Overloading and Templates

函数模板一样可以被重载。既可以被另一个模板重载也可以被一个普通函数重载。当然，它们必须具有不同数量或类型的参数（一般只要能区别开来即可）。

函数匹配规则，在函数模板重载这里有些不同。首先，函数匹配会产生若干候选函数。这些候选函数会按照类型转换来排序。在函数模板重载中，

- 对于一个调用，其候选函数包括所有模板实参推断成功的函数模板实例。
- 候选的函数模板总是可行的，因为模板实参推断会排序任何不可行的模板。
- 可行函数（模板与非模板）按照类型转换排序。这里应当注意，函数模板调用的类型转换是非常有限的。
- 最佳匹配原则。若同样好，则
    - 优先选择非模板函数
    - 优先选择更**特例化** (more specialized) 的函数模板
    - 否则，调用有歧义

Writing Overloaded Templates

例子：

```
// print any type we don't otherwise handle
template<typename T> string debug_rep(const T &t){
    ostringstream ret;
    ret<<t; // uses T's output operator to print a representation of t
    return ret.str();// returns a copy of string to which ret is bound
}
```

此函数用来生成一个对象对应的`string`表示，当然该类型对象应该定义`<<`输出运算符。因此，该对象可以是任意具备输出运算符的类型。

接下来，定义打印指针的版本：

```
// print the value of pointers,
template <typename T> string debug_rep(T *p){
    ostringstream ret;
    ret<<"pointer:"<<p; // print the value of the pointer p itself
    if(p)
        ret<<" "<<debug_rep(*p);
    else
        ret<<"null pointer";
    return ret.str();// returns a copy of string to which is ret bound
}
```

对于`string s("hi"); cout<< debug_rep(s)<<endl`调用来说，只有一个版本的`debug_rep`是可行的。第二个`debug_rep`版本要求一个指针参数，但在此调用中我们传递的是一个非指针对象，因此编译器无法从一个非

指针实参实例化一个期望指针类型参数的函数模板，因为实参推断失败。

我们可以使用一个指针调用：
`cout<<debug_rep(&s)<<endl;`
此时，两个函数都生成了可行的实例：

- `debug_rep(const string * &)`，由第一个版本实例化而来，T被绑定到`string*`
- `debug_rep(string *)`，由第二个版本的`debug_rep`实例化而来，T被绑定到`string`
  然后第二个版本是最佳匹配。第一个版本需要进行普通指针到`const`指针的转换。

Multiple Viable Templates

我们看一个应选择更specialized版本的函数的例子，

```
const string *sp=&s;
cout<<debug_rep(sp)<<endl;
```

此时，以上两个模板都是精确匹配，

- `debug_rep(const string *&)`，由第一个版本实例化，T被绑定到`string *`
- `debug_rep(const string *)`，由第二个版本实例化，T被绑定到`const string`
  这样的情况，该调用就是有歧义的？但是，我们注意到第一个版本`debug_rep(const T &)`本质上可以适用于任何类型，包括指针类型。这个模板比第二个版本更加通用。因此，第二个版本更加特列化，因此编译器将会调用第二个。

Nontemplates and Templates Overloads

接下来是一个候选函数包含模板函数和非模板函数的例子。

```
string debug_rep(const string &s){
    return '"'+s+'"';
}
```

现在，我们对`string`调用`debug_rep`时：

```
string s("hi");
cout<<debug_rep(s)<<endl;
```

有两个同样好的可行函数：

- `debug_rep<string> (const string &)`，第一个模板T被绑定到`string *`
- `debug_rep<const string &>`，普通非模板函数
  此时，两者都是最佳匹配，但是编译器按照，优先非模板函数原则，选择非模板函数。

当存在多个同样好的函数模板时，编译器选择最特例化的版本以及遵循非模板函数优先原则。

Overloading Templates and Type Conversion

当重载模板遇到类型转换：我们考虑一个特殊的情况，即`char *`和`string`。考虑这个调用
`cout<<debug_rep("hi world")<<endl;//调用debug_rep`
其中，`'hi world'`是一个`char [10]`类型的常量。

此时，有 三个`debug_rep`可行版本：

- `debug_rep(const T &)`,`T`被绑定到`char [10]`。
- `debug_rep(T *)`，`T`被绑定到`const char`，该版本进行了一次数组到指针的转换。
- `debug_rep(const string &)`，该版本要求`const char *`到`string`的类型转换。
  这种情况下，前两个版本都提供了精确匹配-第二个需要一次转换，**而对于函数匹配来说，这种转换被认为是精确匹配**。同时，非模板版本需要一次用户定义的类戏转换，因此，它没有精确匹配那么好，所以前两个版本称为了可能调用的函数。而`T*`版本更加specialized，编译器因此选择它。

我们可以提供两外两个非模板重载版本，以将字符指针按`string`处理

```
string debug_rep(char *p){
    return debug_rep(string(p));
}
string debug_rep(const char *p){
    return debug_rep(string(p));
}
```

Declaration Miss Can Casuse the Program to Misbehave

值得注意的是，为了使`char *`版本的`debug_rep`正确工作，在定义此版本时，`debug_rep(const string &)`的声明必须在作用域内。否则，可能调用错误的版本（这不是很显然麽？）：

```
template <typename T> string debug_rep(const T &t);
template <typename T> string debug_rep(T *p);
//
string debug_rep(const string &);
string debug_rep(char *p);{
    // 如果接受一个const string &的版本的声明不在作用域中，
    // 返回语句将会调用debug_rep(const T &)的T实例化为string的版本
    return debug_rep(string(p));
}
```

通常，如果使用了一个忘记声明的函数，代码将编译失败。但对于重载函数模板的函数而言，则不是这样。如果编译器可以从模板实例化出与调用匹配的版本，则缺少的声明就不重要了。在本例中，如果忘记了声明接受了`string`参数的`debug_rep`版本，编译器会默默地实例化接受`const T &`的模板版本。