三个实例：

```
int min(int x, int y) {
    return x < y ? x : y;
}

double min(double x, double y) {
    return x < y ? x : y;
}

string min(string x, string y) {
    return x < y ? x : y;
}
```

函数模板max的三个实例：

```
int max(int x, int y) {
    return x < y ? y : x;
}

double max(double x, double y) {
    return x < y ? y : x;
}

string max(string x, string y) {
    return x < y ? y : x;
}
```

8.2.2.2 使用函数模板时才实例化

函数模板的类型实参放在尖括号中，调用实参放在圆括号中，类型实参必须位于函数模板名和调用实参之间：

```
函数模板名<类型实参1, 类型实参2, ...>(调用实参1, 调用实参2, ...);
```

只要使用函数模板，编译器就会自动引发这样一个实例化过程，程序员并不需要额外地请求对模板实例化。例如：

- `min<int>(nx, ny)` 实例化出 `int min(int x, int y) { ... }` 函数，并调用之
- `max<int>(nx, ny)` 实例化出 `int max(int x, int y) { ... }` 函数，并调用之
- `min<double>(dx, dy)` 实例化出 `double min(double x, double y) { ... }` 函数并调用之
- `max<double>(dx, dy)` 实例化出 `double max(double x, double y) { ... }` 函数并调用之
- `min<string>(sx, sy)` 实例化出 `string min(string x, string y) { ... }` 函数并调用之
- `max<string>(sx, sy)` 实例化出 `string max(string x, string y) { ... }` 函数并调用之
- `min<string>(cx, cy)` 直接调用已经实例化的 `string min(string x, string y) { ... }` 函数
- `max<string>(cx, cy)` 直接调用已经实例化的 `string max(string x, string y) { ... }` 函数

8.2.2.3 二次编译

每个函数模板事实上都被编译了两次：

- 一次是在实例化之前，先检查模板代码本身，查看语法是否正确
- 另一次是在实例化期间，结合传入的类型实参，再次检查模板代码，查看针对具体类型的所有操作是否都有效

只有在第二次编译时，才会真正产生针对具体函数的二进制机器指令，第一次编译仅仅在编译器内部形成的一个用于描述函数模板的内部表示。

8.2.3 函数模板的隐式推断

8.2.3.1 根据调用参数推断模板参数

如果函数模板调用参数的类型相关于该模板的类型参数，那么在调用该函数模板时，即使不显式指明类型实参，编译器也有能力根据调用参数的类型隐式推断出正确的类型实参，以获得与普通函数调用一致的语法表达。例如：

```
int main (void) {
    cout << "Enter two integer: " << flush;
    int nx, ny;
    cin >> nx >> ny;
    cout << "min=" << ::min(nx, ny) << endl;
    cout << "max=" << ::max(nx, ny) << endl;

    cout << "Enter two double: " << flush;
    double dx, dy;
    cin >> dx >> dy;
    cout << "min=" << ::min(dx, dy) << endl;
    cout << "max=" << ::max(dx, dy) << endl;

    cout << "Enter two string: " << flush;
    string sx, sy;
    cin >> sx >> sy;
    cout << "min=" << ::min(sx, sy) << endl;
    cout << "max=" << ::max(sx, sy) << endl;

    cout << "Enter two string: " << flush;
    char cx[256], cy[256];
    cin >> cx >> cy;
    cout << "min=" << ::min<string>(cx, cy) << endl;
    cout << "max=" << ::max<string>(cx, cy) << endl;

    return 0;
}
```

注意在cx和cy中获取最小和最大值的函数模板调用没有使用隐式推断：

```
cout << "min=" << ::min(cx, cy) << endl;
cout << "max=" << ::max(cx, cy) << endl;
```

而依然使用显式实例化：

```
cout << "min=" << ::min<string>(cx, cy) << endl;
cout << "max=" << ::max<string>(cx, cy) << endl;
```

因为cx和cy是字符数组，其类型为char*，隐式推断将以char*作为实例化函数模板的类型实参，实例化结果为：

```cpp
char* min(char* x, char* y) {
    return x < y ? x : y;
}

char* max(char* x, char* y) {
    return x < y ? y : x;
}
```

对由字符指针指向的空字符结尾的C风格字符串，使用小于号比较大小，显然是不合适的。

8.2.3.2 不能隐式推断的三种情况

以下三种情况，不能隐式推断，必须显式指定模板参数：

- 不是全部模板参数都与调用参数的类型相关

```cpp
template<typename V, typename A> void foo(A arg) { ... V var ... }
```

```cpp
foo(1.23); // 错误
foo<int, double>(1.23); // 正确
foo<int>(1.23); // 正确
```

- 隐式推断的同时不允许隐式类型转换

```cpp
template<typename T> void foo(T x, T y) { ... }
```

```cpp
foo(123, 4.56); // 错误
foo<double>(123, 4.56); // 正确
foo(123, (int)4.56); // 正确
```

- 返回类型不能隐式推断

```cpp
template<typename R, typename A> R foo(A arg) { ... }
```

```cpp
int ret = foo(1.23); // 错误
int ret = foo<int, double>(1.23); // 正确
int ret = foo<int>(1.23); // 正确
```

```cpp
// deduction.cpp

// 函数模板参数的隐式推断

#include <iostream>
#include <typeinfo>

using namespace std;

template<typename T>
void foo(T const& x, T const& y) {
    cout << typeid(T).name() << endl;
}
```

```cpp

template<typename R, typename A>
R bar(A const& arg) {
    R ret;
    cout << typeid(R).name() << ' ' << typeid(A).name() << endl;
    return ret;
}

int main(void) {
    // 隐式实例化函数模板

    int a, b;
    foo(a, b);

    double c, d;
    foo(c, d);

    char e[256], f[256];
    foo(e, f);
    foo("hello", "world");
    // foo("hello", "tarena"); // char[6] != char[7]

    string g, h;
    foo(g, h);

    // 隐式推断不允许隐式类型转换
    // foo(a, c);
    // 但可以显式类型转换
    foo(a, (int)c);
    foo(static_cast<double>(a), c);
    // 显式实例化函数模板
    foo<int>(a, c);
    foo<double>(a, c);

    int i;
    double j;
    // 返回类型不能隐式推断
    // i = bar(j);
    i = bar<int, double>(j);
    i = bar<int>(j);

    return 0;
}
```

8.2.3.3 隐式推断与缺省值之间的矛盾

函数模板参数的隐式推断与缺省值之间存在矛盾，因此函数模板的模板参数不能带有缺省值。例如：

```cpp
template<typename T = int> T min(T x, T y) { return x < y ? x : y; } // 错误
```

```cpp
cout << ::min(1.23, 4.56) << endl; // T=double? T=int?
```

C++11以后，允许为函数模板的模板参数指定缺省值，但依然会优先选择隐式推断的结果。

```cpp
// defargs.cpp

// C++11以后，允许为函数模板的模板参数指定缺省值，但依然会优先选择隐式推断的结果

#include <iostream>

using namespace std;

template<typename T = int>
T min(T x, T y) {
    return x < y ? x : y;
}

int main(void) {
    cout << ::min(1.23, 4.56) << endl; // T=double, 1.23

    return 0;
}
```

8.2.4 函数模板的重载

```cpp
// overload.cpp

// 函数模板重载

#include <string.h>

#include <iostream>
#include <typeinfo>

using namespace std;

// 求两个C风格字符串的最小值
char const* min(char const* x, char const* y) {
    cout << "<1" << typeid(char const*).name() << "> " << flush;
    return strcmp(x, y) < 0 ? x : y;
}

// 求两个任意类型值的最小值
template<typename T>
T min(T x, T y) {
    cout << "<2" << typeid(T).name() << "> " << flush;
    return x < y ? x : y;
}

// 求两个任意类型指针所指向目标的最小值
template<typename T>
T* min(T* x, T* y) {
    cout << "<3" << typeid(T).name() << "> " << flush;
    return *x < *y ? x : y;
}

// 求三个任意类型值的最小值
template<typename T>
T min(T x, T y, T z) {
```

```
    cout << "<4" << typeid(T).name() << "> " << flush;
    return ::min(::min(x, y), z);
}

// 求两个C风格字符串的最小值
// char const* min(char const* x, char const* y) {
//     cout << "<1" << typeid(char const*).name() << "> " << flush;
//     return strcmp(x, y) < 0 ? x : y;
// }

int main(void) {
    // 编译器优先选择普通函数
    cout << ::min("hello", "world") << endl; // <1PKc> hello
    // 除非函数模板能够产生具有更好匹配性的函数实例
    cout << ::min(123, 456) << endl; // <2i> 123

    // 在参数传递过程中如需隐式类型转换，编译器将优先选择普通函数
    char s[] = "hello";
    cout << ::min(s, "world") << endl; // <1PKc> hello

    // 通过空模板参数列表告知编译器使用函数模板
    // 针对指针的版本显然比任意类型版本更加匹配
    cout << ::min<>("hello", "hellooo") << endl; // <3c> hellooo

    // 显式指定的模板参数必须在所选择的重载版本中与调用参数的类型保持一致
    cout << ::min<char const*>("hello", "world") << endl; // <2PKc> world

    // 在函数模板的实例化函数中，编译器仍然优先选择普通函数
    cout << ::min("hello", "world", "tarena") << endl; // <4PKc> <1PKc>
<1PKc> hello

    return 0;
}
```

8.2.4.1 普通函数和函数模板构成重载

普通函数和可实例化为该函数的函数模板构成重载关系。在其它条件都相同的情况下，编译器优先选择普通函数，除非函数模板能够产生具有更好匹配性的函数实例。

```
char const* min(char const* x, char const* y) { ... } // 普通函数
template<typename T> T min(T x, T y) { ... } // 函数模板
```

```
cout << ::min("hello", "world") << endl; // 调用普通函数
cout << ::min(123, 456) << endl; // 调用函数模板实例
```

8.2.4.2 函数模板不支持隐式类型转换

函数模板的隐式实例化不支持隐式类型转换，但普通函数支持。因此在参数传递过程中如需隐式类型转换，编译器将优先选择普通函数。

```
char const* min(char const* x, char const* y) { ... } // 普通函数
template<typename T> T min(T x, T y) { ... } // 函数模板
```

```
char s[] = "hello";
cout << ::min(s, "world") << endl; // 调用普通函数，char*->char const*
```

8.2.4.3 显式指定空模板参数列表

可以显式指定一个空的模板参数列表，明确告知编译器使用函数模板，但模板参数却由隐式推断决定。即便是在函数模板中选择，编译器也会尽可能选择类型约束性强的版本，即更特殊的版本。

```
char const* min(char const* x, char const* y) { ... } // 普通函数
template<typename T> T min(T x, T y) { ... } // 函数模板
template<typename T> T* min(T* x, T* y) { ... } // 更特殊的函数模板
```

```
cout << ::min<>("hello", "hellooo") << endl; // 调用更特殊的函数模板实例
```

8.2.4.4 保证模板参数与调用参数一致

如果为函数模板显式指定了模板参数，那么所选择的重载版本必须能够保证模板参数与调用参数的类型一致。

```
char const* min(char const* x, char const* y) { ... } // 普通函数
template<typename T> T min(T x, T y) { ... } // 函数模板
template<typename T> T* min(T* x, T* y) { ... } // 更特殊的函数模板
```

```
cout << ::min<char const*>("hello", "world") << endl; // 调用函数模板实例
```

8.2.4.5 函数模板内优先选择普通函数

即使是在函数模板的实例化函数中，编译器仍然优先选择普通函数，前提是该普通函数在一次编译时可见。

```
char const* min(char const* x, char const* y) { ... } // 普通函数
template<typename T> T min(T x, T y) { ... } // 两参函数模板
template<typename T> T min(T x, T y, T z) {
    return ::min(::min(x, y), z);
} // 三参函数模板
```

```
cout << ::min("hello", "world", "tarena") << endl; // 调用三参函数模板实例，后者调用普通函数
```

8.2.4.6 重载函数模板最好只针对参数的个数或具体类型

```
// bad.cpp

// 重载函数模板最好只针对参数的个数或具体类型，否则极可能导致非预期的后果

#include <string.h>

#include <iostream>
#include <typeinfo>

using namespace std;
```

```cpp

template<typename T>
T const& min(T const& x, T const& y) {
    return x < y ? x : y;
}

// 正确的写法
char const* const& min(char const* const& x, char const* const& y) {
    return strcmp(x, y) < 0 ? x : y;
}

/* 错误的写法
char const* min(char const* x, char const* y) {
    return strcmp(x, y) < 0 ? x : y;
}
*/
template<typename T>
T const& min(T const& x, T const& y, T const& z) {
    return ::min(::min(x, y), z); // 返回对匿名局部变量的引用
}

int main(void) {
    char const* a = "a";
    char const* b = "ab";
    char const* c = "abc";
    char const* const& d = ::min(a, b, c);
    cout << d << endl;

    return 0;
}
```

8.3 类模板

```cpp
// clstmpl.cpp

// 类模板

#include <iostream>

using namespace std;

// 定义类模板

template<class T>
class Comparator {
public:
    Comparator(T x, T y) : x(x), y(y) {}

    T min(void) const {
        return x < y ? x : y;
    }

    T max(void) const {
        return x < y ? y : x;
    }
```

```
23
24 private:
25     T x, y;
26 };
27
28 // 使用类模板
29
30 int main (void) {
31     cout << "Enter two integer: " << flush;
32     int nx, ny;
33     cin >> nx >> ny;
34     Comparator<int> cn(nx, ny);
35     cout << "min=" << cn.min() << endl;
36     cout << "max=" << cn.max() << endl;
37
38     cout << "Enter two double: " << flush;
39     double dx, dy;
40     cin >> dx >> dy;
41     Comparator<double> cd(dx, dy);
42     cout << "min=" << cd.min() << endl;
43     cout << "max=" << cd.max() << endl;
44
45     cout << "Enter two string: " << flush;
46     string sx, sy;
47     cin >> sx >> sy;
48     Comparator<string> cs(sx, sy);
49     cout << "min=" << cs.min() << endl;
50     cout << "max=" << cs.max() << endl;
51
52     cout << "Enter two string: " << flush;
53     char cx[256], cy[256];
54     cin >> cx >> cy;
55     cs = Comparator<string>(cx, cy);
56     cout << "min=" << cs.min() << endl;
57     cout << "max=" << cs.max() << endl;
58
59     return 0;
60 }
```

8.3.1 类模板的定义

8.3.1.1 模板参数的语法形式

模板的类型参数必须用如下形式的语法来声明：

```
1 template<typename 类型形参1, typename 类型形参2, ...>
```

例如：

```
template<typename A, typename b, typename _C>
class MyClass {
public:
    A a;
    b foo(_C c) { ... };
};
```

这里的typename关键字也可以换成class，但建议使用typename。

8.3.1.2 类型参数

在类模板的内部，类型参数可以象其它任何具体类型一样，用于成员变量、成员函数、成员类型（内部类型），甚至基类的声明。例如：

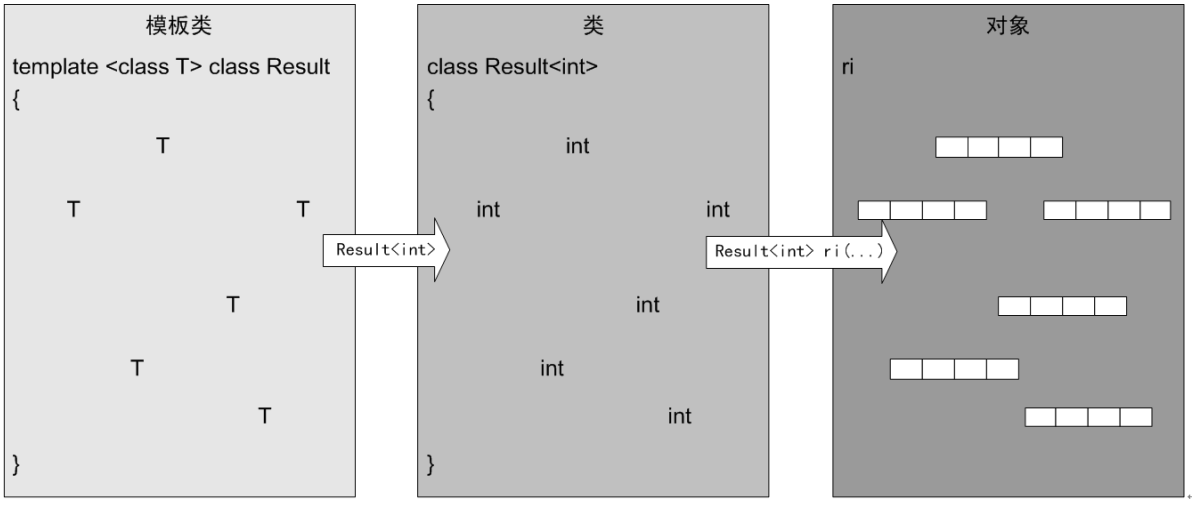
```
template<typename M, typename R, typename A, typename V, typename T, typename B>
class MyClass : public B {
    M mem;
    R function(A arg) { ... V var ... }
    typedef T* pointer;
};
```

8.3.2 类模板的使用

8.3.2.1 类模板的两步实例化

从类模板到对象实际上经历了两个实例化过程

- 编译期：编译器将类模板实例化为类并生成对象创建指令
- 运行期：处理器执行对象创建指令将类实例化为内存对象



类模板本身并不代表一个确定的类型，既不能用于定义对象，也不能用于声明指针或引用。只有通过模板实参将其实例化为具体类以后，才具备类型语义。

```
类模板名<类型实参1, 类型实参2, ...>
```

8.3.2.2 调用谁实例化谁

类模板中，只有那些实际被调用的成员函数才会被实例化，即产生二进制代码。某些类型虽然并没有提供类模板所需要的全部功能，但照样可以实例化该类模板，只要不直接或间接调用那些依赖于未提供功能的成员函数即可。

```
template<class T>
class Comparator {
public:
    Comparator(T x, T y) : x(x), y(y) {}

    T min(void) const {
        return x < y ? x : y; // 若调用此函数，实例化该模板的类型，必须支持小于操作符（<）
    }

    T max(void) const {
        return x > y ? x : y; // 若调用此函数，实例化该模板的类型，必须支持大于操作符（>）
    }

private:
    T x, y;
};
```

```
class Integer {
public:
    Integer(int n) : n(n) {}

    // 只实现了小于操作符（<）
    bool operator<(Integer const& i) const {
        return n < i.n;
    }

    // 没有实现大于操作符（>）

private:
    int n;
};
```

```
Comparator<Integer> cn(123, 456); // 可以用Integer类实例化Comparator类模板
cout << cn.min() << endl; // 只要不调用max函数就没问题
```

8.3.2.3 类模板参数不支持隐式推断

与函数模板不同，类模板的类型参数不支持隐式推断。

```
template<class T>
class Comparator {
public:
    Comparator(T x, T y) : x(x), y(y) {}
    ...
};
```

```
int nx, ny;
...
Comparator<int> cn(nx, ny); // 正确
Comparator cn(nx, ny); // 错误
```

8.3.3 静态成员与递归实例化

8.3.3.1 类模板的静态成员

类模板的静态成员变量，既不是一个对象一份，也不是一个模板一份，而是在该类模板的每个实例化类中，各有一份独立的拷贝，且为该类的所有实例化对象所共享。

```
// static.cpp

// 类模板的静态成员

#include <iostream>

using namespace std;

template<typename T>
class A {
public:
    static void foo(void) {
        cout << &n << ' ' << &t << endl;
    }

private:
    static int n;
    static T t;
};

template<typename T> int A<T>::n;
template<typename T> T A<T>::t;

int main(void) {
    A<int> a, b;
    a.foo();
    b.foo();

    A<double> c, d;
    c.foo();
    d.foo();

    A<string> e, f;
    e.foo();
    f.foo();

    return 0;
}
```

8.3.3.2 类模板的递归实例化

类模板的类型实参可以是任何类型，只要该类型能够提供模板所需要的功能即可。

类模板自身的实例化类亦可实例化其自身，谓之递归实例化。通过这种方法可以很容易地构建那些在空间上具有递归特征的数据结构。

```cpp
// recursion.cpp

// 类模板的递归实例化

#include <iostream>

using namespace std;

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

    T const& operator[](size_t i) const {
        return const_cast<Array&>(*this)[i];
    }

private:
    T arr[3];
};

template<typename T>
ostream& operator<<(ostream& os, Array<T> const& array) {
    for (size_t i = 0; i < 3; ++i)
        os << '(' << array[i] << ')';

    return os;
}

int main(void) {
    Array<int> ai;
    ai[0] = 100;
    ai[1] = 200;
    ai[2] = 300;
    cout << ai << endl;

    Array<string> as;
    as[0] = "北京";
    as[1] = "上海";
    as[2] = "广州";
    cout << as << endl;

    Array<Array<int>> aai;
    for (size_t i = 0; i < 3; ++i)
        for (size_t j = 0; j < 3; ++j)
            aai[i][j] = (i+1)*10+j+1;
    cout << aai << endl;
```

```
50
51      return 0;
52  }
```

- `Array<int>`：一维数组
- `Array<Array<int>>`：二维数组
- `Array<Array<Array<int>>>`：三维数组

8.3.4 类模板的特化

通过特化类模板，可以优化针对某种特定类型的实现，或者克服某种特定类型在实例化类模板时所表现出的不足。

```
1   // error.cpp
2
3   // 特定类型在实例化类模板时所表现出的不足
4
5   #include <iostream>
6
7   using namespace std;
8
9   // 定义类模板
10
11  template<class T>
12  class Comparator {
13  public:
14      Comparator(T x, T y) : x(x), y(y) {}
15
16      T min(void) const {
17          return x < y ? x : y;
18      }
19
20      T max(void) const {
21          return x < y ? y : x;
22      }
23
24  private:
25      T x, y;
26  };
27
28  // 使用类模板
29
30  int main (void) {
31      cout << "Enter two integer: " << flush;
32      int nx, ny;
33      cin >> nx >> ny;
34      Comparator<int> cn(nx, ny);
35      cout << "min=" << cn.min() << endl;
36      cout << "max=" << cn.max() << endl;
37
38      cout << "Enter two double: " << flush;
39      double dx, dy;
40      cin >> dx >> dy;
41      Comparator<double> cd(dx, dy);
```

```
    cout << "min=" << cd.min() << endl;
    cout << "max=" << cd.max() << endl;

    cout << "Enter two string: " << flush;
    string sx, sy;
    cin >> sx >> sy;
    Comparator<string> cs(sx, sy);
    cout << "min=" << cs.min() << endl;
    cout << "max=" << cs.max() << endl;

    cout << "Enter two string: " << flush;
    char cx[256], cy[256];
    cin >> cx >> cy;
    Comparator<char const*> cc(cx, cy);
    cout << "min=" << cc.min() << endl;
    cout << "max=" << cc.max() << endl;

    return 0;
}
```

8.3.4.1 全类特化

特化一个类模板可以特化该类模板的所有成员函数，相当于重新写一个针对某种特定类型的具体类。对一个类模板做全类特化：

- 需要特化该类模板的所有成员函数
- 必须在起始处声明一个`template<>`，接下来声明用来特化类模板的具体类型。该类型被用作类模板的实参，且必须在类模板名后面显式指定

```
template<>
class 类模板名<特化类型> {
    针对特化类型的具体类
};
```

- 每个成员函数都必须被重新定义为普通成员函数。通用版本中的每个模板参数都被具体的特化类型取代
- 特化版本的实现可以和通用版本的实现完全不同

```
// special.cpp

// 类模板的全类特化

#include <string.h>

#include <iostream>

using namespace std;

// 通用版本

template<class T>
class Comparator {
public:
    Comparator(T x, T y) : x(x), y(y) {}
```

```
17
18      T min(void) const {
19          return x < y ? x : y;
20      }
21
22      T max(void) const {
23          return x < y ? y : x;
24      }
25
26  private:
27      T x, y;
28  };
29
30  // 针对char const*类型的特化版本
31
32  template<>
33  class Comparator<char const*> {
34  public:
35      Comparator(char const* x, char const* y) : x(x), y(y) {}
36
37      char const* min(void) const {
38          return strcmp(x, y) < 0 ? x : y;
39      }
40
41      char const* max(void) const {
42          return strcmp(x, y) < 0 ? y : x;
43      }
44
45  private:
46      char const *x, *y;
47  };
48
49  // 使用类模板
50
51  int main (void) {
52      cout << "Enter two integer: " << flush;
53      int nx, ny;
54      cin >> nx >> ny;
55      Comparator<int> cn(nx, ny);
56      cout << "min=" << cn.min() << endl;
57      cout << "max=" << cn.max() << endl;
58
59      cout << "Enter two double: " << flush;
60      double dx, dy;
61      cin >> dx >> dy;
62      Comparator<double> cd(dx, dy);
63      cout << "min=" << cd.min() << endl;
64      cout << "max=" << cd.max() << endl;
65
66      cout << "Enter two string: " << flush;
67      string sx, sy;
68      cin >> sx >> sy;
69      Comparator<string> cs(sx, sy);
70      cout << "min=" << cs.min() << endl;
71      cout << "max=" << cs.max() << endl;
72
```

```
    cout << "Enter two string: " << flush;
    char cx[256], cy[256];
    cin >> cx >> cy;
    Comparator<char const*> cc(cx, cy);
    cout << "min=" << cc.min() << endl;
    cout << "max=" << cc.max() << endl;

    return 0;
}
```

8.3.4.2 成员特化

类模板除了可以整体进行特化以外，也可以只针对部分成员函数进行特化。对一个类模板做成员特化，特化实现与通用实现共享同一份声明，因此特化版本与通用版本的函数原型，除模板参数以外，必须严格一致。

```
// member.cpp

// 类模板的成员特化

#include <string.h>

#include <iostream>

using namespace std;

// 通用版本

template<class T>
class Comparator {
public:
    Comparator(T x, T y) : x(x), y(y) {}

    T min(void) const {
        return x < y ? x : y;
    }

    T max(void) const {
        return x < y ? y : x;
    }

private:
    T x, y;
};

// 针对char const*类型的特化版本

template<>
char const* Comparator<char const*>::min(void) const {
    return strcmp(x, y) < 0 ? x : y;
}

template<>
char const* Comparator<char const*>::max(void) const {
    return strcmp(x, y) < 0 ? y : x;
```

```
}

// 使用类模板

int main (void) {
    cout << "Enter two integer: " << flush;
    int nx, ny;
    cin >> nx >> ny;
    Comparator<int> cn(nx, ny);
    cout << "min=" << cn.min() << endl;
    cout << "max=" << cn.max() << endl;

    cout << "Enter two double: " << flush;
    double dx, dy;
    cin >> dx >> dy;
    Comparator<double> cd(dx, dy);
    cout << "min=" << cd.min() << endl;
    cout << "max=" << cd.max() << endl;

    cout << "Enter two string: " << flush;
    string sx, sy;
    cin >> sx >> sy;
    Comparator<string> cs(sx, sy);
    cout << "min=" << cs.min() << endl;
    cout << "max=" << cs.max() << endl;

    cout << "Enter two string: " << flush;
    char cx[256], cy[256];
    cin >> cx >> cy;
    Comparator<char const*> cc(cx, cy);
    cout << "min=" << cc.min() << endl;
    cout << "max=" << cc.max() << endl;

    return 0;
}
```

8.3.5 类模板的局部特化

8.3.5.1 对部分模板参数自行指定

类模板可以被局部特化，即一方面为类模板指定特定的实现，另一方面又允许用户对部分模板参数自行指定。

```
// partial.cpp

// 类模板的局部特化

#include <iostream>

using namespace std;

// 通用版本

template<typename T1, typename T2>
class A {
```

```
public:
    static void foo(void) {
        cout << "A<T1,T2>" << endl;
    }
};

// 针对第二个类型参数取short的局部特化版本

template<typename T>
class A<T, short> {
public:
    static void foo(void) {
        cout << "A<T,short>" << endl;
    }
};

// 针对两个类型参数取相同类型的局部特化版本

template<typename T>
class A<T, T> {
public:
    static void foo(void) {
        cout << "A<T,T>" << endl;
    }
};

// 针对两个类型参数取指针的局部特化版本

template<typename T1, typename T2>
class A<T1*, T2*> {
public:
    static void foo(void) {
        cout << "A<T1*,T2*>" << endl;
    }
};

// 针对两个类型参数取引用的局部特化版本

template<typename T1, typename T2>
class A<T1&, T2&> {
public:
    static void foo(void) {
        cout << "A<T1&,T2&>" << endl;
    }
};

int main(void) {
    A<int, char>::foo();
    A<int, short>::foo();
    A<int, int>::foo();
    A<int*, char*>::foo();
    A<int&, char&>::foo();

    return 0;
}
```

8.3.5.2 同等程度地特化匹配导致歧义

如果多个局部特化同等程度地匹配某个声明，那么该声明将因二义性而导致歧义错误：

```
// 针对两个类型参数取相同类型的局部特化版本

template<typename T>
class A<T, T> {
public:
    static void foo(void) {
        cout << "A<T,T>" << endl;
    }
};

// 针对两个类型参数取指针的局部特化版本

template<typename T1, typename T2>
class A<T1*, T2*> {
public:
    static void foo(void) {
        cout << "A<T1*,T2*>" << endl;
    }
};
```

```
A<int*, int*>::foo(); // 编译错误，在选择A<T,T>和A<T1*,T2*>时发生歧义
```

除非有更好的匹配：

```
// 针对两个类型参数取相同类型的指针的局部特化版本

template<typename T>
class A<T*, T*> {
public:
    static void foo(void) {
        cout << "A<T*,T*>" << endl;
    }
};
```

8.3.6 类模板参数的缺省值

8.3.6.1 类模板可以带有缺省参数

类模板的模板参数可以带有缺省值，即缺省模板实参。实例化类模板时，如果提供了模板实参则用所提供的模板实参传递给相应的模板形参，如果没有提供模板实参则相应的模板形参取缺省值。

```
// clsdefargs.cpp

// 类模板参数的缺省值

#include <iostream>
#include <typeinfo>

using namespace std;
```

```
9
10 template<typename A = char, typename B = short, typename C = int>
11 class X {
12 public:
13     static void foo(void) {
14         cout << typeid(A).name() << ' ' <<
15                 typeid(B).name() << ' ' <<
16                 typeid(C).name() << endl;
17     }
18 };
19
20 int main(void) {
21     X<long, float, double>::foo();
22     X<long, float>::foo();
23     X<long>::foo();
24     X<>::foo();
25
26     return 0;
27 }
```

如果类模板的某个模板参数带有缺省值，那么它后面的所有模板参数必须都带有缺省值。例如：

```
1 template<typename A = char, typename B = short, typename C = int> class X {
... };
```

```
1 template<typename A, typename B = short, typename C = int> class X { ... };
```

```
1 template<typename A, typename B, typename C = int> class X { ... };
```

以下情况都是不允许出现的：

- 前面的参数带有缺省值，后面的参数不带缺省值

```
1 template<typename A = char, typename B = short, typename C> class X { ...
}; // 错误
```

- 中间的参数带有缺省值，两边的参数不带缺省值

```
1 template<typename A, typename B = short, typename C> class X { ... }; //
错误
```

- 两边的参数带有缺省值，中间的参数不带缺省值

```
1 template<typename A = char, typename B, typename C = int> class X { ...
}; // 错误
```

8.3.6.2 后面的参数可以引用前面参数

类模板后面参数的缺省值可以引用前面参数的值。例如：

```
1 template<typename A, typename B = A*> class X { ... };
```

`X<int>`即等价于`X<int, int*>`。

8.4 非类型模板参数

8.4.1 普通数值作为模板参数

模板的参数并不局限于类型参数，普通数值也可以作为模板的参数，前面不要写typename，而要写具体类型。

```
// clsargs.cpp

// 类模板的非类型模板参数

#include <iostream>

using namespace std;

template<typename T, size_t C = 3>
class Array {
public:
    T& operator[](size_t i) {
        return arr[i];
    }

    T const& operator[](size_t i) const {
        return const_cast<Array&>(*this)[i];
    }

private:
    T arr[C];
};

template<typename T, size_t S>
ostream& operator<<(ostream& os, Array<T, S> const& array) {
    for (size_t i = 0; i < S; ++i)
        os << '(' << array[i] << ')';

    return os;
}

int main(void) {
    Array<int, 3> ai;
    ai[0] = 100;
    ai[1] = 200;
    ai[2] = 300;
    cout << ai << endl;

    Array<string> as;
    as[0] = "北京";
    as[1] = "上海";
    as[2] = "广州";
    cout << as << endl;

    size_t const rows = 4, cols = 5;
    Array<Array<int, cols>, rows> aai;
```

```
    for (size_t i = 0; i < rows; ++i)
        for (size_t j = 0; j < cols; ++j)
            aai[i][j] = (i+1)*10+j+1;
    cout << aai << endl;

    return 0;
}
```

类模板的非类型参数与类型参数一样，也可以带有缺省值。

与类模板一样，函数模板也可以带有非类型参数。

```
// funcargs.cpp

// 函数模板的非类型模板参数

#include <iostream>

using namespace std;

template<typename T, size_t TIMES> // 将重复次数定义为模板参数
void repeat(T const& t) {
    for (size_t i = 0; i < TIMES; ++i)
        cout << t << ' ';
    cout << endl;
}

template<typename IT, typename OP>
void for_each(IT begin, IT end, OP op) {
    while (begin != end)
        op(*begin++);
}

int main(void) {
    repeat<int, 5>(0);
    repeat<char, 5>('A');
    repeat<string, 5>("ABC");

    int ai[] = {1, 2, 3, 4, 5};
    for_each(ai, ai + 5, (void (*)(int const&))repeat<int, 5>);
    // 函数模板的实例化相当于一组重载函数的集合，
    // 只有在转换为具体类型后，才能充当模板实参

    return 0;
}
```

函数模板的非类型参数与类型参数一样，也不可以带有缺省值。

8.4.2 非类型模板参数的限制

- 非类型模板参数只能是常量、常量表达式，以及带有常属性（const）的变量，但不能同时具有挥发性（volatile）
- 非类型模板参数只能是整数（signed/unsigned char/short/int/long），枚举和指向外部变量的指针

- 非类型模板参数不能是浮点类型（float/double）或类类型
- 非类型模板参数不能是指向内部变量的指针，如字符串字面值、全局指针等

```cpp
// limit.cpp

// 非类型模板参数的限制

#include <iostream>

using namespace std;

template<char c>
void foo(void) {
    cout << c << endl;
}

// template<double d>
// void bar(void) {
//     cout << d << endl;
// }
// 编译错误，模板的非类型参数不能是浮点类型

// template<string s>
// void hum(void) {
//     cout << s << endl;
// }
// 编译错误，模板的非类型参数不能是类类型

template<char const* cp>
void fun(void) {
    cout << cp << endl;
}

char const* cp = "北京";
static char sca[] = "天津";
char ca[] = "上海";

int main (void) {
    foo<'A'>();

    // fun<"重庆">(); // 编译错误，模板的非类型实参不能是字符串字面值
    // fun<cp>(); // 编译错误，模板的非类型实参不能是全局指针
    fun<sca>();
    fun<ca>();

    return 0;
}
```

8.5 模板技巧

8.5.1 typename

8.5.1.1 声明模板参数

声明模板参数：template<typename T> ...，只有在这种语境下，typename关键字才可以和class关键字互换

- class关键字
  - 声明类：class A { ... };
  - 声明模板参数：template<class T> ...
- typename关键字
  - 声明模板参数：template<typename T> ...
  - 解决嵌套依赖：typename T::A a;
- 无论是声明模板参数还是解决嵌套依赖，都不能使用struct关键字

8.5.1.2 解决嵌套依赖

```
class X {
public:
    // A是X的嵌套类型
    class A {
        ...
    };
};
```

```
template<typename T> // T<-X
void foo(void) {
    ... T::A a ...  // 编译错误
}
```

```
foo<X>(); // X->T
```

在第一次编译模板代码时，模板参数的具体类型尚不明确，编译器会把依赖于模板参数的嵌套类型理解为某个类的静态成员变量。因此当它看到代码中使用这样的标识符声明其它变量时，会报告错误，这就叫做嵌套依赖。

```
template<typename T> // T<-X
void foo(void) {
    ... typename T::A a ... // 编译通过
}
```

typename关键字旨在告诉编译器，所引用的标识符是个类型名，可以声明变量，具体类型等到实例化时再定。

```
// typename.cpp

// 借助typename关键字解决嵌套依赖

```

```cpp
#include <iostream>
#include <typeinfo>

using namespace std;

class X {
public:
    // A是X的嵌套类型
    class A {
    };
};

template<typename T> // T<-X
void foo(void) {
    // T::A a; // 编译错误
    typename T::A a; // 编译通过
    cout << typeid(a).name() << endl;
}

int main(void) {
    foo<X>(); // X->T

    return 0;
}
```

8.5.2 template

8.5.2.1 声明模板

template关键字的典型用法是声明模板

- 函数模板

```cpp
template<typename T> void foo(void) { ... T ... }
```

- 类模板

```cpp
template<typename T> class X { ... T ...};
```

8.5.2.2 依赖模板参数的模板型成员访问

```cpp
class A {
public:
    template<typename T>
    void foo(void) const { ... T ... }
};
```

```cpp
template<typename T>
void bar(T const& r, T const* p) {
    r.foo<int>(); // 编译错误
    p->foo<double>(); // 编译错误
}
```

```
A a, *p = &a;
bar(a, p);
```

在模板代码中，通过依赖于模板参数的对象、引用或指针，访问其带有模板特性的成员，编译器常常因为无法正确理解模板参数列表的左右尖括号，而报告编译错误。在模板名前面加上template关键字，意在告诉编译器其后的名称是一个模板，编译器就可以正确理解“<>”了。

```
template<typename T>
void bar(T const& r, T const* p) {
    r.template foo<int>();
    p->template foo<double>();
}
```

template关键字旨在告诉编译器，所引用的标识符是个模板，其后尖括号括起的是传递给该模板的模板参数。

```
// template.cpp

// 依赖模板参数的模板型成员访问

#include <iostream>
#include <typeinfo>

using namespace std;

class A {
public:
    template<typename T>
    void foo(void) const {
        cout << typeid(T).name() << endl;
    }
};

template<typename T>
void bar(T const& r, T const* p) {
    // r.foo<int>(); // 编译错误
    // p->foo<double>(); // 编译错误
    r.template foo<int>();
    p->template foo<double>();
}

int main (void) {
    A a, *p = &a;

    bar(a, p);

    return 0;
}
```

8.5.3 子模板访问基模板

```
template<typename T>
class Base {
public:
    class Nested {};

    void foo(void) {
        ++t;
    }

    void exit(int status) const {
        cout << "Base::exit(" << status << ")" << endl;
    }

    T t;
};
```

```
template<typename T>
class Derived : public Base<T> {
public:
    void bar(void) {
        Nested nested; // 编译错误
        foo(); // 编译错误
        cout << t << endl; // 编译错误
        exit(0); // 所调用并非基类的exit成员函数
    }
};
```

在子类模板中直接访问那些依赖于模板参数的基类模板的成员，编译器在第一次编译时，通常会因为基类类型不明确而只在子类和全局作用域中搜索所引用的符号。在子类模板中可以通过作用域限定符，或者显式使用this指针，迫使编译器到基类作用域中搜索所引用的符号。

```
// this.cpp

// 在子模板中访问基模板

#include <iostream>

using namespace std;

template<typename T>
class Base {
public:
    class Nested {};

    void foo(void) {
        ++t;
    }

    void exit(int status) const {
        cout << "Base::exit(" << status << ")" << endl;
    }
```

```
21
22      T t;
23  };
24
25  template<typename T>
26  class Derived : public Base<T> {
27  public:
28      void bar(void) {
29          // Nested nested; // 编译错误
30          // foo(); // 编译错误
31          // cout << t << endl; // 编译错误
32          // exit(0); // 所调用并非基类的exit成员函数
33
34          // 在子模板中访问基模板的成员，需要使用作用域限定符
35          typename Base<T>::Nested nested;
36          Base<T>::foo();
37          cout << Base<T>::t << endl;
38
39          // 也可以借助this指针访问基模板的成员函数和成员变量
40          this->foo();
41          cout << this->t << endl;
42
43          // 调用基模板的exit函数
44          Base<T>::exit(0);
45          this->exit(0);
46      }
47  };
48
49  int main(void) {
50      Derived<int> d;
51
52      d.bar();
53
54      return 0;
55  }
```

8.5.4 模板型模板成员

8.5.4.1 模板型成员变量

类模板的成员变量，如果其类型又源自一个类模板的实例化类，那么它就是一个模板型模板成员变量。

```
1   // membvar.cpp
2
3   // 类模板的模板型成员变量
4
5   #include <iostream>
6   #include <typeinfo>
7
8   using namespace std;
9
10  template<typename T>
11  class A {
12  public:
13      void foo(void) const {
```

```
        cout << typeid(T).name() << endl;
    }
};

template<typename T>
class B {
public:
    A<T> a; // 模板型成员变量
};

int main(void) {
    B<int> b;

    b.a.foo();

    return 0;
}
```

8.5.4.2 模板型成员函数

类模板的成员函数，如果除了类模板的模板参数外，还需要其它模板参数，那么它就是一个模板型模板成员函数。

```
// membfun.cpp

// 类模板的模板型成员函数

#include <iostream>
#include <typeinfo>

using namespace std;

template<typename U>
class A {
public:
    template<typename V>
    void foo(void) const {
        cout << typeid(U).name() << ' ' << typeid(V).name() << endl;
    } // 模板型成员函数

    template<typename V> void bar(void) const; // 模板型成员函数
};

template<typename U>
    template<typename V>
void A<U>::bar(void) const {
    cout << typeid(U).name() << ' ' << typeid(V).name() << endl;
}

int main(void) {
    A<char> a;

    a.foo<short>();
    a.bar<int>();

```

```
    return 0;
}
```

8.5.4.3 模板型成员类型

类模板的成员类型，如果除了类模板的模板参数外，还需要其它模板参数，那么它就是一个模板型模板成员类型。

```
// membcls.cpp

// 类模板的模板型成员类型

#include <iostream>
#include <typeinfo>

using namespace std;

template<typename U>
class A {
public:
    template<typename V>
    class B {
    public:
        void foo(void) const {
            cout << typeid(U).name() << ' ' << typeid(V).name() << endl;
        }
    }; // 模板型成员类型

    template<typename V> class C; // 模板型成员类型
};

template<typename U>
    template<typename V>
class A<U>::C {
public:
    void foo(void) const {
        cout << typeid(U).name() << ' ' << typeid(V).name() << endl;
    }
};

int main(void) {
    A<char>::B<short> b;
    b.foo();

    A<int>::C<long> c;
    c.foo();

    return 0;
}
```

8.5.5 模板型模板参数

函数模板和类模板的模板型模板参数：

- 函数模板和类模板的模板参数本身又可以是个模板，但注意该模板参数的声明，不能使用typename关键字，而要用template<...> class结构
- 可以使用任何类型，甚至其它模板参数，实例化模板型模板参数
- 模板型模板参数的模板参数名，若未被引用，则可以省略
- 模板型模板参数的模板参数也可以带有缺省值

```cpp
// tmplargs.cpp

// 函数模板和类模板的模板型模板参数

#include <iostream>
#include <typeinfo>

using namespace std;

template<typename T>
class A {
public:
    void foo(void) const {
        cout << typeid(T).name() << endl;
    }
};

// 带有模板型模板参数的函数模板

template<typename U, template<typename> class V = A>
void foo(void) {
    V<short> v;
    v.foo();
}

// 带有模板型模板参数的类模板

template<typename U, template<typename> class V = A>
class B {
public:
    void foo(void) const {
        V<U> v;
        v.foo();
    }
};

int main(void) {
    foo<char>();

    B<int, A> b;
    b.foo();

    return 0;
```

```
}
```

8.5.5.1 函数模板和类模板的模板参数可以是模板

函数模板和类模板的模板参数本身又可以是个模板。

```
template<typename U, template<typename> class V> void foo(void) { ... }
```

```
template<typename U, template<typename> class V> class B { ... };
```

8.5.5.2 模板型模板参数的实例化

可以使用任何类型，甚至其它模板参数，实例化模板型模板参数。

```
template<typename T> class A { ... };
```

```
template<typename U, template<typename> class V> void foo(void) { ... V<short> ... }
```

```
template<typename U, template<typename> class V> class B { ... V<U> ... };
```

```
foo<char, A>();
```

```
B<int, A> b;
```

8.5.5.3 模板型模板参数的缺省参数

模板型模板参数的模板参数也可以带有缺省值。

```
template<typename T> class A { ... };
```

```
template<typename U, template<typename> class V = A> void foo(void) { ... V<short> ... }
```

```
template<typename U, template<typename> class V = A> class B { ... V<U> ... };
```

```
foo<char>();
```

```
B<int> b;
```

8.5.6 零初始化

```
// init.cpp

// 零初始化

#include <iostream>
```

```cpp

using namespace std;

template<typename T>
void foo(void) {
    T var; // 若T为基本类型，则var的值未定义
    cout << var << endl;
}

template<typename T>
class A {
public:
    T var; // 若T为基本类型，则var的值未定义
};

template<typename T>
void bar(void) {
    T var = T(); // 无论T为类类型还是基本类型，var都以缺省方式初始化
    cout << var << endl;
}

template<typename T>
class B {
public:
    B(void) : var() {} // 无论T为类类型还是基本类型，var都以缺省方式初始化

    T var;
};

int main(void) {
    foo<int>();
    A<int> a;
    cout << a.var << endl;

    bar<int>();
    B<int> b;
    cout << b.var << endl;

    return 0;
}
```

8.5.6.1 未初始化的基本类型

基本类型的局部变量，如果没有被显式初始化，其初值通常是不确定的：

```cpp
void foo(void) {
    int var; // 初值不确定
    ...
}
```

类类型的局部变量，如果没有被显式初始化，其初值通常由该类的缺省构造函数决定：

```
void foo(void) {
    Student var; // 初值由缺省构造函数决定
}
```

这样就会在包含模板的代码中，表现出某种程度的不一致：

```
template<typename T> void foo(void) {
    T var; // 当T是基本类型时，初值不确定，而当T是类类型时，初值由该类的缺省构造函数决定
    ...
}
```

```
foo<int>(); // int->T
```

```
foo<Student>(); // Student->T
```

8.5.6.2 显式缺省构造

如果希望模板中所有参数化类型的变量，无论是类类型还是基本类型，都能以缺省方式被初始化，就必须对其进行显式地缺省构造：

```
template<typename T> void foo(void) {
    T var = T(); // 显式缺省构造
    ...
}
```

当用int实例化该函数模板时：

```
void foo<int>(void) {
    int var = int(); // 用int类型的零值初始化
    ...
}
```

当用Student实例化该函数模板时：

```
void foo<Student>(void) {
    Student var = Student(); // 用Student类的缺省构造函数初始化
    ...
}
```

对类型一致性的追求，始终是模板设计的核心准则之一。

8.5.6.3 在初始化表中显式初始化

对于类模板，可在其缺省构造函数的初始化表中，显式初始化各个成员变量，无论是类类型的还是基本类型的：

```
template<typename T> class A {
public:
    A(void) : var() {}
    ...
private:
    T var;
    ...
};
```

当用int实例化该类模板时：

```
class A<int> {
public:
    A(void) : var() {} // 用int类型的零值初始化
    ...
private:
    int var;
    ...
};
```

当用Student实例化该类模板时：

```
class A<Student> {
public:
    A(void) : var() {} // 用Student类的缺省构造函数初始化
    ...
private:
    Student var;
    ...
};
```

类型一致性的设计准则，同样得到了完美的体现。

8.5.7 字符串型函数模板实参

将字符串传递给带有引用型参数和非引用型参数的函数模板，将导致完全不同的结果。函数模板的隐式推断会把引用型参数推断为对整个字符数组的引用，而非引用型参数则仅仅以字符指针的形式接收字符串的首地址。

对同一个函数模板，同时声明引用和非引用两个版本可能会引发歧义。而只使用非引用的版本又会导致无谓的拷贝。如果不想强迫用户通过显式类型转换或者显式实例化，完全用C++的string取代C风格的字符串，那么为非引用版本专门提供一个针对字符指针的重载总是有效的。

```
// string.cpp

// 字符串型函数模板实参

#include <string.h>

#include <iostream>
#include <typeinfo>

using namespace std;
```

```cpp

// 接收引用型参数的函数模板

template<typename T>
void foo(T const& t) {
    // t是实参字符数组的引用
    cout << typeid (t).name () << ": " << t << endl;
}

// 接收非引用型参数的函数模板

template<typename T>
void bar (T t) {
    // t是实参字符指针的副本
    cout << typeid (t).name () << ": " << t << endl;
}

// 通用版本，借助引用避免拷贝

template<typename T>
T const& min(T const& x, T const& y) {
    return x < y ? x : y;
}

// 特殊版本，仅针对字符指针指向的空结尾字符串，即C风
// 格字符串。既能避免编译错误，又能以正确方式实现比较

const char* min(const char* x, const char* y) {
    return strcmp(x, y) < 0 ? x : y;
}

int main (void) {
    foo("string is a character array");
    bar("string is a character pointer");

    cout << ::min("hello", "world") << endl;
    cout << ::min("hello", "tarena") << endl;

    return 0;
}
```

8.5.8 类模板的虚成员函数

8.5.8.1 类模板可以定义虚函数

类模板的普通成员函数可以是虚函数，即可以为类模板定义虚成员函数。和普通类的虚成员函数一样，类模板的虚成员函数亦可表现出多态性：

```cpp
template<typename PEN> class Rect {
    ...
    virtual void draw(PEN const& pen) { ... }
    ...
};
```

```
template<typename PEN> class RoundRect : public Rect<PEN> {
    ...
    void draw(PEN const& pen) { ... }
};
```

```
Rect<SolidPen>* rect = new RoundRect<SolidPen>(...);
rect->draw(SolidPen(...));
```

8.5.8.2 类模板的虚函数不能是模板

无论是类还是类模板，其虚成员函数都不能是模板函数。基于虚函数的多态机制，需要一个名为虚函数表的函数指针数组。该数组在类被编译或类模板被实例化的过程中产生，而此时那些模板形式的成员函数尚未被实例化，其入口地址和重载版本的个数，要等到编译器处理完对该函数的所有调用以后才能确定。成员函数模板的延迟编译阻碍了虚函数表的静态构建。

```
template<typename PEN> class Rect {
    ....
    template<typename BRUSH>
    virtual void draw(PEN const& pen, BRUSH const& brush) { ... } // 编译错误
};
```

8.5.8.3 实例化类模板时所用的类型实参会影响虚函数覆盖的条件

```
template<...> class A {
    virtual void foo(T arg) { ... }
}
```

```
template<...> class B : public A<...> {
    void foo(T arg) const { ... }
}
```

只有当类模板B中的T，与类模板A中的T，取相同类型时，B中的foo函数才是虚函数，并对A中的foo函数构成覆盖。

```
// virtual.cpp

// 类模板的虚成员函数

#include <iostream>

using namespace std;

template<typename U>
class A {
public:
    // 类模板的所有非模板型成员函数（包括析构函数）都可以是虚函数

    virtual ~A(void) {}

    virtual void foo(U const& u) const {
        cout << "A::foo(" << u << ") invoked" << endl;
    }
```

```
19
20      // 类模板的所有模板型成员函数（包括析构函数）都不能是虚函数
21
22      // template<typename V>
23      // virtual void bar(V const& v) {
24      //     cout << "A::bar(" << v << ") invoked" << endl;
25      // }
26  };
27
28  template<typename U>
29  class B : public A<U> {
30  public:
31      void foo(U const& u) const {
32          cout << "B::foo(" << u << ") invoked" << endl;
33      }
34  };
35
36  template<typename U, typename V>
37  class C : public A<V> {
38  public:
39      void foo(U const& u) const {
40          cout << "C::foo(" << u << ") invoked" << endl;
41      }
42  };
43
44  int main (void) {
45      A<int>* a = new B<int>;
46      a->foo(123); // 多态
47      delete a;
48
49      a = new C<short, int>;
50      a->foo(456); // 非多态
51      delete a;
52
53      return 0;
54  }
```

8.6 模板实战

8.6.1 模板的编译模型

8.6.1.1 单一模型

将模板的声明、定义和实例化放在单一的编译单元中，无论编译还是链接，其结果总是对的。

```
1  // comparator1.cpp
2
3  // 单一模型
4
5  #include <string.h>
6
7  #include <iostream>
8
9  using namespace std;
10
```

```cpp
// 在声明模板的同时给出定义

template<class T>
T min(T x, T y) {
    return x < y ? x : y;
}

template<typename T>
T max(T x, T y) {
    return x < y ? y : x;
}

char const* min(char const* x, char const* y) {
    return strcmp(x, y) < 0 ? x : y;
}

char const* max(char const* x, char const* y) {
    return strcmp(x, y) < 0 ? y : x;
}

template<class T>
class Comparator {
public:
    Comparator(T x, T y) : x(x), y(y) {}

    T min(void) const {
        return x < y ? x : y;
    }

    T max(void) const {
        return x < y ? y : x;
    }

private:
    T x, y;
};

template<>
class Comparator<char const*> {
public:
    Comparator(char const* x, char const* y) : x(x), y(y) {}

    char const* min(void) const {
        return strcmp(x, y) < 0 ? x : y;
    }

    char const* max(void) const {
        return strcmp(x, y) < 0 ? y : x;
    }

private:
    char const *x, *y;
};

// 根据特定的模板实参对模板进行实例化

```

```cpp
int main(void) {
    int nx = 123, ny = 456;
    double dx = 1.23, dy = 4.56;
    string sx = "hello", sy = "world";
    char const *cx = "hello", *cy = "world";

    cout << ::min(nx, ny) << ' ' << ::max(nx, ny) << endl;
    cout << ::min(dx, dy) << ' ' << ::max(dx, dy) << endl;
    cout << ::min(sx, sy) << ' ' << ::max(sx, sy) << endl;
    cout << ::min(cx, cy) << ' ' << ::max(cx, cy) << endl;

    Comparator<int> cn(nx, ny);
    Comparator<double> cd(dx, dy);
    Comparator<string> cs(sx, sy);
    Comparator<char const*> cc(cx, cy);

    cout << cn.min() << ' ' << cn.max() << endl;
    cout << cd.min() << ' ' << cd.max() << endl;
    cout << cs.min() << ' ' << cs.max() << endl;
    cout << cc.min() << ' ' << cc.max() << endl;

    return 0;
}
```

8.6.1.2 分离模型

大多数C/C++程序员会这样组织他们的代码：

- 把公共头文件包含和宏定义，放在头文件（.h）中
- 把类、结构、联合、枚举、类型别名等，放在头文件（.h）中
- 把全局变量和全局函数的外部声明，放在头文件（.h）中
- 把全局变量、全局函数和成员函数的定义，放在源文件（.c或.cpp）中

这样做既能保证所有的类型信息，在整个项目被编译的过程中都可见，避免编译器报告“未声明”错误，同时也不会因为变量和函数定义的重复出现，导致链接器报告“重定义”错误。

```cpp
// comparator2.h

// 分离模型

#pragma once

// 在头文件中仅给出模板的声明

template<class T>
T min(T x, T y);

template<typename T>
T max(T x, T y);

char const* min(char const* x, char const* y);

char const* max(char const* x, char const* y);

```

```
template<class T>
class Comparator {
public:
    Comparator(T x, T y);

    T min(void) const;

    T max(void) const;

private:
    T x, y;
};
```

模板的定义只是一种规范性的描述，并非真正意义上的类型定义。当编译器看到模板定义时，仅做一般性的语法检查，同时生成一份模板的内部表示，并不生成指令代码。只有当编译器看到模板被实例化为具体函数或具体类时，才会用具体的模板实参，结合之前生成的内部表示，产生二进制指令代码。这个过程被称为模板的后期编译。

```
// comparator2.cpp

// 分离模型

#include <string.h>

#include "comparator2.h"

// 在源文件中给出模板的定义

// 在编译此文件时，编译器看不到任何模板实参，因此不会对所定义的模板进行实例化

template<class T>
T min(T x, T y) {
    return x < y ? x : y;
}

template<typename T>
T max(T x, T y) {
    return x < y ? y : x;
}

char const* min(char const* x, char const* y) {
    return strcmp(x, y) < 0 ? x : y;
}

char const* max(char const* x, char const* y) {
    return strcmp(x, y) < 0 ? y : x;
}

template<class T>
Comparator<T>::Comparator(T x, T y) : x(x), y(y) {}

template<class T>
T Comparator<T>::min(void) const {
    return x < y ? x : y;
}
```

```
38
39  template<class T>
40  T Comparator<T>::max(void) const {
41      return x < y ? y : x;
42  }
43
44  template<>
45  char const* Comparator<char const*>::min(void) const {
46      return strcmp(x, y) < 0 ? x : y;
47  }
48
49  template<>
50  char const* Comparator<char const*>::max(void) const {
51      return strcmp(x, y) < 0 ? y : x;
52  }
```

每个C或C++源文件都是被单独编译的。编译器在编译模板定义文件时所生成的模板内部表示，此刻早已荡然无存。因此所有基于模板实例的类和函数，编译器只能假设它们被定义在其它模块中，并产生一个指向该（不存在的）定义的引用，期待链接器能在日后解决此问题。但事实上，由于模板的内部表示并没有真正化身为可执行的指令代码，链接器最终报告“未定义的引用”错误。

```
1   // main2.cpp
2
3   // 分离模型
4
5   #include <iostream>
6
7   #include "comparator2.h"
8
9   using namespace std;
10
11  // 根据特定的模板实参对模板进行实例化
12
13  // 编译此文件时，编译器虽然可以看到传递给模板的实参，但却看不到模板的定
14  // 义，因此也不会对模板进实例化。模板始终得不到实例化，最终导致链接错误
15
16  int main(void) {
17      int nx = 123, ny = 456;
18      double dx = 1.23, dy = 4.56;
19      string sx = "hello", sy = "world";
20      char const *cx = "hello", *cy = "world";
21
22      cout << ::min(nx, ny) << ' ' << ::max(nx, ny) << endl;
23      cout << ::min(dx, dy) << ' ' << ::max(dx, dy) << endl;
24      cout << ::min(sx, sy) << ' ' << ::max(sx, sy) << endl;
25      cout << ::min(cx, cy) << ' ' << ::max(cx, cy) << endl;
26
27      Comparator<int> cn(nx, ny);
28      Comparator<double> cd(dx, dy);
29      Comparator<string> cs(sx, sy);
30      Comparator<char const*> cc(cx, cy);
31
32      cout << cn.min() << ' ' << cn.max() << endl;
33      cout << cd.min() << ' ' << cd.max() << endl;
34      cout << cs.min() << ' ' << cs.max() << endl;
```

```
    cout << cc.min() << ' ' << cc.max() << endl;

    return 0;
}
```

由此可见，分离模型显然并不适合于包含模板的项目代码。

8.6.1.3 包含模型

把模板定义文件包含在模板声明文件的内部（通常位于声明之后），即让模板的声明和定义同处于一个头文件中。

```
// comparator3.h

// 包含模型

#pragma once

// 在头文件中仅给出模板的声明

template<class T>
T min(T x, T y);

template<typename T>
T max(T x, T y);

char const* min(char const* x, char const* y);

char const* max(char const* x, char const* y);

template<class T>
class Comparator {
public:
    Comparator(T x, T y);

    T min(void) const;

    T max(void) const;

private:
    T x, y;
};

// 在声明模板的头文件中包含定义模板的源文件

// 任何需要获得模板声明的程序必包含此头文件，
// 与此同时，自然也包含了定义该模板的源文件

#include "comparator3.cpp"
```

```
// comparator3.cpp

// 包含模型

#include <string.h>
```

```
6
7  // 在源文件中给出模板的定义
8
9  // 此文件不单独参加编译，以包含文件的形式内嵌于此模板的
10 // 声明文件中，与包含该声明文件的实例化程序一起参加编译
11
12 template<class T>
13 T min(T x, T y) {
14     return x < y ? x : y;
15 }
16
17 template<typename T>
18 T max(T x, T y) {
19     return x < y ? y : x;
20 }
21
22 char const* min(char const* x, char const* y) {
23     return strcmp(x, y) < 0 ? x : y;
24 }
25
26 char const* max(char const* x, char const* y) {
27     return strcmp(x, y) < 0 ? y : x;
28 }
29
30 template<class T>
31 Comparator<T>::Comparator(T x, T y) : x(x), y(y) {}
32
33 template<class T>
34 T Comparator<T>::min(void) const {
35     return x < y ? x : y;
36 }
37
38 template<class T>
39 T Comparator<T>::max(void) const {
40     return x < y ? y : x;
41 }
42
43 template<>
44 char const* Comparator<char const*>::min(void) const {
45     return strcmp(x, y) < 0 ? x : y;
46 }
47
48 template<>
49 char const* Comparator<char const*>::max(void) const {
50     return strcmp(x, y) < 0 ? y : x;
51 }
```

任何希望使用模板的程序都必须包含模板的声明文件，而模板定义文件亦因包含而内嵌于该声明文件，最终与模板的实例化代码同处一个编译单元。模板的声明、定义与实例化同处一个编译单元中，编译器有能力对模板进行正确地实例化，没有任何链接错误。

```
1  // main3.cpp
2
3  // 包含模型
4
```

```cpp
#include <iostream>

#include "comparator3.h" // 既包含了模板的声明文件，也包含了模板的定义文件

using namespace std;

// 根据特定的模板实参对模板进行实例化

int main(void) {
    int nx = 123, ny = 456;
    double dx = 1.23, dy = 4.56;
    string sx = "hello", sy = "world";
    char const *cx = "hello", *cy = "world";

    cout << ::min(nx, ny) << ' ' << ::max(nx, ny) << endl;
    cout << ::min(dx, dy) << ' ' << ::max(dx, dy) << endl;
    cout << ::min(sx, sy) << ' ' << ::max(sx, sy) << endl;
    cout << ::min(cx, cy) << ' ' << ::max(cx, cy) << endl;

    Comparator<int> cn(nx, ny);
    Comparator<double> cd(dx, dy);
    Comparator<string> cs(sx, sy);
    Comparator<char const*> cc(cx, cy);

    cout << cn.min() << ' ' << cn.max() << endl;
    cout << cd.min() << ' ' << cd.max() << endl;
    cout << cs.min() << ' ' << cs.max() << endl;
    cout << cc.min() << ' ' << cc.max() << endl;

    return 0;
}
```

包含模型会延长总体的编译时间，而且模板定义文件不得不连同其声明文件一起提供给模板的使用者，向用户暴露出他们不希望或者不应该了解的实现细节。

8.6.1.4 实例模型

在头文件中仅给出模板的声明，不包含模板定义文件。

```cpp
// comparator4.h

// 实例模型

#pragma once

// 在头文件中仅给出模板的声明

template<class T>
T min(T x, T y);

template<typename T>
T max(T x, T y);

char const* min(char const* x, char const* y);

```

```
char const* max(char const* x, char const* y);

template<class T>
class Comparator {
public:
    Comparator(T x, T y);

    T min(void) const;

    T max(void) const;

private:
    T x, y;
};
```

在模板定义文件中使用实例化指示符，强制编译器在编译该文件时，即根据特定的模板实参对模板进行实例化。

```
// comparator4.cpp

// 实例模型

#include <string.h>

#include <string>

#include "comparator4.h"

using namespace std;

// 在源文件中给出模板的定义

template<class T>
T min(T x, T y) {
    return x < y ? x : y;
}

template<typename T>
T max(T x, T y) {
    return x < y ? y : x;
}

char const* min(char const* x, char const* y) {
    return strcmp(x, y) < 0 ? x : y;
}

char const* max(char const* x, char const* y) {
    return strcmp(x, y) < 0 ? y : x;
}

template<class T>
Comparator<T>::Comparator(T x, T y) : x(x), y(y) {}

template<class T>
T Comparator<T>::min(void) const {
```

```
    return x < y ? x : y;
}

template<class T>
T Comparator<T>::max(void) const {
    return x < y ? y : x;
}

template<>
char const* Comparator<char const*>::min(void) const {
    return strcmp(x, y) < 0 ? x : y;
}

template<>
char const* Comparator<char const*>::max(void) const {
    return strcmp(x, y) < 0 ? y : x;
}

// 显式实例化

template int min<int>(int x, int y);
template double min<double>(double x, double y);
template string min<string>(string x, string y);

template int max<int>(int x, int y);
template double max<double>(double x, double y);
template string max<string>(string x, string y);

template class Comparator<int>;
template class Comparator<double>;
template class Comparator<string>;
template class Comparator<char const*>;
```

显式实例化指示符是由template关键字和紧接其后的实例化实体（类、全局函数、成员函数等）的声明组成。该声明是一个用实参完全替代形参之后的声明。注意，前面已经显式实例化过的成员不能再次实例化。

根据特定的模板实参对模板进行实例化的工作已在模板定义文件中完成。使用该模板时无需再行实例化，直接引用具体类型，调用具体函数即可。

```
// main4.cpp

// 实例模型

#include <iostream>

#include "comparator4.h"

using namespace std;

int main(void) {
    int nx = 123, ny = 456;
    double dx = 1.23, dy = 4.56;
    string sx = "hello", sy = "world";
    char const *cx = "hello", *cy = "world";
```

```
16
17      cout << ::min(nx, ny) << ' ' << ::max(nx, ny) << endl;
18      cout << ::min(dx, dy) << ' ' << ::max(dx, dy) << endl;
19      cout << ::min(sx, sy) << ' ' << ::max(sx, sy) << endl;
20      cout << ::min(cx, cy) << ' ' << ::max(cx, cy) << endl;
21
22      Comparator<int> cn(nx, ny);
23      Comparator<double> cd(dx, dy);
24      Comparator<string> cs(sx, sy);
25      Comparator<char const*> cc(cx, cy);
26
27      cout << cn.min() << ' ' << cn.max() << endl;
28      cout << cd.min() << ' ' << cd.max() << endl;
29      cout << cs.min() << ' ' << cs.max() << endl;
30      cout << cc.min() << ' ' << cc.max() << endl;
31
32      return 0;
33  }
```

显式实例化的类型总是有限的，即便只考虑项目中有限类型的情况，也必须仔细跟踪每个需要实例化的类或函数，这对于大型项目而言，将成为十分繁琐的工作。

8.6.1.5 整合包含模型和实例模型

在声明头文件中给出模板的声明：

```
1   // comparator5.h
2
3   // 整合包含模型和实例模型
4
5   #pragma once
6
7   // 在头文件中给出模板的声明
8
9   template<class T>
10  T min(T x, T y);
11
12  template<typename T>
13  T max(T x, T y);
14
15  char const* min(char const* x, char const* y);
16
17  char const* max(char const* x, char const* y);
18
19  template<class T>
20  class Comparator {
21  public:
22      Comparator(T x, T y);
23
24      T min(void) const;
25
26      T max(void) const;
27
28  private:
29      T x, y;
```

```
};
```

在定义头文件中给出模板的定义：

```
// comparator_def.h

// 整合包含模型和实例模型

#include <string.h>

#include "comparator5.h"

using namespace std;

// 在定义头文件中给出模板的定义

template<class T>
T min(T x, T y) {
    return x < y ? x : y;
}

template<typename T>
T max(T x, T y) {
    return x < y ? y : x;
}

char const* min(char const* x, char const* y) {
    return strcmp(x, y) < 0 ? x : y;
}

char const* max(char const* x, char const* y) {
    return strcmp(x, y) < 0 ? y : x;
}

template<class T>
Comparator<T>::Comparator(T x, T y) : x(x), y(y) {}

template<class T>
T Comparator<T>::min(void) const {
    return x < y ? x : y;
}

template<class T>
T Comparator<T>::max(void) const {
    return x < y ? y : x;
}

template<>
char const* Comparator<char const*>::min(void) const {
    return strcmp(x, y) < 0 ? x : y;
}

template<>
char const* Comparator<char const*>::max(void) const {
    return strcmp(x, y) < 0 ? y : x;
```

```
}
```

在源文件中对模板做显式实例化：

```cpp
// comparator5.cpp

// 整合包含模型和实例模型

#include <string>

#include "comparator_def.h"

using namespace std;

// 在源文件中对模板做显式实例化

template int min<int>(int x, int y);
template double min<double>(double x, double y);
template string min<string>(string x, string y);

template int max<int>(int x, int y);
template double max<double>(double x, double y);
template string max<string>(string x, string y);

template class Comparator<int>;
template class Comparator<double>;
template class Comparator<string>;
template class Comparator<char const*>;
```

究竟使用包含模型还是实例模型由模板的使用者决定：

```cpp
// main5.cpp

// 整合包含模型和实例模型

#include <iostream>

// #include "comparator_def.h" // 使用包含模型
#include "comparator5.h" // 使用实例模型

using namespace std;

int main(void) {
    int nx = 123, ny = 456;
    double dx = 1.23, dy = 4.56;
    string sx = "hello", sy = "world";
    char const *cx = "hello", *cy = "world";

    cout << ::min(nx, ny) << ' ' << ::max(nx, ny) << endl;
    cout << ::min(dx, dy) << ' ' << ::max(dx, dy) << endl;
    cout << ::min(sx, sy) << ' ' << ::max(sx, sy) << endl;
    cout << ::min(cx, cy) << ' ' << ::max(cx, cy) << endl;

    Comparator<int> cn(nx, ny);
```

```
    Comparator<double> cd(dx, dy);
    Comparator<string> cs(sx, sy);
    Comparator<char const*> cc(cx, cy);

    cout << cn.min() << ' ' << cn.max() << endl;
    cout << cd.min() << ' ' << cd.max() << endl;
    cout << cs.min() << ' ' << cs.max() << endl;
    cout << cc.min() << ' ' << cc.max() << endl;

    return 0;
}
```

8.6.1.6 导出模型

通过export关键字将模板声明为导出，即可在其定义不可见的编译单元中，实例化该模板。

```
// comparator6.h

// 导出模型

#pragma once

// 在头文件中仅给出模板的声明

// 借助export关键字将其声明为导出模板

export template<class T>
T min(T x, T y);

export template<typename T>
T max(T x, T y);

char const* min(char const* x, char const* y);

char const* max(char const* x, char const* y);

export template<class T>
class Comparator {
public:
    Comparator(T x, T y);

    T min(void) const;

    T max(void) const;

private:
    T x, y;
};
```

在模板的定义部分不需要再使用export关键字。

```
// comparator6.cpp

// 导出模型

```

```cpp
#include <string.h>

#include "comparator6.h"

// 在源文件中给出模板的定义

// 模板声明部分所强调的export特性，因头文件包含，而为此处的定义隐式保留

template<class T>
T min(T x, T y) {
    return x < y ? x : y;
}

template<typename T>
T max(T x, T y) {
    return x < y ? y : x;
}

char const* min(char const* x, char const* y) {
    return strcmp(x, y) < 0 ? x : y;
}

char const* max(char const* x, char const* y) {
    return strcmp(x, y) < 0 ? y : x;
}

template<class T>
Comparator<T>::Comparator(T x, T y) : x(x), y(y) {}

template<class T>
T Comparator<T>::min(void) const {
    return x < y ? x : y;
}

template<class T>
T Comparator<T>::max(void) const {
    return x < y ? y : x;
}

template<>
char const* Comparator<char const*>::min(void) const {
    return strcmp(x, y) < 0 ? x : y;
}

template<>
char const* Comparator<char const*>::max(void) const {
    return strcmp(x, y) < 0 ? y : x;
}
```

编译器对导出型模板的内部表示会有更加持久的记忆，即便该模板的实例化与它的定义不在同一个编译单元中，亦能被正确地编译，不存在任何链接错误。

```cpp
// main6.cpp

// 导出模型
```

```cpp

#include <iostream>

#include "comparator6.h"

using namespace std;

// 根据特定的模板实参对模板进行实例化

// 此处实例化的模板被声明为导出，因此即便看不到模板的定
// 义，编译器亦能对其进行正确地实例化，不会发生链接错误

int main(void) {
    int nx = 123, ny = 456;
    double dx = 1.23, dy = 4.56;
    string sx = "hello", sy = "world";
    char const *cx = "hello", *cy = "world";

    cout << ::min(nx, ny) << ' ' << ::max(nx, ny) << endl;
    cout << ::min(dx, dy) << ' ' << ::max(dx, dy) << endl;
    cout << ::min(sx, sy) << ' ' << ::max(sx, sy) << endl;
    cout << ::min(cx, cy) << ' ' << ::max(cx, cy) << endl;

    Comparator<int> cn(nx, ny);
    Comparator<double> cd(dx, dy);
    Comparator<string> cs(sx, sy);
    Comparator<char const*> cc(cx, cy);

    cout << cn.min() << ' ' << cn.max() << endl;
    cout << cd.min() << ' ' << cd.max() << endl;
    cout << cs.min() << ' ' << cs.max() << endl;
    cout << cc.min() << ' ' << cc.max() << endl;

    return 0;
}
```

导出模型在很大程度上增加了编译器的实现难度。截至目前，真正支持export关键字的编译器少之又少（仅Comeau C/C++和Intel 7.x编译器支持），GNU和Microsoft的C++编译器都不支持，而且在C++11以后的标准中，此特性已被废除，export关键字也被移作他用。

8.6.2 预编译头文件

截至目前，包含模型仍然是使用模板的主流策略。每个含有实例化模板代码的源文件，都需要同时包含模板的声明和定义，这部分内容会被重复编译多次，这将导致整个编译时间的延长。预编译头文件的作用就是将那些被多个源文件包含的头文件只被编译一次，并在包含该头文件的编译单元中直接引用编译结果。极大地缩短整体编译时间。

8.6.2.1 编译器的内部状态

当编译一个文件时，编译器从文件的开头一直扫描到文件的结束。对其所见到的每个标记（包括来自头文件的标记），编译器都会更新其内部状态，以反应该标记所表达的语义。编译器的内部状态直接决定了其在目标文件中所产生的代码。

如果有多个需要编译的文件，其前N行代码完全相同，那么编译器在编译第一个文件时，就可以把与这N行代码相对应的内部状态保存在一个文件中。待其编译剩下的文件时，先从这个文件中重新加载事先保存好的内部状态，然后从第N+1行开始编译。从一个文件中加载与N行代码相对应的内部状态，比实际编译N行代码快得多。

8.6.2.2 预编译头文件

多个文件的前N行代码完全相同，这种情况更多地表现为，以相同的顺序包含一组相同的头文件，特别是那些几乎每个程序都会用到的标准头文件，如stdio.h或iostream等。先编译这组头文件，并将编译过程中所形成的内部状态保存在一个专门的文件中，然后在编译所有以包含这组头文件为前N行代码的文件时，直接从这个专门的文件中加载与头文件有关的信息，避免重复编译完全相同的内容，缩短编译时间，提高编译速度。这种编译技术称为预编译头技术。目前大多数C/C++编译器都支持该技术，而那个用于保存编译器内部状态的专门文件则称为预编译头文件（GNU编译器生成的.gch文件或Microsoft编译器生成的.pch文件）。

8.6.2.3 头文件并集

在使用预编译头技术时，所包含的头文件应该是各模块所需头文件的并集。这对于每个具体模块而言可能会有部分冗余，但就整体而言要比只选择有用的头文件，能够获得更快的编译速度。

更一般化的做法是，把那些包括标准头文件在内的，不会或很少修改的头文件及代码集中放在一个头文件中；编译该头文件，得到预编译头文件；然后在每个需要使用其中内容的文件的第一行包含该头文件，编译器会自动为其加载预编译头文件，而不是重新编译该头文件。

```
g++ -c comparator3.h -> comparator3.h.gch
```

8.6.3 浅层实例化

8.6.3.1 诊断信息跟踪所有层次

底层模板往往是在实例化上层模板的过程中被实例化的。因此，一旦因为底层模板的错误而导致上层模板不能被正确地实例化，编译器所给出的诊断信息通常会包含对产生这个问题的所有层次的完整跟踪。程序员往往很难从这么多信息中快速找到症结所在。

```
// behind.cpp

// 诊断信息跟踪所有层次

template<typename P>
void zero(P p) {
    *p = 0; // 完整的错误信息包含对模板实例化过程的全程跟踪
}

template<typename P>
void core(P p) {
    zero(p);
}

template<typename T>
void middle(typename T::P p) {
    core(p);
}

template<typename T>
```

```
void shell(void) {
    typename T::P p;
    middle<T>(p);
}

class Object {
public:
    typedef int P;
};

int main(void) {
    shell<Object>();

    return 0;
}
```

8.6.3.2 提前验证模板实参

在程序中增加一些不可能执行到哑代码，只是为了在实例化上层模板时，提前验证一下所用模板实参能否满足底层模板所需要的操作，这种编程技巧称为浅层实例化。

```
// ahead.cpp

// 提前验证模板实参

template<typename P>
void zero(P p) {
    *p = 0; // 完整的错误信息包含对模板实例化过程的全程跟踪
}

template<typename P>
void core(P p) {
    zero(p);
}

template<typename T>
void middle(typename T::P p) {
    core(p);
}

template<typename T>
void shell(void) {
    // 浅层实例化
    class Shallow {
        void deref(typename T::P p) {
            *p = 0; // 当高层模板实参不符合低层模板约束时，提前报错
        }
    };

    typename T::P p;
    middle<T>(p);
}

class Object {
public:
```

```
    typedef int P;
};

int main(void) {
    shell<Object>();

    return 0;
}
```

8.6.4 跟踪器

8.6.4.1 测试模板功能和性能

跟踪器是一个用户自定义的类，可以做为测试模板功能和性能的类型实参。

```
// tracer.h

// 测试模板功能和性能

class Tracer {
public:
    Tracer(void);
    Tracer(int n);
    Tracer(Tracer const& tracer);

    Tracer& operator=(Tracer const& tracer);

    ~Tracer(void);

private:
    int n;
};
```

8.6.4.2 跟踪器仅供测试

跟踪器的定义有且仅有满足模板测试的功能。

```
// tracer.cpp

// 跟踪器仅供测试

#include <iostream>

#include "tracer.h"

using namespace std;

Tracer::Tracer(void) {
    cout << "缺省构造: " << this << endl;
}

Tracer::Tracer(int n) : n(n) {
    cout << "有参构造: " << this << endl;
}
```

```
18
19  Tracer::Tracer(Tracer const& tracer) : n(tracer.n) {
20      cout << "拷贝构造: " << &tracer << "->" << this << endl;
21  }
22
23  Tracer& Tracer::operator=(Tracer const& tracer) {
24      cout << "拷贝赋值: " << &tracer << "->" << this << endl;
25
26      if (&tracer != this)
27          n = tracer.n;
28
29      return *this;
30  }
31
32  Tracer::~Tracer(void) {
33      cout << "析构函数: " << this << endl;
34  }
```

8.6.4.3 跟踪器的覆盖范围

对于被跟踪模板的每个操作，跟踪器都应该有一个与之对应的跟踪动作。

```
1   // vector.cpp
2
3   // 跟踪器的覆盖范围
4
5   #include <vector>
6
7   #include "tracer.h"
8
9   using namespace std;
10
11  int main(void) {
12      vector<Tracer> v1(3);
13      vector<Tracer> v2(3, 10);
14
15      v2.push_back(20);
16      v2.pop_back();
17
18      v2.erase(v2.begin());
19
20      v2.resize(3);
21      v2.resize(4);
22      v2.reserve(10);
23
24      v2.clear();
25
26      v2.push_back(30);
27
28      v1 = v2;
29
30      return 0;
31  }
```

8.6.5 静态多态

所谓多态，即令同一种类型表现出不同的行为特征。如果把类模板也看作一种类型，为其提供不同的类型实参，即可表现出不同的行为特征，这种多态称为静态多态。与基于虚函数的动态多态相比，静态多态具有更好的时间性能，而且代码也更为简洁。

```
// poly.cpp

// 基于模板的静态多态

#include <iostream>

using namespace std;

template<typename Drawable>
class Shape {
public:
    Shape(Drawable const& drawable) : drawable(drawable) {}

    void draw(void) const {
        drawable.draw();
    }

private:
    Drawable const& drawable;
};

class Rect {
public:
    Rect(double x, double y, double w, double h) : x(x), y(y), w(w), h(h) {}

    void draw(void) const {
        cout << "Rect(" << x << ',' << y << ',' << w << ',' << h << ')' <<
endl;
    }

private:
    double x, y, w, h;
};

class Circle {
public:
    Circle(double x, double y, double r) : x(x), y(y), r(r) {}

    void draw(void) const {
        cout << "Circle(" << x << ',' << y << ',' << r << ')' << endl;
    }

private:
    double x, y, r;
};

int main(void) {
    Rect rect(1, 2, 3, 4);
    Shape<Rect> shape1 = rect;
```

```
    shape1.draw();

    Circle circle(5, 6, 7);
    Shape<Circle> shape2 = circle;
    shape2.draw();

    return 0;
}
```