# 8.5 函数模板

现在的C++编译器实现了C++新增的一项特性——函数模板。函数模板是通用的函数描述，也就是说，它们使用泛型来定义函数，其中的泛型可用具体的类型（如int或double）替换。通过将类型作为参数传递给模板，可使编译器生成该类型的函数。由于模板允许以泛型（而不是具体类型）的方式编写程序，因此有时也被称为通用编程。由于类型是用参数表示的，因此模板特性有时也披称为参数化类别（parameterized types）。下面介绍为何需要这种将性以及其工作原理。

在前面的程序清单8.4中，定义了一个交换两个int值的函数。假设要交换两个double值，则一种办法是复制原来的代码，并用double替换所有的int。如果需要交换两个char值，可以再次使用同样的技术。进行这种修改将浪费宝贵的时间，且容易出错。如果进行手工修改，则可能会漏掉一个int。如果进行全局查找和替换（如用double替换int）时，可能将：

int x;

short interval;

转换为：

double x; // 预期的类型转换

short doubleerval; // 非预期的变量名称的改变

C++的函数模板功能能自动完成这一过程，可以节省时间，而且更可靠。

函数模板允许以任意类型的方式来定义函数。例如，可以这样建立一个交换模板：

template <typename AnyType>

void Swap (AnyType &a,AnyType &b)

{

 AnyType temp;

 temp = a;

 a = b;

 b = temp;

}

第一行指出，要建立一个模板，并将类型命名为AnyType。关键字template和typename是必需的，除非可以使用关键字class代替typename、另外，必须使用尖括号。类型名可以任意选择（这里为AnyType），只要遵守C++命名规则即可；许多程序员都使用简单的名称，如T。余下的代码描述了交换两个AnyType值的算法。模板并不创建任何函数，而只是告诉编译器如何定义函数。需要交换int的函数时，编萍器将按模板模式创建这样的函数，并用int代替AnyType。同样，需要交换double的函数时，编译器将按模板模式创建这样的函数，并用double代替AnyType。

在标准C++98添加关键typename之前，C++使用关键字class来创建模板。也就是说，可以这样编写模板定义：

template <class AnyType>

void Swap (AnyType &a,AnyType &b)

{

AnyType temp;

temp = a;

 a = b;

 b = temp;

}

typename关键字使得参数AnyType表示类型这一点更为明显；然而，有大量代码库是使用关键字class开发的。在这种上下文中，这两个关键字是等价的。本书使用了这两种形式，旨在让您在其他地方遇到它们时不会感到陌生。

**提示**：如果需要多个将同一种算法用于不同类型的函数，请使用模板。如果不考虑向后兼容的问题，并愿意键入较长的单词，则声明类型参数时，应使用关键字typename而不使用class。

要让编译器知道程序需要一个特定形式的交换函数，只需在程序中使用Swap()函数即可。编译器将检查所使用的参数类型，并生成相应的函数。程序清单8.11演示为何可以这样做。该程序的布局和使用常规函数时相同，在文件的开始位置提供模板函数的原型，并在main()后面提供模板函数的定义。这个示例采用了更常见的做法，即将T而不是AnyType用作类型参数。

**程序清单8.11 funtemp.cpp**

// funtemp.cpp -- 使用函数模板

#include <iostream>

// 函数模板原型

template <typename T> // 或class T

void Swap(T &a, T &b);

int main()

{

 using namespace std;

 int i = 10;

 int j = 20;

 cout << "i, j = " << i << ", " << j << ".\n";

 cout << "Using compiler-generated int swapper:\n";

 Swap(i,j); // 生成void Swap(int &, int &)

 cout << "Now i, j = " << i << ", " << j << ".\n";

 double x = 24.5;

 double y = 81.7;

 cout << "x, y = " << x << ", " << y << ".\n";

 cout << "Using compiler-generated double swapper:\n";

 Swap(x,y); // 生成void Swap(double &, double &)

 cout << "Now x, y = " << x << ", " << y << ".\n";

 cin.get();

 return 0;

}

// 函数模板定义

template <typename T> // 或class T

void Swap(T &a, T &b)

{

 T temp; // 类型为T的临时变量

 temp = a;

 a = b;

 b = temp;

}

程序清单8.11中的第一个Swap()函数接受两个int参数，因此编译器生成该函数的int版本。也就是说，用int替换所有的T，生成下面这样的定义：

void Swap(int &a, int &b)

{

 int temp;

 temp = a;

 a = b;

 b = temp;

}

程序员看不到这些代码。，但编译器确实生成并在程序中使用了它们。第二个Swap()函数接受两个double参数，因此编译器将生成double版本。也就是说，用double替换T，生成下述代码：

void Swap(double &a, double &b)

{

 double temp;

 temp = a;

 a = b;

 b = temp;

}

下面是程序清单8.11中程序的输出，从中可知，这种处理方式是可行的：

i,j = 10, 20.

Using compiler-generated int swapper:

Now I,j = 20,10

x,y =24.5,81.7

Using compiler-generated double swapper:

Now x,y = 81.7,24.5.

注意，函数模板不能缩短可执行程序。对于程序清单8.11，最终仍将有两个独立的函数定义，就像以前手工方式定义了这些函数一样。最终的代码不包含任何模板，而只包含了为程序生成的实际函数。使用模板的好处是，它使生成多个函数定义更简单、更可靠。

更常见的情形是，将模板放在头文件总，并在需要使用模板的文件中包含头文件。头文件将在第9章中讨论讨论。

## 8.5.1 重载的模板

需要多个对不同类型使用同一种算法的函数时，可使用模板，如程序清单8.11所示。然而，并非所有的类型都使用相同的算法。为满足这种需求，可以像重载常规函数定义那样重载模板定义。和常规重载一样，被重载的模板的函数特征标必须不同。例如，程序清单8.12新增了一个交换模板，用于交换两个数组中的元素。原来的模板的特征标为(T &,T &)，而新模板的特征标为( T[]，T[],int)。注意，在后一个模板中，最后一个参数的类型为具体类型（int），而不是泛型。并非所有的模板参数都必须是模板参数类型。

编译器见到twotemps.cpp中第一个Swap()函数调用时，发现它有两个int参数，因此将它与原来的模板匹配。但第二次调用将两个int数组和一个int值用作参数，这与新模板匹配。

**程序清单8.12 twotemps.cpp**

// twotemps.cpp -- 使用重载的函数模板

#include <iostream>

template <typename T> // 初始模板

void Swap(T &a, T &b);

template <typename T> // 新模板

void Swap(T \*a, T \*b, int n);

void Show(int a[]);

const int Lim = 8;

int main()

{

 using namespace std;

 int i = 10, j = 20;

 cout << "i, j = " << i << ", " << j << ".\n";

 cout << "Using compiler-generated int swapper:\n";

 Swap(i,j); // 匹配初始模板

 cout << "Now i, j = " << i << ", " << j << ".\n";

 int d1[Lim] = {0,7,0,4,1,7,7,6};

 int d2[Lim] = {0,7,2,0,1,9,6,9};

 cout << "Original arrays:\n";

 Show(d1);

 Show(d2);

 Swap(d1,d2,Lim); // 匹配新模板

 cout << "Swapped arrays:\n";

 Show(d1);

 Show(d2);

 // cin.get();

 return 0;

}

template <typename T>

void Swap(T &a, T &b)

{

 T temp;

 temp = a;

 a = b;

 b = temp;

}

template <typename T>

void Swap(T a[], T b[], int n)

{

 T temp;

 for (int i = 0; i < n; i++)

 {

 temp = a[i];

 a[i] = b[i];

 b[i] = temp;

 }

}

void Show(int a[])

{

 using namespace std;

 cout << a[0] << a[1] << "/";

 cout << a[2] << a[3] << "/";

 for (int i = 4; i < Lim; i++)

 cout << a[i];

 cout << endl;

}

下面是程序清单8.12中程序的输出：

i，j = 10, 20.

Using compiler-generated int swapper:

Now i,j = 20, 10.

Original arrays：

07/04/1776

07/20/1969

Swapped arrays：

07/20/1969

07/04/1776

## 8.5.2 模板的局限性

假设有如下模板函数：

template<class T> // 或者template <typename T>

void f(T a，T b)

{… }

通常，代码假定可执行哪些操作。例如，下面的代码假定定义了赋值，但如果T为数组，这种假设将不成立：

a = b;

同样，下面的语句假设定义了<，但如果T为结构，该假设便不成立：

if(a > b)

另外，为数组名定义了运算符>，但由于数组名为地址，因此它比较的是数组的地址，而这可能不是您希望的。下面的语句假定为类型T定义了乘法运算符，但如果T为数组、指针或结构，这种假设便不成立：

T c = a\*b;

总之，编写的模板函数很可能无法处理某些类型。另一方面，有时候通用化是有意义的，但C++语法不允许这样做。例如，将两个包含位置坐标的结构相加是有意义的，虽然没有为结构定义运算符+。一种解决方案是，C++允许您重载运算符+，以便能够将其用于特定的结构或类（运算符重载将存第11章讨论）。这样使用运算符+的模板便可处理重载了运算符+的结构。另一种解决方案是，为特定类型提供具体化的模板定义，下面就来介绍这种解决方案。

## 8.5.3 显式具体化

假设定义了如下结构：

struct job

{

 char name[40];

 double salary;

 int floor;

};

另外，假设希望能够交换两个这种结构的内容。原来的模板使用下面的代码来完成交换：

temp = a;

a = b;

b = temp;

由于C++允许将一个结构赋给另一个结构，因此即使T是一个job结构，上述代码也适用。然而，假设只想交换salary和floor成员，而不交换name成员，则需要使用不同的代码，但Swap()的参数将保持不变（两个job结构的引用），因此无法使用模板重载来提供其他的代码。

然而，可以提供一个具体化函数定义——称为**显式具体化（explicit specialization）**，其中包含所需的代码。当编译器找到与函数调用匹配的具体化定义时，将使用该定义，而不再寻找模板。

具体化机制随着C++的演变而不断变化。下面介绍C++标准定义的形式。

### 1．第三代具体化（ISO/ANSI C++标准）

试验其他具体化方法后，C++98标准选择了下面的方法。

* 对于给定的函数名，可以有非模板函数、模板函数和显式具体化模板函数以及它们的重载版本。
* 显式具体化的原型和定义应以template<>打头，并通过名称来指出类型。
* 具体化优先于常规模板，而非模板函数优先于具体化和常规模板。

下面是用于交换job结构的非模板函数、模板函数和具体化的原型：

// 非模板函数原型

void Swap(job & , job &);

// 模板函数原型

template <typename T>

void Swap(T &, T &);

// 用于job类型的显式具体化原型

template <> void Swap<job>(job & , job &);

正如前面指出的，如果有多个原型，则编译器在选择原型时，非模板版本优先于显式具体化和模板版本，而显式具体化优先于使用模板生成的版本。例如，在下面的代码中，第一次调用Swap()时使用通用版本，而第二次调用使用基于job类型的显式具体化版本。

template <class T> // 模板

void Swap(T & ,T &);

// 用于job类型的显示具体化

template <> void Swap<job>(job &, job &);

int main ( )

{

 double u, v;

 …

 Swap(u,v); // 使用模板

 job a,b;

 Swap(a.b); // 使用void Swap<job>(job &, job &)

}

Swap<job>中的<job>是可选的，因为函数的参数类型表明，这是job的一个具体化。因此，该原型也可以这样编写：

template<> void Swap(job & , job&); // 简化表示

下面来看一看显式具体化的工作方式。

### 2．显式具体化示例

程序清单8.13演示了显式具体化的工作方式。

**程序清单8.13 twoswap.cpp**

// twoswap.cpp -- 显式具体化覆盖一个模板

#include <iostream>

template <typename T>

void Swap(T &a, T &b);

struct job

{

 char name[40];

 double salary;

 int floor;

};

// 显式具体化

template <> void Swap<job>(job &j1, job &j2);

void Show(job &j);

int main()

{

 using namespace std;

 cout.precision(2);

 cout.setf(ios::fixed, ios::floatfield);

 int i = 10, j = 20;

 cout << "i, j = " << i << ", " << j << ".\n";

 cout << "Using compiler-generated int swapper:\n";

 Swap(i,j); // 生成Swap(int &, int &)

 cout << "Now i, j = " << i << ", " << j << ".\n";

 job sue = {"Susan Yaffee", 73000.60, 7};

 job sidney = {"Sidney Taffee", 78060.72, 9};

 cout << "Before job swapping:\n";

 Show(sue);

 Show(sidney);

 Swap(sue, sidney); // 使用void Swap(job &, job &)

 cout << "After job swapping:\n";

 Show(sue);

 Show(sidney);

 cin.get();

 return 0;

}

template <typename T>

void Swap(T &a, T &b) // 通用版本

{

 T temp;

 temp = a;

 a = b;

 b = temp;

}

// 只交换job结构中的salary和floor字段

template <> void Swap<job>(job &j1, job &j2) // 具体化

{

 double t1;

 int t2;

 t1 = j1.salary;

 j1.salary = j2.salary;

 j2.salary = t1;

 t2 = j1.floor;

 j1.floor = j2.floor;

 j2.floor = t2;

}

void Show(job &j)

{

 using namespace std;

 cout << j.name << ": $" << j.salary

 << " on floor " << j.floor << endl;

}

下面是该程序的输出：

i, j = 10, 20.

Using compiler-generated int swapper

Now i, j = 20, 10.

Before job swapping :

Susan Yaffee : $73000.60 on floor 7

Sidney Taffee: $78060.72 on floor 9

After job swapping :

Susan Yaffee: $78060.72 on floor 9

Sidney Taffee: $73000.60 on floor 7

## 8.5.4 实例化和具体化

为进一步了解模板，必须理解术语实例化和具体化。记住，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）。例如，在程序清单8.13中，函数调用Swap(i,j)导致编译器生成Swap()的一个实例，该实例使用int类型。模板并非函数定义，但使用int的模板实例是函数定义。这种实例化方式被称为隐式实例化（implicit instantiation），因为编译器之所以知道需要进行定义，是由于程序调用Swap()函数时提供了int参数。

最初，编译器只能通过隐式实例化，来使用模板生成函数定义，但现在C++还允许显式实例化（explicit instantiation）。这意味着可以直接命令编译器创建特定的实例，如Swap<int>()。其语法是，声明所需的种类——用<>符号指示类型，并在声明前加上关键字template：

template void Swap<int>(int ,int); // 显式实例化

实现了这种特性的编译器看到上述声明后，将使用Swap()模板生成一个使用int类型的实例。也就是说，该声明的意思是“使用Swap()模板生成int类型的函数定义。”

与显式实例化不同的是，显式具体化使用下面两个等价的声明之一：

template <> void Swap<int>(int &,int&); // 显式具体化

template <> void Swap(int & ,int &); // 显式具体化

区别在于，这些声明的意思是“不要使用Swap()模板来生成函数定义，而应使用专门为int类型显式地定义的函数定义”。这些原型必须有自己的函数定义。显式具体化声明在关键字template后包含<>，而显式实例化没有。

**警告**：试图在同一个文件（或转换单元）中使用同一种类型的显式实例和显式具体化将出错。

还可通过在程序中使用函数来创建显式实例化。例如，请看下面的代码：

template <class T>

T Add(T a,T b)

{

 return a+b;

}

…

int m = 6;

double x = 10.2;

cout << Add<double>(x,m)<<endl; // 显式实例化

这里的模板与函数调用Add(x,m)不匹配，因为该模板要求两个函数参数的类型相同。但通过使用Add<double>(x，m)，可强制为double类型实例化，并将参数m强制转换为double类型，以便与函数Add<double>(double,double)的第二个参数匹配。

如果对Swap()做类似的处理，结果将如何呢？

int m = 5;

double x = 14.3;

Swap<double> (m, x);

这将为类型double生成一个显式实例化。不幸的是，这些代码不管用，因为第一个形参的类型为double &，不能指向int变量m。

隐式实例化、显式实例化和显式具体化统称为具体化（specialization）。它们的相同之处在于，它们表示的都是使用具体类型的函数定义，而不是通用描述。

引入显式实例化后，必须使用新的语法—一在声明中使用前缀template和template<>，以区分显式实例化和显式具体化。通常，功能越多，语法规则也越多。下面的代码片段总结了这些概念：

…

template <class,T>

void Swap(T &,T &); // 模板原型

template <> void Swap<job>(job &,job &); // 用于job类型的显式具体化

int main(void)

{

 template void Swap<char>(char &,char &) // 用于char的显式实例化

 short a,b;

 …

 Swap(a,b); // 用于short的隐式实例化

 job n,m;

 …

 Swap(n,m); // 使用用于job类型的显式具体化

 char g,h;

 …

 Swap(g,h); // 使用用于char的显式模板实例化

 …

}

编译器看到char的显式实例化后，将使用模板定义来生成Swap()的char版本。对于其他Swap()调用，编译器根据函数调用中实际使用的参数，生成相应的版本。例如，当编译器看到函数调用Swap(a,b)后，将生成Swap()的short版本，因为两个参数的类型都是short。当编译器看到Swap(n,m)后，将使用为job类型提供的独立定义（显式具体化）。当编译器看到Swap(g.h)后，将使用处理显式实例化时生成的模板具体化。

## 8.5.5 编译器选择使用哪个函数版本

对于函数重载、函数模板和函数模板重载，C++需要（且有）一个定义良好的策略，来决定为函数调用使用哪一个函数定义，尤其是有多个参数时。这个过程称为重载解析（overloading resolution）。详细解释这个策略将需要将近一章的篇幅，因此我们先大致了解一下这个过程是如何进行的。

* 第1步：创建候选函数列表。其中包含与被调用函数的名称相同的函数和模板函数。
* 第2步：使用候选函数列表创建可行函数列表。这些都是参数数目正确的函数，为此有一个隐式转换序列，其中包括实参类型与相应的形参类型完全匹配的情况。例如，使用float参数的函数调用可以将该参数转换为double，从而与double形参匹配，而模板可以为float生成一个实例。
* 第3步：确定是否有最佳的可行函数。如果有，则使用它，否则该函数调用出错。

考虑只有一个函数参数的情况，如下面的调用：

may(‘B’}; // 实参为char类型

首先，编译器将寻找候选者，即名称为may()的函数和函数模板。然后寻找那些可以用一个参数调用的函数。例如，下面的函数符合要求，因为其名称与被调用的函数相同，且可只给它们传递一个参数：

void may(int); // #1

float may(float,float=3); // #2

void may (char); // #3

char \* may(const char \*); // #4

char may(const char &); // #5

template<class T> void may(const T &); // #6

template<class T> void may(T \*); // #7

注意，只考虑特征标，而不考虑返回类型。其中的两个候选函数（#4和#7）不可行，因为整数类型不能被隐式地转换（即没有显式强制类型转换）为指针类型。剩余的一个模板可用来生成具体化，其中T被替换为char类型。这样剩下5个可行的函数，其中的每一个函数，如果它是声明的唯一个函数，都可以被使用。

接下来，编译器必须确定哪个可行函数是最佳的。它查看为使函数调用参数与可行的候选函数的参数匹配所需要进行的转换。通常，从最佳到最差的顺序如下所述。

1．完全匹配，但常规函数优先于模板。

2．提升转换（例如，char和shorts自动转换为int，float自动转换为double）。

3．标准转换（例如，int转换为char，long转换为double）。

4．用户定义的转换，如类声明中定义的转换。

例如，函数#1优于函数#2，因为char到int的转换是提升转换（参见第3章），而char到float的转换是标准转换（参见第3章）。函数#3、函数#5和函数#6都优于函数#1和#2，因为它们都是完全匹配的。#3和#5优于#6，因为#6函数是模板。这种分析引出了两个问题。什么是完全匹配？如果两个函数（如#3和#5）都完全匹配，将如何办呢？通常，有两个函数完全匹配是一种错误，但这一规则有两个例外。显然，我们需要对这一点做更深入的探讨。

### 1．完全匹配和最佳匹配

进行完全匹配时，C++允许某些“无关紧要的转换”。表8.1列出了这些转换——Type表示任意类型。例如，int实参与int &形参完全匹配。注意，Type可以是char &这样的类型，因此这些规则包括从char &到const char &的转换。Type（argument-list）意味着用作实参的函数名与用作形参的函数指针只要返回类型和参数列表相同，就是匹配的（第7章介绍了函数指针以及为何可以将函数名作为参数传递给接受函数指针的函数）。第9章将介绍关键字volatile。

**8.1 完全匹配允许的无关紧要转换**

|  |  |
| --- | --- |
| 从实参 | 到形参 |
| Type | Type & |
| Type & | Type |
| Type[] | \* Type |
| Type (argumcnt-list) | Type ( \* ) (argument-list) |
| Type | const Type |
| Type | volatile Type |
| Type \* | const Type |
| Type \* | volatile Type \* |

假设有下面的函数代码：

struct blot {int a;char b[10];};

blot ink={25, "spots”};

…

recycle (ink);

在这种情况下，下面的原型都是完全匹配的：

void recycle (blot); // #1 blot-to-blot

void recycle (const blot); // #2 blot-to- (const blot)

void recycle(blot &); // #3 blot-to-{blot &）

void recycle(const blot &); // #4 blot-to- (const blot&)

正如您预期的，如果有多个匹配的原型，则编译器将无法完成重载解析过程：如果没有最佳的可行函数，则编译器将生成一条错误消息，该消息可能会使用诸如“ambiguous（二义性）”这样的词语。

然而，有时候，即使两个函数都完全匹配，仍可完成重载解析。首先，指向非const数据的指针和引用优先与非const指针和引用参数匹配。也就是说，recycle()示例中，如果只定义了函数#3和#4是完全匹配的，则将选择#3，因为ink没有被声明为const。然而，const和非const之间的区别只适用于指针和引用指向的数据。也就是说，如果只定义了#1和#2，则将出现二义性错误。

一个完全匹配优于另一个的另一种情况是，其中一个是非模板函数，而另一个不是。在这种情况下，非模板函数将优先于模板函数（包括显式具体化）。

如果两个完全匹配的函数都是模板函数，则较具体地模板函数优先。例如，这意味着显式具体化将优于使用模板隐式生成的具体化：

struc blot (int a;char b[10];);

template <class Type> void recycle (Type t) // 模板

template <> void recycled<blot> (blot & t); // 用于blot的具体化

…

blot ink = {25,”spots”};

…

recycle(ink) // 使用具体化

术语“最其体（most specialized）”并不意味着显式具体化，而是指编译推断使用哪种类型时执行的转换最少。例如，请看下面两个模板：

template <class Type> void recycle(Type t); // #1

template <class Type> void recycle(Type \* t); // #2

假设包含这些模板的程序也包含如下代码：

struc blot (int a;char b[10];);

blot ink = {25,”spots”};

…

recycle(ink);

recycle(&ink)调用与#1模板匹配，匹配时将Type解释为blot\*。recycle(&ink)函数调用也与#2模板匹配，这次Type被解释为ink。因此将两个隐式实例——recycle<blot\*>(blot\*)和recycle<blot>(blot\*)发送到可行函数池中。

在这两个模板函数中，recycle<blot\*>(blot\*)被认为是更具体的，因为在生成过程中，它需要进行的转换更少。也就是说，#2模板已经显式指出，函数参数是指向Type的指针，因此可以直接用blot标识Type；而#1模板将Type作为函数参数，因此Type必须被解释为指向blot的指针。也就是说，在#2模板中，Type已经被具体化为指针，因此说它“更具体”。

用于找出最具体的模板的规则被称为函数模板的部分排序规则（partial ordering rules）。和显式实例一样，这也是C++98新增的特性。

### 2．部分排序规则示例

我们先看一个完整的程序，它使用部分排序规则来确定要使用哪个模板定义。程序清单8.14有两个用来显示数组内容的模板定义。第一个定义（模板A）假设作为参数传递的数组中包含了要显示的数据；第二个定义（模板B）假设数组元素为指针，指向要显示的数据。

**程序清单-8.14- tempover.cpp**

// tempover.cpp --- 模板重载

#include <iostream>

template <typename T> // 模板A

void ShowArray(T arr[], int n);

template <typename T> // 模板B

void ShowArray(T \* arr[], int n);

struct debts

{

 char name[50];

 double amount;

};

int main()

{

 using namespace std;

 int things[6] = {13, 31, 103, 301, 310, 130};

 struct debts mr\_E[3] =

 {

 {"Ima Wolfe", 2400.0},

 {"Ura Foxe", 1300.0},

 {"Iby Stout", 1800.0}

 };

 double \* pd[3];

// set pointers to the amount members of the structures in mr\_E

 for (int i = 0; i < 3; i++)

 pd[i] = &mr\_E[i].amount;

 cout << "Listing Mr. E's counts of things:\n";

// things is an array of int

 ShowArray(things, 6); // 使用模板A

 cout << "Listing Mr. E's debts:\n";

// pd is an array of pointers to double

 ShowArray(pd, 3); // 使用模板B (更具体)

 // cin.get();

 return 0;

}

template <typename T>

void ShowArray(T arr[], int n)

{

 using namespace std;

 cout << "template A\n";

 for (int i = 0; i < n; i++)

 cout << arr[i] << ' ';

 cout << endl;

}

template <typename T>

void ShowArray(T \* arr[], int n)

{

 using namespace std;

 cout << "template B\n";

 for (int i = 0; i < n; i++)

 cout << \*arr[i] << ' ';

 cout << endl;

}

请看下面的函数调用：

ShowArray(things,6);

标识符things是一个int数组的名称，因此与下面的模板匹配：

template <typename T> // 模板A

void ShowArray(T arr[],int n);

其中T被替换为int类型。

接下来，请看下面的函数调用：

ShowArray(pd,3f);

其中pd是一个double\*数组的名称。这与模板A匹配：

template <typename T> // 模板A

void ShowArray（T arr[],int n）;

其中，T被替换为类型double\*，在这种情况下，模板函数将显示pd数组的内容，即3个地址。该函数调用也与模板B匹配：

template <typename T> // 模板B

void ShowArray（T \* arr[],int n）;

在这里，T被替换为类型double，而函数将显示被解除引用的元素\*arr[i]，即数组内容指向的double值。在这两个模板中，模板B更具体，因为它做了特定的假设——数组内容是指针，因此被使用。

下面是程序清单8.14中程序的输出：

Listingl Mr.E’s counts of things：

template A

13 31 103 301 310 130

Listing Mr．E’S debts:

template B

2400 1300 1800

如果将模板B从程序中册除，则编译器将使用模板A来显示pd的内容，因此显示的将是地址，而不是值。请试试看。

简而言之，重载解析将寻找最匹配的函数。如果只存在一个这样的函数，则选择它；如果存在多个这样的函数，但其中只有一个是非模板函数，则选择该函数；如果存在多个适合的函数，且它们都为模板函数，但其中有一个函数比其他函数更具体，则选择该函数。如果有多个同样合适的非模板函数或模板函数，但没有一个函数比其他函数更具体，则函数调用将是不确定的，因此是错误的；当然，如果不存在匹配的函数，则也是错误的。

### 3．创建自定义选择

在有些情况下：可通过编写合适的函数调用，引导编译器做出您希望的选择。请看程序清单8.15，该程序将模板函数定义放在文件开头，从而无需提供模板原型。与常规函数一样，通过在使用函数前提供模板函数定义，它让它也充当原型。

**程序清单8.15 choice.cpp**

// choices.cpp -- 选择模板

#include <iostream>

template<class T>

T lesser(T a, T b) // #1

{

 return a < b ? a : b;

}

int lesser (int a, int b) // #2

{

 a = a < 0 ? -a : a;

 b = b < 0 ? -b : b;

 return a < b ? a : b;

}

int main()

{

 using namespace std;

 int m = 20;

 int n = -30;

 double x = 15.5;

 double y = 25.9;

 cout << lesser(m, n) << endl; // use #2

 cout << lesser(x, y) << endl; // use #1 with double

 cout << lesser<>(m, n) << endl; // use #1 with int

 cout << lesser<int>(x, y) << endl; // use #1 with int

 // cin.get();

 return 0;

}

最后的函数调用将double转换为int，有些编译器会针对这一点发出警告。

该程序的输出如下：

20

15.5．

-30

15

程序清单8.15提供了一个模板和一个标准函数，其中模板返回两个值中较小的一个，而标准函数返回两个值中绝对值较小的那个。如果函数定义是在使用函数前提供的，它将充当函数原型，因此这个示例无需提供原型。请看下面的语句：

cout << lesser(m,n)<<endl; // 使用#2

这个函数调用与模板函数和非模板函数都匹配，因此选择非模板函数，返回20。

接下来，下述语句中的函数调用与模板匹配（T为double）：

cout<< lesser(x,y)<<endl; // 使用#1的double类型

现在来看下面的语句：

cout<<lesser<>(m,n)<<endl; // 使用#1的int类型

lesser<>(m,n)中的<>指出，编译器应选择模板函数，而不是非模板函数；编译器注意到实参的类型为int，因此使用int替代T对模板进行实例化。

最后，请看下面的语句：

cout<<lesser<int>(x,y)<<endl; // 使用#1的int类型

这条语句要求进行显式实例化（使用int替代T），将使用显式实例化得到的函数。x和y的值将被强制转换为int，该函数返回一个int值，这就是程序显示15而不是15.5的原因所在。

### 4．多个参数的函数

将有多个参数的函数调用与有多个参数的原型进行匹配时，情况将非常复杂。编译器必须考虑所有参数的匹配情况。如果找到比其他可行函数都合适的函数，则选择该函数。一个函数要比其他函数都合适，其所有参数的匹配程度都必须不比其他函数差，同时至少有一个参数的匹配程度比其他函数都高。

本书并不是要解释复杂示例的匹配过程，这些规则只是为了让任何一组函数原型和模板都存在确定的结果。

## 8.5.6 模板函数的发展

在C++发展的早期，大多数人都没有想到模板函数和模板类会有这么强大而有用，它们甚至没有就这个主题发挥想象力。但聪明而专注的程序员挑战模板技术的极限，阐述了各种可能性。根据熟悉模板的程序员提供的反馈，C++98标准做了相应的修改，并添加了标准模板库。从此以后，模板程序员在不断探索各种可能性，并消除模板的局限性。C++11标准根据这些程序员的反馈做了相应的修改。下面介绍一些相关的问题及其解决方案。

### 1．是什么类型

在C++98中，编写模板函数时，一个问题是并非总能知道应在声明中使用哪种类型。请看下面这个不完整的示例：

template<class T1, class T2>

void ft(Tl x, T2 y)

{

 …

 ?type? xpy = x+y;

 …

}

xpy应为什么类型呢？由于不知道ft()将如何使用，因此无法预先知道这一点。正确的类型可能是T1、T2或其他类型。例如，T1可能是double，而T2可能是int，在这种情况下，两个变量的和将为double类型。T1可能是short，而T2可能是int，在这种情况下，两个变量的和为int类型。T1还可能是short，而T2可能是char，在这种情况下，加法运算将导致自动整型提升，因此结果类型为int，另处，结构和类可能重载运算符+，这导致问题更加复杂。因此，在C++98中，没有办法声明xpy的类型。

### 2．关键字decltype（C++11）

C++11新增的关键字decltype提供了解决方案。可这样使用该关键字：

int x;

decltype(x) y; // 使y的类型与x相同

给decltype提供的参数可以是表达式，因此在前面的模板函数ft()中，可使用下面的代码：

decltype(x+y) xpy; // 使xpy的类型与x+y相同

xpy = x+y;

另一种方法是，将这两条语句合而为一：

decltype(x+y) xpy = x+y;

因此，可以这样修复前面的模板函数ft()：

template<class T1,class T2>

void ft(T1 x,T2 y)

{

 decltype(x+y) xpy = x+y;

}

decltype比这些示例演示的要复杂些。为确定类型，编译器必须遍历一个核对表。假设有如下声明：

decltype (expression) var;

则核对表的简化版如下。

第一步：如果expression是一个没有用括号括起的标识符，则var的类型与该标识符的类型相同，包括const等限定符：

double x = 5.5;

double y = 7.9;

double &rx = x;

const double \* pd;

decltype(x) w // w为double类型

decltype(rx) u = y // u为double &类型

decltype(pd） v; // v为double \*类型

第二步：如果expression是一个函数调用，则var的类型与函数的返回类型相同：

long indeed(int);

decltype (indeed(3)); // m为int类型

**注意**：并不会实际调用函数。编译器通过查看函数的原型来获悉返回类型，而无需实际调用函数。

第三步：如果expression是一个左值，则var为指向其类型的引用。这好像意味着前面的w应为引用类型，因为x是一个左值。但别忘了，这种情况已经在第一步处理过了。要进入第三步，expression不能是未用括号括起的标识符。那么，expression是什么时将进入第三步呢？一种显而易见的情况是，expression是用括号括起的标识符：

double xx = 4.4;

decltype((xx)) r2 = xx; // r2是double &类型

decltype(xx) w = xx // w是double类型（符合第一步的情况）

顺便说一句，括号并不会改变表达式的值和左值性。例如，下面两条语句等效：

xx = 98.6;

(xx) =98.6; // ()并不影响xx的使用

第四步：如果前面的条件都不满足，则var的类型与expression的类型相同：

int j = 3;

int &k = j;

int &n = j;

decltype(j+6) il; // il是int类型

decltype(100L) i2; // i2是long类型

decltype(k+n) i3; // i3是int类型

请注意，虽然k和n都是引用，但表达式k+n不是引用；它是两个int的和，因此类型为int。

如果需要多次声明，可结合使用typedef和decltype：

template<class T1, class T2>

void ft(T1 x．T2 y)

{

 typedef decltype(x+y) xytype;

 xytype xpy = x+y;

 xytype arr[10];

 xytype & rxy = arr[2];

 …

}

### 3．另一种函数声明语法（C++11后置返回类型）

有一个相关时问题是decltype本身无法解决的。请看下面这个不完整的模板函数：

template<class T1, class T2>

?type? gt(T1 x,T2 y)

{

 …

 return x+y;

}

同样，无法预先知道将x和y相加得到的类型。好像可以将返回类型设置为decltype(x+y)，但不幸的是，此时还未声明参数x和y，它们不在作用域内（编译器看不到它们，也无法使用它们）。必须在声明参数后使用decttype。为此，C++新增了一种声明和定义函数的语法。下面使用内置类型来说明这种语法的工作原理。对于下面的原型：

double h(int x,float y);

使用新增的语法可编写成这样：

auto h(int x,float y)->double;

这将返回类型移到了参数声明后面。->double被称为**后置返回类型（trailing return type）**。其中auto是一个占位符，表示后置返回类型提供的类型，这是C++11给auto新增的一种角色。这种语法也可用于函数定义：

auto h(int x,float y)->double

{ /\* function body \*/};

通过结合使用这种语法和decltype，便可给gt()指定返回类型，如下所示：

template<class T1, class T2>

auto gt(T1 x,T2 y)->decltype(x+y)

{

 …

 return x+y;

}

现在，decltype在参数声明后面，因此x和y位于作用域内，可以使用它们。