**Chapter 8 Adventures in Functions**

* **C++ Inline Functions**

Inline functions are a C++ enhancement designed to speed up programs. In an inline function, the compiled code is “in line” with the other code in the program. That is, the compiler replaces the function call with the corresponding function code. With inline code, the program doesn’t have to jump to another location to execute the code and then jump back. Inline functions thus run a little faster than regular functions, but they come with a memory penalty.

**You should be selective about using inline functions**. If the time needed to execute the function code is long compared to the time needed to handle the function call mechanism, then the time saved is a relatively small portion of the entire process. **If the code execution time is short, then an inline call can save a large portion of the time** used by the non-inline call. **On the other hand, you are now saving a large portion of a relatively quick process, so the absolute time savings may not be that great unless the function is called frequently**.

To use this feature, you must take at least one of two actions:

1. Preface the function declaration with the keyword inline.
2. Preface the function definition with the keyword inline.

**A common practice is to omit the prototype and to place the entire definition (meaning the function header and all the function code) where the prototype would normally go.**

**The compiler does not have to honor your request to make a function inline**. It might decide the ***function is too large*** or notice that ***it calls itself (recursion is not allowed or indeed possible for inline functions),*** or the feature might not be turned on or implemented for your particular compiler.

// an inline function definition

**inline** double square(double x) { return x \* x; }

Note that the entire definition is on one line. That’s not required, but if the definition doesn’t fit on one or two lines (assuming you don’t have lengthy identifiers), the function is probably a poor candidate for an inline function.

**Inline Versus Macros**

#define SQUARE(X) X\*X

a = SQUARE(5.0); is replaced by a = 5.0\*5.0;

b = SQUARE(4.5 + 7.5); is replaced by b = 4.5 + 7.5 \* 4.5 + 7.5;

d = SQUARE(c++); is replaced by d = c++\*c++;

* **Reference Variables**

C++ adds a new compound type to the language—the reference variable. **A reference is a name that acts as an alias, or an alternative name, for a previously defined variable**.

**The main use for a reference variable** is as a formal argument to a function. If you use a reference as an argument, **the function works with the *original data* instead of with a copy**. **References provide a convenient alternative to pointers** for processing large structures with a function, and they are essential for designing classes.

* **Creating a Reference Variable**

To make rodents an alternative name for the variable rats, you could do the following:

int rats;

**int & rodents = rats**; // makes rodents an alias for rats, ***int &* means reference-toint.**

The reference declaration allows you to use rats and rodents interchangeably; **both refer to the same value and the same memory location**.

#include <iostream>

int main()

{

using namespace std;

int rats = 101;

int & rodents = rats; // rodents is a reference

cout << "rats = " << rats;

cout << ", rodents = " << rodents << endl;

rodents++;

cout << "rats = " << rats;

cout << ", rodents = " << rodents << endl;

cout << "rats address = " << &rats;

cout << ", rodents address = " << &rodents << endl;

return 0;

}

书上给出的测试结果是：

rats = 101, rodents = 101

rats = 102, rodents = 102

rats address = 0x0065fd48, rodents address = 0x0065fd48

The ‘*rodents++’* operation increments **a single variable** for which **there are two names**.

It is necessary to **initialize the reference when you declare it**; you **can’t** declare the reference and then assign it a value later the way you can with a pointer:

int rat;

int & rodent;

**rodent = rat; // No, you can't do this. But with pointer, you can initialize it later.**

**A reference is rather like a const pointer**; **you have to initialize it when you create it**, and when a reference pledges its allegiance to a particular variable, it sticks to its pledge. That is,

int & rodents = rats;

is, in essence, a disguised notation for something like this:

**int \* const** pr = &rats;

Here, the reference rodents plays the same role as the expression \*pr.

#include <iostream>

int main()

{

using namespace std;

int rats = 101;

**int & rodents = rats; // rodents is a reference**

cout << "rats = " << rats;

cout << ", rodents = " << rodents << endl;

cout << "rats address = " << &rats;

cout << ", rodents address = " << &rodents << endl;

**int bunnies = 50;**

**rodents = bunnies; // can we change the reference?**

cout << "bunnies = " << bunnies;

cout << ", rats = " << rats;

cout << ", rodents = " << rodents << endl;

cout << "bunnies address = " << &bunnies;

cout << ", rodents address = " << &rodents << endl;

return 0;

}

自己电脑上的测试结果：

momo@HMI:~/C++PrimerPlus/Chapter8$ ./secref

rats = 101, rodents = 101

rats address = **0x7fffaabf3d80**, rodents address = 0x7fffaabf3d80

bunnies = 50, rats = 50, rodents = 50

bunnies address = 0x7fffaabf3d84, **rodents address = 0x7fffaabf3d80**

momo@HMI:~/C++PrimerPlus/Chapter8$

对reference rodents的第二个assignment只改变了rodents以及rats的值，他们的地址都没有变.

**In short, you can set a reference by an initializing declaration, not by assignment.**

* **References as Function Parameters**

Most often, references are used as function parameters, making a variable name in a function an alias for a variable in the calling program. This method of passing arguments is called passing by reference. Passing by reference allows a called function to access variables in the calling function. C++’s addition of the feature is a break from C, which only passes by value.

void swapr(int & a, int & b); // a, b are aliases for ints

int wallet1 = 300;

int wallet2 = 350;

**swapr(wallet1, wallet2)**; // pass variables, wallet1 = $350 wallet2 = $300

void swapr(**int & a, int & b**) // use references

{

int temp;

temp = **a**; // use a, b for values of variables

**a** = **b**;

**b** = temp;

}

上面这个传递引用看起来和传递value是一样的. **The only way you can tell that *swapr()* passes by reference is by looking at the prototype or the function definition**.

If your intent is that a function use the information passed to it **without modifying the information, and if you’re using a reference**, you should use a constant reference. Here, for example, you should use const in the function prototype and function header:

double refcube(**const** double &ra);

If you do this, the compiler generates an error message when it finds code altering the value of *ra*. Reference arguments become useful with larger data units, such as structures and classes, as you’ll soon see.

* **Temporary Variables, Reference Arguments, and const**

Currently, C++ permits that, **when argument of a function is *const* reference, and the actual argument doesn’t match that reference argument**, C++ can generate a temporary variable.

**When is a temporary variable created?** Provided that the **reference parameter is a const**, the compiler generates a temporary variable in two kinds of situations:

1. When the actual argument is the correct type but isn’t an lvalue
2. When the actual argument is **of the wrong type**, **but it’s of a type that can be converted to the correct type;**

Now, to return to our example, suppose you redefine refcube() so that it has a constant reference argument:

double refcube(const double &ra)

{

return ra \* ra \* ra;

}

Next, consider the following code:

double side = 3.0;

double \* pd = &side;

double & rd = side;

long edge = 5L;

double lens[4] = { 2.0, 5.0, 10.0, 12.0};

double c1 = refcube(side); // ra is side

double c2 = refcube(lens[2]); // ra is lens[2]

double c3 = refcube(rd); // ra is rd is side

double c4 = refcube(\*pd); // ra is \*pd is side

double c5 = refcube(edge); **// ra is temporary variable**

double c6 = refcube(7.0); **// ra is temporary variable**

double c7 = refcube(side + 10.0); **// ra is temporary variable**

*Edge* is a variable, it is of the wrong type. A reference to a double can’t refer to a long. The arguments 7.0 and side + 10.0, on the other hand, are the right type, but they are not named data objects. In each of these cases, the compiler generates a temporary, anonymous variable and makes *ra* refer to it. These temporary variables last for the duration of the function call, but then the compiler is free to dump them.

**So why is this behavior okay for *constant* references but not otherwise?**

如果你的函数是要修改reference参数对应的原始值, 那么你不会把他定义为const, 如果这种情况下还对参数创建一个temporary variable, 那么最终修改的结果也是针对temporary variable的, 不会涉及参数对应的原始值, 这是这种修改却毫无意义;

In short, if the intent of a function with reference arguments is to modify variables passed as arguments, situations that create temporary variables thwart that purpose.

Now think about the refcube() function. **Its intent is merely to use passed values, not to modify them, so temporary variables cause no harm and make the function more general in the sorts of arguments it can handle.** Therefore, if the declaration states that a reference is const, C++ generates temporary variables when necessary. In essence, a C++ function with a const reference formal argument and a non-matching actual argument mimics the traditional passing by value behavior, guaranteeing that the original data is unaltered and using a temporary variable to hold the value.

C++11 introduces a second kind of reference, called an rvalue reference, that can refer to an rvalue. It’s declared using &&:

**double &&** rref = std::sqrt(36.00);// not allowed for double &

double j = 15.0;

**double &&** jref = 2.0\* j + 18.5; // not allowed for double &

std::cout << rref << '\n'; // display 6.0

std::cout << jref << '\n'; // display 48.5;

The rvalue reference was introduced mainly to help library designers provide more efficient implementations of certain operations.

* **Using References with a Structure**

Indeed, references were introduced primarily for use with types of structures and classes, not for use with the basic built-in types.

struct free\_throws

{

**std::**string name;

int made;

int attempts;

float percent;

};

Then a function using a reference to this type could be prototyped as follows:

void set\_pc(**free\_throws &** ft); // use a reference to a structure

If the intent is that the function doesn’t alter the structure, use const:

void display(**const** free\_throws & ft); // don't allow changes to structure

The program below does exactly these things. It also adds an interesting twist by having a function **return a reference to the structure.**

#include <iostream>

**#include <string>**

**struct free\_throws**

{

std::string name;

int made;

int attempts;

float percent;

};

void display(**const** free\_throws & ft);

void set\_pc(free\_throws & ft);

**free\_throws & accumulate(free\_throws & target, const free\_throws & source);**

int main()

{

// partial initializations – remaining members set to 0

free\_throws one = {"Ifelsa Branch", 13, 14};

free\_throws two = {"Andor Knott", 10, 16};

free\_throws three = {"Minnie Max", 7, 9};

free\_throws four = {"Whily Looper", 5, 9};

free\_throws five = {"Long Long", 6, 14};

**free\_throws team** = {"Throwgoods", 0, 0};

// no initialization

**free\_throws dup;**

set\_pc(one);

display(one);

accumulate(team, one);

display(team);

// use return value as argument

**display(accumulate(team, two));**

**accumulate(accumulate(team, three), four);**

display(team);

// use return value in assignment

dup = accumulate(team,five);

std::cout << "Displaying team:\n";

display(team);

std::cout << "Displaying dup after assignment:\n";

display(dup);

set\_pc(four);

// ill-advised assignment

accumulate(dup,five) = four;

std::cout << "Displaying dup after ill-advised assignment:\n";

display(dup);

return 0;

}

void display(const free\_throws & ft)

{

using std::cout;

cout << "Name: " << ft.name << '\n';

cout << " Made: " << ft.made << '\t';

cout << "Attempts: " << ft.attempts << '\t';

cout << "Percent: " << ft.percent << '\n';

}

void set\_pc(free\_throws & ft)

{

if (ft.attempts != 0)

ft.percent = 100.0f \*float(ft.made)/float(ft.attempts);

else

ft.percent = 0;

}

**free\_throws &** accumulate(free\_throws & target, const free\_throws & source)

{

target.attempts += source.attempts;

target.made += source.made;

set\_pc(target);

**return target;**

}

The program begins by initializing several structure objects. Recall that if there are fewer initializers than members, the remaining members (just the percent members in this case) are set to 0.

set\_pc()也可以这么写：

set\_pcp(&one); // using pointers instead - &one instead of one

...

void set\_pcp(free\_throws \* pt)

{

if (pt->attempts != 0)

pt->percent = 100.0f \*float(pt->made)/float(pt->attempts);

else

pt->percent = 0;

}

display() displays the contents of the structure without altering them, the function uses a const reference parameter. In this case, one could have passed the structure by value, **but using a reference is more economical in time and memory than making a copy of the original structure**.

**For function accumulate()**, if the return type were declared *free\_throws* instead of *free\_throws &*, **the same return statement would return *a copy of target*** (and hence a copy of team). **But the return type is a reference**, so that means **the return value is the *original team* object first passed to accumulate().**

* **Why Return a Reference?**

dup = accumulate(team,five);

If accumulate() returned a structure instead of a reference to a structure, this could involve copying the entire structure to a temporary location and then copying that copy to dup. **But with a reference return value, team is copied directly to dup, a more efficient approach**.

* **Being Careful About What a Return Reference Refers To**

The single most important point to remember when returning a reference is to avoid returning a reference to a memory location that ceases to exist when the function terminates.

const free\_throws & clone2(free\_throws & ft)

{

free\_throws newguy; // first step to big error

newguy = ft; // copy info

return newguy; // return reference to copy

}

This has the unfortunate effect of returning a reference to a temporary variable (newguy) **that passes from existence as soon as the function terminates**.

**The simplest way** to avoid this problem is to return a reference that was passed as an argument to the function. This, for example, is what accumulate() does in above example.

**A second method** is to use new to create new storage.

const free\_throws & clone(free\_throws & ft)

{

free\_throws \* pt;

\*pt = ft; // copy info

return \*pt; // return reference to copy

}

The first statement creates a nameless *free\_throws* structure. The pointer pt points to the structure, so \*pt is the structure. **The code appears to return the structure, but the function declaration indicates that the function really returns a reference to this structure.** You could then use the function this way:

free\_throws & jolly = clone(three);

This makes jolly a reference to the new structure.

**There is a problem with this approach**: You should use **delete** to free memory allocated by ***new*** when the memory is no longer needed. A call to clone() conceals the call to new, making it simpler to forget to use delete later. The auto\_ptr template or, better, the C++11 unique\_ptr discussed in Chapter 16,“The string Class and the Standard Template Library,” can help automate the deletion process.

* **Why Use const with a Reference Return?**

Suppose you want to use a reference return value but don’t want to permit behavior such as assigning a value to accumulate(). Just make the return type a const reference:

const free\_throws &

accumulate(free\_throws & target, const free\_throws & source);

The return type now is const, hence a nonmodifiable lvalue. Therefore, **the assignment no longer is allowed**:

accumulate(dup,five) = four; **// not allowed for const reference return**

What about the other function calls in the program? With a const reference return type, the following statement would still be allowed:

display(accumulate(team, two));

That’s because the formal parameter for display() also is type const free\_thows &. But the following statement would not be allowed because the first formal parameter for accumulate() is not const:

accumulate(accumulate(team, three), four);

* **Using References with a Class Object**

The usual C++ practice for passing class objects to a function is to use references.

Let’s look at an example that uses the string class and illustrates some different design choices, some of them bad. The general idea is to create a function that adds a given string to each end of another string. Listing below provides three functions that are intended to do this.

#include <iostream>

#include <string>

using namespace std;

**string** version1(const string & s1, const string & s2);

**const string** & version2(string & s1, const string & s2); // has side effect

**const string** & version3(string & s1, const string & s2); // bad design

int main()

{

string input;

string copy;

string result;

cout << "Enter a string: ";

**getline(cin, input);**

**copy = input;**

cout << "Your string as entered: " << input << endl;

**result = version1(input, "\*\*\*");**

cout << "Your string enhanced: " << result << endl;

cout << "Your original string: " << input << endl;

result = version2(input, "###");

cout << "Your string enhanced: " << result << endl;

cout << "Your original string: " << input << endl;

cout << "Resetting original string.\n";

input = copy;

result = version3(input, "@@@");

cout << "Your string enhanced: " << result << endl;

cout << "Your original string: " << input << endl;

return 0;

}

string version1(**const string & s1**, **const string & s2**)

{

string temp;

temp = s2 + s1 + s2;

return temp;

}

const string & version2(string & s1, const string & s2) // has side effect

{

s1 = s2 + s1 + s2;

// safe to return reference passed to function

return s1;

}

const string & version3(string & s1, const string & s2) // bad design

{

string temp;

temp = s2 + s1 + s2;

// unsafe to return reference to local variable

return temp;

}

书上给的测试结果：

Enter a string: It’s not my fault.

Your string as entered: It's not my fault.

Your string enhanced: \*\*\*It's not my fault.\*\*\*

Your original string: It's not my fault.

Your string enhanced: ###It's not my fault.###

Your original string: ###It's not my fault.###

Resetting original string.

**At this point the program crashed.**

上面程序里的string是class对象, 这个class支持’+’操作符.

首先, 看函数version1. 它的参数是两个string的reference, 相比于直接用string做参数, 用引用省去了copy参数to temporary variable的开销, 效率更高.

The temp object is a new object, local to the version1() function, and it ceases to exist when the function terminates. **Thus, returning temp as a reference won’t work**, **so the function type is string**. This means the contents of temp will be copied to a temporary return location. Then, in main(), the contents of the return location are copied to the string named result.

**You may have noticed a rather interesting fact** about the version1() function: **Both formal parameters (s1 and s2) are type const string &, but the actual arguments (input and "\*\*\*") are type string and const char \***, respectively. Because input is type string, there is no problem having s1 refer to it. **But how is it that the program accepts passing a pointer-to-char argument to a string reference?**

**Two things are going on here**:

1. String class defines a char \*-to-string conversion, which makes it possible to initialize a string object to a C-style string;
2. The second is a property of ***const*** reference formal parameters that is discussed earlier in this chapter. Suppose the actual argument type doesn’t match the reference parameter type but can be converted to the reference type. Then the program creates a temporary variable of the correct type, initializes it to the converted value, and passes a reference to the temporary variable. Earlier this chapter you saw, for instance, that a const double & parameter can handle an int argument in this fashion. **Similarly, a *const* string & parameter can handle a char \* or const char \* argument in this fashion.**

The convenient outcome of this is that if the formal parameter is type ***const string &,*** the actual argument used in the function call can be a **string object** or a **C-style string**, such as a **quoted string literal**, a **null-terminated array of char**, or a **pointer variable that points to a char**. Hence the following works fine:

result = version1(input, "\*\*\*");

The version2() function doesn’t create a temporary string. Instead, it directly alters the original string:

const string & version2(string & s1, const string & s2) // has side effect

{

s1 = s2 + s1 + s2;

// safe to return reference passed to function

return s1;

}

Because s1 is a reference to an object (input) in main(), it’s safe to return s1 as a reference.

Version3()is a reminder of what not to do:

const string & version3(string & s1, const string & s2) // bad design

{

string temp;

temp = s2 + s1 + s2;

// unsafe to return reference to local variable

return temp;

}

It has the fatal flaw of returning a reference to a variable declared locally inside version3().This function compiles (with a warning), but the program crashes when attempting to execute the function.

* **Another Object Lesson: Objects, Inheritance, and References**

The language feature that makes it possible to pass features from one class to another is called **inheritance**, and Chapter 13,”Class Inheritance”, discusses this feature in detail. In brief, **ostream is termed a base class** (because the ofstream class is based on it) and **ofstream is termed a *derived* class** (because it is derived from ostream). A derived class inherits the base class methods, which means that an ofstream object can use base class features such as the precision() and setf() formatting methods.

**Another aspect of inheritance** is that **a *base* class reference can refer to a *derived* class object without requiring a type cast**. The practical upshot of this is that you can define a function having a base class reference parameter, and that function can be used with base class objects and also with derived objects.

For example, a function with a type **ostream &** parameter can accept an ostream object, such as cout, or an **ofstream object**, such as you might declare, equally well.

书中在这里举了个ostream和ofstream的例子, 但没看懂, 以后再看.

* **When to Use Reference Arguments**

There are two main reasons for using reference arguments:

1. To allow you to **alter a data** object in the calling function;
2. To **speed up a program by passing a reference instead of an entire data object**;

These two reasons are the same reasons you might have for using a pointer argument. **This makes sense because reference arguments are really just a different interface for pointer-based code**.

* **A function uses passed data without modifying it:**

1. If the data object is small, such as a built-in data type or a small structure, pass it by value;
2. **If the data object is an array, use a pointer because that’s your only choice**. Make the pointer a pointer to const;
3. If the data object is a good-sized structure, use a const pointer or a const reference to increase program efficiency. You save the time and space needed to copy a structure or a class design. Make the pointer or reference const; 因为const也可以接受非const的参数, 只是不能再修改;
4. If the data object is a class object, use a const reference. The semantics of class design often require using a reference, which is the main reason C++ added this feature. Thus, the standard way to pass class object arguments is by reference;

* **A function modifies data in the calling function:**

1. If the data object is a built-in data type, use a pointer. If you spot code like **fixit(&x),** where x is an int, it’s pretty clear that this function intends to modify x;
2. If the data object is an array, use your only choice: a pointer;
3. If the data object is a structure, use a reference or a pointer;
4. If the data object is a class object, use a reference.

* **Default Arguments**

A default argument is a value that’s used automatically if you omit the corresponding actual argument from a function call.

**How do you establish a default value? You must use the function prototype.** Because the compiler looks at the prototype to see how many arguments a function uses, the function prototype also has to alert the program to the possibility of default arguments.

char \* left(const char \* str, int n = 1);

You want the function to return a new string, so its type is char\*, or pointer-to-char. You want to leave the original string unaltered, so you use the const qualifier for the first argument. You want n to have a default value of 1, so you assign that value to n. A default argument value is an initialization value. Thus, the preceding prototype initializes n to the value 1. If you leave n alone, it has the value 1, but if you pass an argument, the new value overwrites the 1.

When you use a function with an argument list, you must add defaults from right to left. That is, you can’t provide a default value for a particular argument unless you also provide defaults for all the arguments to its right:

int harpo(int n, int m = 4, int j = 5); // VALID

int chico(int n, int m = 6, int j); // INVALID

int groucho(int k = 1, int m = 2, int n = 3); // VALID

For example, the harpo() prototype permits calls with one, two, or three arguments:

beeps = harpo(2); // same as harpo(2,4,5)

beeps = harpo(1,8); // same as harpo(1,8,5)

beeps = harpo (8,7,6); // no default arguments used

* **Function Overloading**

Function polymorphism is a neat C++ addition to C’s capabilities. Whereas default arguments let you call the same function by using varying numbers of arguments, **function** **polymorphism**, also **called function overloading**, **lets you use multiple functions sharing the same name**.

**The key to function overloading is a function’s argument list, also called the function signature**. If two functions use the same number and types of arguments in the same order, they have the same signature; the variable names don’t matter. **C++ enables you to define two functions by the same name, provided that the functions have different signatures.**

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

When you then use a print() function, the compiler matches your use to the prototype that has the same signature:

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

When you use overloaded functions, you need to be sure you use the proper argument types in the function call.

unsigned int year = 3210;

print(year, 6); // ambiguous call

Which prototype does the print() call match here? It doesn’t match any of them! A lack of a matching prototype doesn’t automatically rule out using one of the functions because C++ will try to use standard type conversions to force a match. If, say, the only print() prototype were #2, the function call print(year, 6) would convert the year value to type double. **But in the earlier code there are three prototypes that take a number as the first argument, providing three different choices for converting year. Faced with this ambiguous situation, C++ rejects the function call as an error.**

Some signatures that appear to be different from each other nonetheless can’t coexist. For example, consider these two prototypes:

double cube(double **x**);

double cube(double **& x**);

You might think this is a place you could use function overloading because the function signatures appear to be different. But consider things from the compiler’s standpoint. Suppose you have code like this:

cout << cube(x);

**The x argument matches both the double x prototype and the double &x prototype**. The compiler has no way of knowing which function to use.

The function-matching process does **discriminate between const and non-const variables**.

void dribble(char \* bits); // overloaded

void dribble (const char \*cbits); // overloaded

void dabble(char \* bits); // not overloaded

void drivel(const char \* bits); // not overloaded

const char p1[20] = "How's the weather?";

char p2[20] = "How's business?";

dribble(p1); // dribble(const char \*);

dribble(p2); // dribble(char \*);

dabble(p1); // **no match, 当函数形参不是const时，不能接受const的实参;**

**// 当函数形参是const时，可以接受不是const的实参;**

dabble(p2); // dabble(char \*);

drivel(p1); // drivel(const char \*);

drivel(p2); // drivel(const char \*);

**The reason for this difference in behavior between drivel() and dabble() is that it’s valid to assign a non-const value to a const variable, but not vice versa.**

Keep in mind that the **signature, *NOT the function type*, enables function overloading**. For example, the following two declarations are incompatible:

long gronk(int n, float m); // same signatures,

double gronk(int n, float m); // hence not allowed

Therefore, C++ doesn’t permit you to overload gronk() in this fashion. You can have different return types, but only if the signatures are also different.

* **Overloading Reference Parameters**

Class designs and the STL often use reference parameters, and it’s useful to know how overloading works with different reference types.

void **sink**(double & r1); // matches modifiable lvalue

void **sank**(const double & r2); // **matches modifiable or const lvalue, rvalue**

void **sunk**(double && r3); // **matches rvalue**

The lvalue reference parameter r1 matches a modifiable lvalue argument, such as a double variable. The const lvalue reference parameter r2 matches a modifiable lvalue argument, a const lvalue argument, and an rvalue argument, such as the sum of two double values. Finally, the rvalue reference r3 matches an rvalue. **Note how r2 can match the same sort of arguments that r1 and r3 match**. This raises the question of what happens when you overload a function on these three types of parameters. **The answer is that the more exact match is made**:

void staff(double & rs); // matches modifiable lvalue

voit staff(const double & rcs); // matches rvalue, const lvalue

void **stove**(double & r1); // matches modifiable lvalue

void **stove**(const double & r2); // **matches const lvalue**

void **stove**(double && r3); // matches rvalue

This allows you to customize the behavior of a function based on the lvalue, const, or rvalue nature of the argument:

double x = 55.5;

const double y = 32.0;

stove(x); // calls stove(double &)

stove(y); // calls stove(const double &)

stove(x+y); // calls stove(double &&)

If, say, you omit the stove(double &&) function, then stove(x+y) will call the stove(**const** double &) function instead.

**估计这里的意思是说,** 在没有继承时, const本身可以接收非const, const, 以及rvalue; 一旦有了继承, 如果其他几个函数里定义了针对非const及rvalue的函数, 那么用const做参数的那个函数就只接受const参数, 不再对非const以及rvalue具有作用.

#include <iostream>

unsigned long left(unsigned long num, unsigned ct);

char \* left(const char \* str, int n = 1);

int main()

{

using namespace std;

char \* trip = "Hawaii!!"; // test value

unsigned long n = 12345678; // test value

int i;

char \* temp;

for (i = 1; i < 10; i++)

{

cout << left(n, i) << endl;

temp = left(trip,i);

cout << temp << endl;

**delete [] temp**; // point to temporary storage

}

return 0;

}

// This function returns the first ct digits of the number num.

**unsigned long left(unsigned long num, unsigned ct)**

{

unsigned digits = 1;

unsigned long n = num;

if (ct == 0 || num == 0)

return 0; // return 0 if no digits

while (n /= 10)

digits++;

if (digits > ct)

{

ct = digits - ct;

while (ct--)

num /= 10;

return num; // return left ct digits

}

else // if ct >= number of digits

return num; // return the whole number

}

// This function returns a pointer to a new string

char \* left(const char \* str, int n)

{

if(n < 0)

n = 0;

char \* p = **new** char[n+1];

int i;

for (i = 0; i < n && str[i]; i++)

p[i] = str[i]; // copy characters

while (i <= n)

p[i++] = '\0'; // set rest of string to '\0'

return p;

}

* **Function Templates**

A function template is a generic function description; that is, it defines a function in terms of a generic type for which a specific type, such as int or double, can be substituted. By passing a type as a parameter to a template, you cause the compiler to generate a function for that particular type. Because templates let you program in terms of a generic type instead of a specific type, the process is sometimes termed generic programming.

Function templates enable you to define a function in terms of some arbitrary type. For example, you can set up a swapping template like this:

template <typename **AnyType**>

void Swap(**AnyType** &a, **AnyType** &b)

{

**AnyType** temp;

temp = a;

a = b;

b = temp;

}

The keywords *template* and *typename* are obligatory, except that you can use the keyword class instead of *typename*.

**The template does not create any functions**. Instead, it provides the compiler with directions about how to define a function.

Before the C++98 Standard added the keyword typename to the language, C++ used the keyword class in this particular context.That is, you can write the template definition this way:

template <**class** AnyType>

void Swap(AnyType &a, AnyType &b)

{

AnyType temp;

temp = a;

a = b;

b = temp;

}

The C++ Standard treats the two keywords identically when they are used in this context.

#include <iostream>

**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>**

void Swap(T &a, T &b)

{

T temp;

temp = a;

a = b;

b = temp;

}

In Listing above, **you still wind up with two separate function definitions**, just as you would if you defined each function manually. **And the final code doesn’t contain any templates**; it just contains the actual functions generated for the program. The benefits of templates are that they make generating multiple function definitions simpler and more reliable.

* **Overloaded Templates**

You use templates when you need functions that apply the ***same algorithm*** to a variety of types. It might be, however, that not all types would use the same algorithm.

#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;

}

* **Template Limitations**

Suppose you have a template function:

template <class T> // or template <typename T>

void f(T a, T b)

{...}

Often the code makes assumptions about what operations are possible for the type.

a = b; // not be true if type T is a built-in array type

if (a > b) // not true if T is an ordinary structure

// Also the > operator is defined for array names, but because array names are

// addresses, it compares the addresses of the arrays, which may not be what you have // in mind.

T c = a\*b; // not the case if T is an array, a pointer, or a structure

* **Explicit Specializations**

Suppose you define a structure like the following:

struct job

{

char name[40];

double salary;

int floor;

};

Also suppose you want to be able to swap the contents of two such structures. The original template uses the following code to effect a swap:

temp = a;

a = b;

b = temp;

Because C++ allows you to assign one structure to another, this works fine, even if type T is a job structure.

But **suppose you only want to swap the *salary* and *floor* members**, keeping the name members unchanged. **This requires different code, but the arguments to Swap() would be the same as for the first case** (references to two job structures), **so you can’t use template overloading to supply the alternative code**.

However, you can supply a specialized function definition, called an **explicit specialization**, with the required code. **If the compiler finds a specialized definition that exactly matches a function call**, **it uses that definition without looking for templates**.

1. For a given function name, you can have a non template function, a template function, and an explicit specialization template function, along with overloaded versions of all of these;
2. **The prototype and definition for an explicit specialization should be *preceded by template*** <> and should **mention the specialized type by name**;
3. **A specialization overrides the regular template, and a non template function overrides both;**

Here’s how prototypes for swapping type job structures would look for these three forms:

// **non template function** prototype

void Swap(job &, job &);

// **template** prototype

template <typename T>

void Swap(T &, T &);

// **explicit specialization** for the job type

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

As mentioned previously, if more than one of these prototypes is present, the compiler chooses the **non template version** ***over*** **explicit specializations** and **template versions**, and it chooses an **explicit specialization** ***over*** a version generated from a **template**.

template <class T> // template

void Swap(T &, T &);

**// explicit specialization for the job type**

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

int main()

{

double u, v;

...

Swap(u,v); // use template

job a, b;

...

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

}

**The <job> in Swap<job> is optional** because the function argument types indicate that this is a specialization for job. Thus, the prototype can also be written this way:

**template <> void Swap(job &, job &); // simpler form**

**An Example of Explicit Specialization**

#include <iostream>

**// function template**

template <typename T>

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

struct job

{

char name[40];

double salary;

int floor;

};

**// explicit specialization**

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);** **// generates void 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);** **// 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;

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;

}

***Instantiations* and *Specializations***

Keep in mind that including a function template in your code does not in itself generate a function definition. It’s merely a plan for generating a function definition. When the compiler uses the template to generate a function definition for a particular type, the result is termed an instantiation of the template.

The template is not a function definition, but the specific instantiation using int is a function definition. This type of instantiation is termed implicit instantiation because the compiler deduces the necessity for making the definition by noting that the program uses a Swap() function with int parameters.

But now C++ allows for **explicit *instantiation****.* That means you can instruct the compiler to create a particular instantiation—for example, Swap<int>()—directly. **The syntax is to declare the particular variety you want, using the <> notation to indicate the type and prefixing the declaration with the keyword template**:

template void Swap<int>(int, int); // explicit ***instantiation***

A compiler that implements this feature will, upon seeing this declaration, use the Swap() template to generate an instantiation, using the int type. That is, this declaration means “**Use the Swap() template to generate a function definition for the int type**.”

template **<>** void Swap<int>(int &, int &); // explicit ***specialization***

template **<>** void Swap(int &, int &); // explicit **specialization**

The difference is that these last two declarations mean “**Don’t use the Swap() template to generate a function definition. Instead, use a separate, specialized function definition explicitly defined for the int type**.”

These prototypes have to be coupled with their own function definitions. The explicit **specialization declaration has <> after the keyword template, whereas the explicit instantiation omits the <>**.

**It is an error to try to use both an explicit instantiation and an explicit specialization for the same type(s) in the same file, or, more generally, the same translation unit.**

**Explicit instantiations also can be created by using the function in a program**.

* For instance, consider the following:

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**

The template would fail to match the function call Add(x, m) because the template expects both function arguments to be of the same type. **But** using **Add<double>(x, m)** forces the type double instantiation, and the argument *m* is type cast to type double to match the second parameter of the Add<double>(double, double) function.

* What if you do something similar with Swap()?

int m = 5;

double x = 14.3;

Swap<double>(m, x); // almost works

This generates an explicit instantiation for type double. Unfortunately, in this case, **the code won’t work** because the **first formal parameter, being type double &, can’t refer to the type int variable m**.

Implicit instantiations, explicit instantiations, and explicit specializations collectively are termed **specializations**. What they all have in common is that they represent a function definition that uses specific types rather than one that is a generic description.

The addition of the explicit instantiation led to the new syntax of using **template** and **template <>** prefixes in declarations to distinguish between the explicit instantiation and the explicit specialization.

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 defined in the 1st line

}

When the compiler reaches Swap(g,h), it uses the template specialization it already generated when it processed the explicit instantiation.

* **Which Function Version Does the Compiler Pick?**

What with function overloading, function templates, and function template overloading, C++ needs, and has, a well-defined strategy for deciding which function definition to use for a function call.

Consider a case with just one function argument, for example, the following call:

may('B'); // actual argument is type char

First, the compiler rounds up the suspects, which are functions and function templates that have the name may().Then, it finds those that can be called with one argument. For example, the following pass muster because they have the same name and can be used with one argument:

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

Note that **just the signatures and not the return types are considered**.

Two of these candidates **(#4 and #7**), however, are **not viable** because **an integral type** (‘B’ is actually integer) **cannot be converted implicitly** (that is, without an explicit type cast) **to a pointer type.** The remaining template is viable because it can be used to generate a specialization, with T taken as type char.

Next, the compiler has to determine which of the viable functions is best. In general, the ranking from best to worst is this:

1. **Exact match**, with **regular functions outranking templates**;
2. **Conversion by promotion** (for example, the automatic conversions of char and short to int and of float to double);
3. **Conversion by standard conversion** (for example, converting int to char or long to double);
4. User-defined conversions, such as those defined in class declarations;

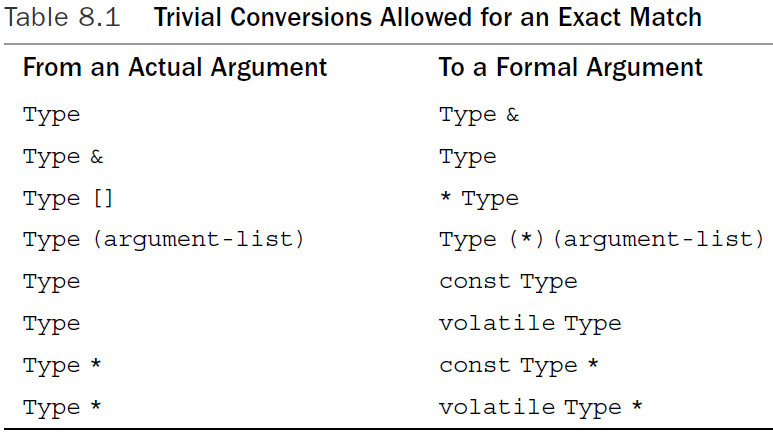
For example:

1. **Function #1 is better than Function #2** because **char-to-int is a promotion** (refer to Chapter 3,“Dealing with Data”), whereas **char-to-float is a standard conversion** (refer to Chapter 3);
2. Functions **#3, #5, and #6 are better than either #1 or #2** because they are **exact matches**;
3. Both **#3 and #5 are better than #6** because #6 is a template;

This analysis raises a couple questions. What is an exact match? **And what happens if you get two of them, such as #3 and #5?** Usually, **as is the case with this example, two exact matches are an error**; but a couple special cases are exceptions to this rule.

**Exact Matches and Best Matches**

C++ allows some “trivial conversions” when making an exact match. Table 8.1 lists them, with Type standing for some arbitrary type. For example, an **int** *actual argument* is an **exact match** to an **int &** *formal parameter*. Note that Type can be something like **char &**, so these rules include converting **char &** to **const char &**.The **Type (argument-list)** entry means that a **function name** as an *actual argument* matches a **function pointer** as a *formal parameter*, as long as both have the **same return type and argument list**. (Remember function pointers from Chapter 7.Also recall that you can pass the name of a function as an argument to a function that expects a pointer to a function.)



Suppose you have the following function code:

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

blot ink = {25, "spots"};

...

recycle(ink);

In that case, **all the following prototypes would be exact matches**:

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 &)

As you might expect, the result of having several matching prototypes is that the compiler cannot complete the overload resolution process. **There is no best viable function**, and the compiler generates an error message, probably using words such as ***ambiguous***.

However, sometimes there can be **overload resolution** even if two functions are an **exact match**.

First, **pointers** and **references** to **non-const data** are preferentially matched to **non-const pointer and reference parameters**. That is, if only Functions #3 and #4 were available in the recycle() example, **#3 would be chosen because ink wasn’t declared as const**. However, **this discrimination between const and non-const applies just to data referred to by *pointers* and *references***. That is, if only #1 and #2 were available, you would get an ambiguity error;

Another case in which one exact match is better than another is when one function is a non template function and the other isn’t. In that case, the **non template is considered better than a template, including explicit specializations**;

If you wind up with two exact matches that both happen to be template functions, the template function that is the more specialized, if either, is the better function.

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

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

template <> void recycle<blot> (blot & t); // **specialization for blot**

...

blot ink = {25, "spots"};

...

recycle(ink); **// use specialization**

The term **most specialized** ***doesn’t*** necessarily imply an **explicit specialization**; more generally, it indicates that ***fewer*** **conversions take place when the compiler deduces what type to use**. For example, consider the following two templates:

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

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

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

blot ink = {25, "spots"};

...

recycle(&ink); // address of a structure

The recycle(&ink) call **matches Template #1**, with ***Type* interpreted as *blot \**.** The recycle(&ink) function call **also matches Template #2**, this time with ***Type* being *blot***.

This combination sends two **implicit *instantiations***, *recycle<****blot \*****>(blot \*)* and *recycle<****blot****>(blot \*),* to the viable function pool.

Of these two template functions, recycle<blot \*>(blot \*) is considered the more specialized because it underwent fewer conversions in being generated. That is, Template #2 already explicitly said that the function argument was **pointer-to-Type**, so Type could be directly identified with blot. However, Template #1 had Type as the function argument, so **Type had to be interpreted as pointer-to-blot**. **That is, in Template #2, Type was already specialized as a pointer; hence it is “more specialized”**.

The rules for finding the most specialized template are called the *partial ordering rules* for function templates.

**A Partial Ordering Rules Example**

The first definition (**Template A**) assumes that the **array that is passed as an argument contains the data to be displayed**. The second definition (**Template B**) assumes that the **array elements are pointers to the data to be displayed**.

#include <iostream>

template <typename T> // template A

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

template <typename T> // template 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); **// uses template A, T taken to be type int**

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

**// pd is an array of pointers to double**

ShowArray(pd, 3); **// uses template B (more specialized), T is type double,**

return 0; **// T would be taken to be type double \* using template A**

}

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;

}

**If you remove Template B from the program**, the compiler then uses Template A for listing the contents of pd, so it lists the addresses instead of the values.

In short, the overload resolution process looks for a function that’s the best match. If there’s just one, that function is chosen. If more than one are otherwise tied, but only one is a non template function, that non template function is chosen. If more than one candidate is otherwise tied and all are template functions, but one template is more specialized than the rest, that one is chosen. If there are two or more equally good non template functions, or if there are two or more equally good template functions, none of which is more specialized than the rest, the function call is ambiguous and an error. If there are no matching calls, of course, that is also an error.

**Making Your Own Choices**

In some circumstances, you can lead the compiler to make the choice you want by suitably writing the function call.

#include <iostream>

template<class T> // or template <typename T>

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

{

return a < b ? a : b;

}

**int lesser (int a, int b) // #2, normal function**

{

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

return 0;

}

Here is the program output:

20

15.5

-30

15

These 2 functions don’t have prototype, since if a function definition appears before its first use, the definition acts as a prototype, so this example omits the prototypes.

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

**The function call arguments match both** the template function and the non template function, so the non template function is chosen, and it returns the value 20.

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

**The presence of the angle brackets in lesser<>(m, n) indicates that the compiler should choose a template function rather than a non template function**, and the compiler, noting that the actual arguments are type int, **instantiate**s the template using **int for T**.

Finally, consider this statement:

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

Here we have a request for an **explicit instantiation** using **int for T**, and that’s the function that gets used. The values of x and y are type cast to type int, and the function returns an int value, **which is why the program displays 15 instead of 15.5**.

**What’s That Type?**

One problem is that when you write a template function, it’s not always possible in C++98 to know what type to use in a declaration.

template<class T1, class T2>

void ft(T1 x, T2 y)

{

...

?type? xpy = x + y;

...

}

What should the type for *xpy* be? We don’t know in advance how ft() might be used. The proper type might be T1 or T2 or some other type altogether.

* **The decltype Keyword (C++11)**

The C++11 solution is a new keyword: ***decltype***. It can be used in this way:

int x;

decltype(x) y; // make y the same type as x

The argument to *decltype* can be an expression, so in the ft() example, we could use this code:

decltype(x + y) xpy; // make xpy the same type as x + y

xpy = x + y;

Alternatively, we could combine these two statements into an initialization:

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

So we can fix the ft() template this way:

template<class T1, class T2>

void ft(T1 x, T2 y)

{

...

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

...

}

Suppose we have the following:

decltype(expression) var;

Here’s a somewhat simplified version of the list.

***Stage 1***: If expression is an unparenthesized identifier (that is, no additional parentheses), then var is of the same type as the identifier, including qualifiers such as const:

double x = 5.5;

double y = 7.9;

double &rx = x;

const double \* pd;

decltype(x) w; // w is type double

decltype(rx) u = y; // u is type double &

decltype(pd) v; // v is type const double \*

***Stage 2***: If expression is a function call, then *var* has the type of the **function return type**:

long indeed(int);

decltype (indeed(3)) m; // m is type long

***Stage 3***: If expression is an **lvalue**, then *var* is a **reference** to the expression type.

double xx = 4.4;

decltype (**(xx)**)r2 = xx; **// r2 is double &**

decltype(xx) w = xx; // w is double (Stage 1 match)

For this stage to apply, **expression *can’t* be an unparenthesized** identifier. So what can it be? One obvious possibility is a **parenthesized identifier**.

Incidentally, parentheses don’t change the value or lvaluedness of an expression. For example, the following two statements have the same effect:

xx = 98.6;

(xx) = 98.6; // () don't affect use of xx

***Stage 4***: If none of the preceding special cases apply, *var* is of the same type as expression:

int j = 3;

int &k = j

int &n = j;

decltype(j+6) i1; // i1 type int

decltype(100L) i2; // i2 type long

decltype(k+n) i3; // **i3 type int;**

Note that **although k and n are references**, the expression **k+n is not a reference**; **it’s just the sum of two ints, hence an int**.

The difference between stage 3 and 4 is that stage 3 has (), while stage 4 doesn’t.

If you need more than one declaration, you can use **typedef** with **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]; // rxy a reference

...

}

* **Alternative Function Syntax (C++11 Trailing Return Type)**

The *decltype* mechanism by itself leaves another related problem unsolved. Consider this incomplete template function:

template<class T1, class T2>

?type? gt(T1 x, T2 y)

{

...

return x + y;

}

we don’t know in advance what type results from adding x and y. **It might seem that we could use decltype(x + y) for the return type**. ***Unfortunately, at that point in the code, the parameters x and y have not yet been declared, so they are not in scope (visible and usable to the compiler).***The decltype specifier has to come after the parameters are declared. To make this possible, C++11 allows a new syntax for declaring and defining functions. Here’s how it works using built-in types. The prototype:

double h(int x, float y);

can be written with this alternative syntax:

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

**This moves the return type to after the parameter declarations.** The combination ***-> double*** is called a **trailing return type**. Here, *auto*, in another new C++11 role, is a placeholder for the type provided by the trailing return type. The same form would be used with the function definition:

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

{/\* function body \*/};

Combining this syntax with decltype leads to the following solution for specifying the return type for gt():

template<class T1, class T2>

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

{

...

return x + y;

}

Now decltype comes after the parameter declarations, so x and y are in scope and can be used.

**[Summary]**

这一章讲了function, function overloading, template, template overloading, explicit specification, 和explicit instantiation.

1. **Function比较简单, 不用再说;**
2. **Function Overloading**

Function overloading是说一个函数名对应多个具有不同参数类型的参数列表 (signature).

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

用function overloading是一定要注意实参的类型是否和overloading里的某一个能对应的上. 比如下面这个:

unsigned int year = 3210;

print(year, 6); **// ambiguous call**

如果上面的定义里只有#2, 那还好说, 尽管类型不match, C++ Compiler还是会把unsigned int转换成double的 (这个转换叫Standard Type Conversion); 但上面#2, #3, #4通过Standard Type Conversion都能满足参数要求, Compiler就不知道该选谁了, 就报错.

还有下面这个例子:

Some signatures that appear to be different from each other nonetheless can’t coexist.

double cube(double **x**);

double cube(double **& x**); // These two overloadings can’t coexist

You might think this is a place you could use function overloading because the function signatures appear to be different. But consider things from the compiler’s standpoint. Suppose you have code like this:

cout << cube(x);

**The *x* argument matches both the *double x* prototype and the *double &x* prototype**. The compiler has no way of knowing which function to use.

还有, 参数里有const和没有const的兼容关系. 具体的见前面的例子.

char arrayname[20] = “……”, 这里arrayname的类型是char \*.

第14页还给了一个在子函数里new一块内存, 然后传出地址, 最后在main()中delete这个地址的例子.

1. **Template, template overloading, and explicit specialization**

**Template**定义了对一个generic type的函数操作, 是generic programming的代表.

**template <typename T>** // or **class T**

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

int main()

{

……

Swap(i, j); // int

Swap(m, n); // double

}

// function template definition

**template <typename T>**

void Swap(T &a, T &b)

{

T temp;

temp = a;

a = b;

b = temp;

}

注意template函数的函数名后面没有<>.

In Listing above, **you still wind up with two separate function definitions**, just as you would if you defined each function manually. **And the final code doesn’t contain any templates**; it just contains the actual functions generated for the program.

**有时候, 函数里的操作不能cover所有类型**, 这时需要**template overload**. 比如上面的Swap()无法适用于array：

#include <iostream>

**template <typename T>** // original template

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

**template <typename T>** // new template, **因为array做参数需要指定长度**

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

const int Lim = 8;

int main()

{

using namespace std;

int i = 10, j = 20;

Swap(i,j); // matches original template

……

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

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

Swap(d1,d2,Lim); // matches new template

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;

}

}

**有时候, 即使函数里的操作能cover所有类型, 但有些操作对不同的数据类型意义不一样, 比如下面的:**

**a = b**; // not be true if type T is a built-in array type

**if (a > b)** // not true if T is an ordinary structure

// Also the > operator is defined for array names, but because array names are

// addresses, it compares the addresses of the arrays, which may not be what you have // in mind.

**T c = a\*b**; // not the case if T is an array, a pointer, or a structure

这就引出了**explicit specialization**.

比如前面的Swap(T &a, T &b)模板, 它通过’=’实现数据的交换, 这个函数对于同一个class的不同object也是适用的, 能交换它们的数据成员. 但如果我只想交换部分成员呢? 这时函数的参数不变, 但函数里面的算法变了, template overloading无法满足这一点. 这也是explicit specialization的原因.

// **non template function** prototype

void Swap(job &, job &);

// **template** prototype

template <typename T>

void Swap(T &, T &);

// **explicit specialization** for the job type

**template <> void Swap<job>(job &, job &); // Swap()后的<job>是optional的;**

As mentioned previously, if more than one of these prototypes is present, the compiler chooses the **non template version** ***over*** **explicit specializations** and **template versions**, and it chooses an **explicit specialization** ***over*** a version generated from a **template**.

1. **Instantiation**

定义模板本身并不是定义函数, 只有当你在程序中具体指定某个类型并且用到模板时, 这个函数调用才会产生函数定义, 这个过程叫implicit instantiation.

但C++也允许explicit instantiation. 这是说在调用模板时对模板指定一个具体的类型. **The syntax is to declare the particular variety you want, using the <> notation to indicate the type and prefixing the declaration with the keyword template**:

template void Swap**<int>**(int, int); // explicit ***instantiation***

A compiler that implements this feature will, upon seeing this declaration, use the Swap() template to generate an instantiation, using the int type. That is, this declaration means “**Use the Swap() template to generate a function definition for the int type**.”

template **<>** void Swap<int>(int &, int &); // explicit ***specialization***

template **<>** void Swap(int &, int &); // explicit ***specialization***

**It is an error to try to use both an explicit instantiation and an explicit specialization for the same type(s) in the same file, or, more generally, the same translation unit.**

**注意: explicit specialization是需要有独立的函数定义的, 而explicit instantiation没有独立的函数定义, 它只是在调用模板时指定一下类型**.

比如:

cout << **Add<double>(x, m)** << endl; // explicit **instantiation, x is int, m is double**

如果直接用Add(x, m)会失败, 因为模板要求x和m是同一类型的; 但用了Add<double>(x, m)会强迫进行double类型的explicit instantiation, 这会使得x被cast到double类型上.

template <class T>

void Swap (T &, T &); **// template prototype, need template definition**.

template <> void Swap<**job**>(job &, job &); // explicit **specialization** for job.

// **Need function definition**

int main(void)

{

**template void Swap<char>(char &, char &);** // explicit **instantiation** for char.

// **Doesn’t need function definition**.

short a, b;

Swap(a,b); // **implicit** template instantiation for short

job n, m;

Swap(n, m); // use **explicit specialization** for job

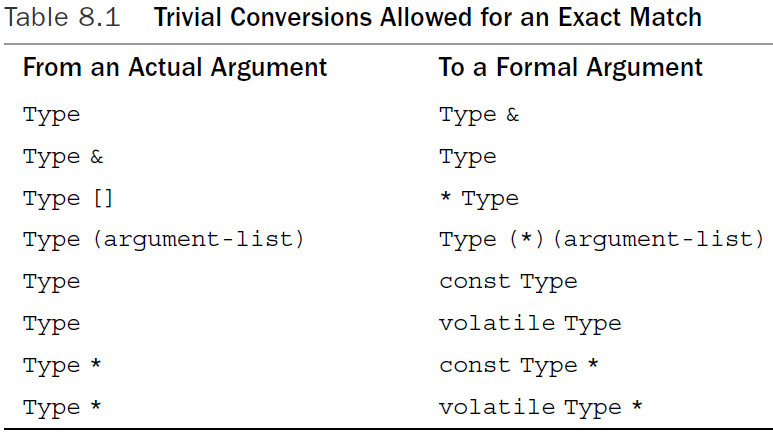
char g, h;

Swap(g, h); // use explicit template **instantiation** for char defined in the 1st line

}

**当这些都存在时, 到底哪个会被调用呢? 最重要的原则是Exact Match.**

Exact Match并不仅仅是说函数的实参和形参完全一致, 当然完全一致是exact match; 它还允许一些简单的conversion. 前面有个表里有exact match的一些标准.



In general, the ranking from best to worst is this:

1. **Exact match**, with **regular functions outranking templates**;
2. **Conversion by promotion** (for example, the automatic conversions of char and short to int and of float to double);
3. **Conversion by standard conversion** (for example, converting int to char or long to double);
4. User-defined conversions, such as those defined in class declarations;

所以, exact match优于promotion优于standard conversion. **char-to-int is a promotion** (refer to Chapter 3,“Dealing with Data”), whereas **char-to-float is a standard conversion** (refer to Chapter 3).

**当最后可选的具有相同优先级的函数既有普通函数, 也有template, 也有explicit specialization时**, As mentioned previously, if more than one of these prototypes is present, the compiler chooses the **non template version** ***over*** **explicit specializations** and **template versions**, and it chooses an **explicit specialization** ***over*** a version generated from a **template**.

**当最后具有相同优先级的可选的函数都是同一类型的函数时**, 判断原则是**more specialize**. 比如当最后的函数都是template时(普通template, explicit specialization, explicit instantiation都是template的函数), more specialize意味着选需要做的转换(针对函数参数而言)最少的那个. 比如第23页最下面那个例子, 两个可选的都是exact match, 选了需要转换最少的那个.

当参数是指针和reference时, const和非const是有优先级差别的, 最贴近实参原型的具有高优先级, const本身针对这两个类型没有优先.

当普通函数和template同等优先级, 也可以强行指定使用template的.