chapter8 函数探幽（下）

1）函数重载（函数多态）：

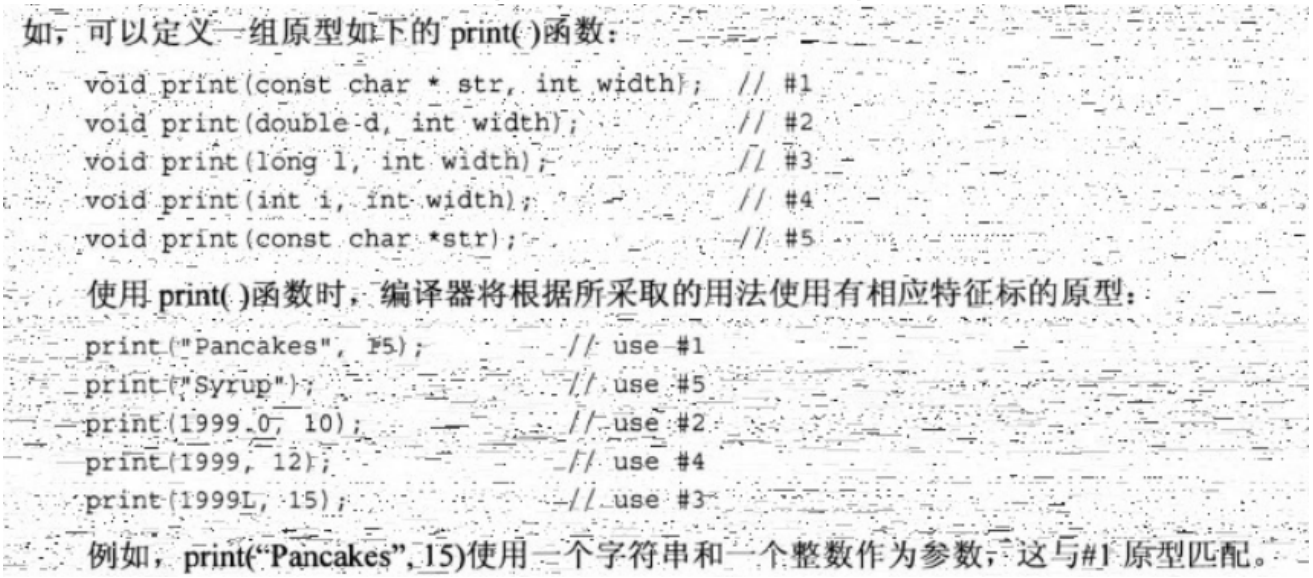
（1）功能：能让我们使用多个重名的函数，我们通过函数重载设计一系列函数——它们完成相同的工作，但使用不同的参数列表

（2）关键：函数的参数列表（也叫作：函数的特征标 function signature）

> poi：是特征标，而不是函数类型使得可以对函数进行重载！

```
long gronk(int n , float m);
double gronk(int n , float m);  // not allowed!
// 这两者是互斥的!
```

（3）要点：

> 【1】c++允许定义名称相同的函数，条件是特征标不同 [如果参数 数目/类型 不同，则特征标不同]
>
> 
> 如，可以定义一组原型如下的 print( )函数：
> ```
> void print(const char * str, int width);   // #1
> void print(double d, int width);           // #2
> void print(long l, int width);             // #3
> void print(int i, int width);              // #4
> void print(const char *str);               // #5
> ```
> 使用 print( )函数时，编译器将根据所采取的用法使用有相应特征标的原型：
> ```
> print("Pancakes", 15);      // use #1
> print("Syrup");             // use #5
> print(1999.0, 10);          // use #2
> print(1999, 12);            // use #4
> print(1999L, 15);           // use #3
> ```
> 例如，print(“Pancakes”, 15)使用一个字符串和一个整数作为参数，这与#1 原型匹配。
>
> 编译器会自动根据 所采取的用法· 使用 有相应特征标 的原型

> 【2】使用被重载的函数时，需要在函数调用中提供正确的参数类型
> point：如果某函数没有找到原型匹配，它也不会停止使用其中的某个函数，因为c++会**尝试**使用标准类型，转换强制进行匹配

```
eg1:
（1）已经有三个函数声明：
void print(const char* str , int width);  // 1)
void print(double d , int width);         // 2)
void print(const char* str);              // 3)

（2）现在要求实现：
unsigned int year = 3210;
print(year,6);

（3）分析：
没有找到完全合适的原型，2)最为接近，因此c++将year强制转化成double类型

eg2:
（1）已经有三个函数声明：
void print(int i , int width);            // 1)
void print(double d , int width);         // 2)
void print(long long l , int width);      // 3)

（2）现在要求实现：
unsigned int year = 3210;
print(year,6);

（3）分析：
```

```
有三个函数将数字作为第一个参数的原型，因此有三种转换year的方式！
这种情况下，c++不知道该咋办。
c++将拒绝这种函数调用，并将其视为错误！
```

> 【3】编译器检查函数特征标时，将把类型引用和类型本身视为同一个特征标

```
double cube(int x);
double cube(int &x);
参数x 在这里与原型 int x 和 int &x 都可以匹配，因此编译器无法确定究竟应当使用哪个原型！
```

> 【4】要区分 const 与 非const 变量:
> 将非const值 赋给 const变量，则合法；反之是非法的！

分析:
dribble( ): 有两个原型，一个用于const指针，另一个用于常规指针，编译器根据实参是否是const来决定使用哪个原型！
dabble( ): 只与带非const参数的调用匹配！
drivel( ): 虽然表面上的原型只有含const的，但它可以与 带const 或 非const 的参数调用进行匹配！【reason：上文 黄字段】

2）函数模板（通用编程）：

（1）简介：函数模板是通用的函数描述，它们通过泛式来定义函数，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

（2）常规模板写作格式：

```
template <typename AnyType> // 也可以换成：template <class AnyType>
void hbx_swap(AnyType &a , AnyType &b)
{
        AnyType tmp;
        tmp = a;
        a = b;
        b = tmp;
}
```

> ps：模板并不创建任何函数，而只是告诉编译器如何定义函数！

eg1：**基本例证展示**

```
// funtemp.cpp -- using a function template
#include <iostream>  // function template prototype
template <typename T>  // or 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);  // generates 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);  // generates void Swap(double &, double &)
    cout << "Now x, y = " << x << ", " << y << ".\n";
    // cin.get();
    return 0;
}

// function template definition
template <typename T>  // or class T
void Swap(T &a, T &b)
{
    T temp;   // temp a variable of type T
    temp = a;
    a = b;
    b = temp;
}
```

> ps：函数模板只是格式变得简洁，并不能缩短可执行程序!

eg2：**被重载的模板特征标必须不同**

```
// twotemps.cpp -- using overloaded template functions
#include <iostream>
template <typename T>     // original template
void Swap(T &a, T &b);

template <typename T>     // new template
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);              // matches original template
    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);        // matches new template
    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;
}
```

这个例子里：
原模板特征标是 (T& , T&)，而新模板特征标是(T[ ], T[ ], int)
在后一个模板中，最后一个参数的类型是具体类型(int)，而不是泛式！
并非所有的模板参数都必须是模板参数类型！

> ps：并非所有的模板参数都必须是模板参数类型

(3) 常规模板的局限性：

假设有如下模板函数：

```
template <class T>      // or 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（运算符重载将在第 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。

(4) 显示具体化模板函数：

【1】存在的目的：

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。

【2】显式具体化写作格式：

```
（1）非模板函数：
void swap(job& , job&)

（2）常规模板函数：
template <typename T>
void swap(T& , T&)

（3）显式具体化模板函数：
template <> void swap(job& , job&)
```

【3】性能比较 (优先级顺序)： 非模板函数 > 常规模板函数 > 显式具体化模板函数

eg：

```
// twoswap.cpp -- specialization overrides a template
#include <iostream>

//1) template_func:
template <typename T>
void Swap(T &a, T &b);

struct job {
    char name[40];
    double salary;
    int floor;
};

//2) explicit specialization:
template <> void Swap<job>(job &j1, job &j2);

//3) display_func:
void Show(job &j);

int main()
{
    using namespace std;
    cout.precision(2);
    cout.setf(ios::fixed, ios::floatfield);

    //1) 值交换部分:
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper:\n";
    Swap(i,j);    // generates void Swap(int &, int &)
    cout << "Now i, j = " << i << ", " << j << ".\n";
    //2) 结构交换（全换 & 部分换）
```

```
    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); // uses 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)    // general version
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}

// swaps just the salary and floor fields of a job structure
template <> void Swap<job>(job &j1, job &j2)  // specialization
{
    double t1;
    t1 = j1.salary;
    j1.salary = j2.salary;
    j2.salary = t1;

    int t2;
    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;
}

结果:
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
```

【4】实例化与具体化:

(1) 隐式实例化、显式实例化、显式具体化 统称为**具体化(specialization)**

(2) 区分显式实例化 / 显式具体化:

```
[1]显式实例化:
template Func_Class Func_name<目标类型>(参数列表);     // explicit instantiation

eg: template void swap<int>(int,int);
=>  使用swap()模板生成int类型的函数定义

[2]显式具体化:
template <> Func_Class Func_name<目标类型>(参数列表);  // explicit spacialization
也可以省略<目标类型> :
template <> Func_Class Func_name(参数列表);           // explicit spacialization
```

ps：试图在同一个文件（或转换单元）中使用同一种类型的**显式实例化**and**显式具体化**将出错！

（3）展示使用方式：

```
template <class T>
T Add(T a, T b)     // pass by value
{
    return a + b;
}
...
int m = 6;
double x = 10.2;
cout << Add<double>(x, m) << endl;  // explicit instantiation
```

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。

对于前例中的swap()函数：

```
int m = 5;
double x  = 14.3;
Swap<double>(m, x);  // almost works
```

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。

（4）综合分析：

```
...
template <class T>
void Swap (T &, T &);  // template prototype

template <> void Swap<job>(job &, job &);   // explicit specialization for job
int main(void)
{
  template void Swap<char>(char &, char &); // explicit instantiation for char
  short a, b;
  ...
  Swap(a,b);     // implicit template instantiation for short
  job n, m;
  ...
  Swap(n, m);    // use explicit specialization for job
  char g, h;
  ...
  Swap(g, h);  // use explicit template instantiation for char
  ...
}
```

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。