1，定义模版

1.1函数模版

template <typename T>

int compare(const T &v1, const T &v2){}

在模版定义中，模版参数列表不能为空。

当我们调用一个函数模版时，编译器通常用函数实参来为我们推断模版实参。

类型参数前必须使用关键字class或typename

一个模版参数列表中可以同时使用两个关键字

template <template T,class U> calc(const T&, const U&)//typename和class没什么不同

除了定义类型参数，还可以在模版中定义非类型参数，一个非类型参数表示一个值而非一个类型。

当一个模版被实例化时，非类型参数被一个用户提供的或编译器推断出的值代替。这些值必须是常量表达式。

第一个模版参数表示第一个数组的长度，第二个参数表示第二个数组的长度：

template<unsigned N, unsigned M>

int compare(const char (&p1)[N], const char (&p2)[M]){}

compare("hi","mom")

一个非类型参数可以是一个整型，或者是一个指向对象或函数类型的指针或引用。

绑定到非类型整型参数的实参必须是一个常量表达式。

绑定到指针或引用非类型参数的实参必须具有静态的生存期。

非类型模版参数的模版实参必须是常量表达式。

函数模版可以声明为inline或constexpr的，说明符放在模版参数列表之后，返回类型之前。

编写泛型代码的两个重要原则：1，模版中的函数参数是const的引用；2，函数体中的条件判断仅使用<比较运算

函数模版和类模版成员函数的定义通常放在头文件中。

1.2 类模版

与函数模版的不同之处是，编译器不能为类模版推断模版参数类型。

template <typename T> class Blob{}

当编译器从我们的Blob模版实例化出一个类时，它会重写Blob模版，将模版参数T的每个实例替换为给定的模版实参。

定义在类模版之外的成员函数就必须以关键字tempalte开始，后接类模版参数列表。

当我们处于一个类模版的作用域中时，编译器处理模版自身引用时就好像我们已经提供了与模版参数匹配的实参一样。

在一个类模版的作用域内，我们可以直接使用模版名而不必指定模版实参。

如果一个类模版包含了一个非模版友元，则友元被授权可以访问所有模版实例。

如果友元自身是模版，类可以授权给所有友元模版实例，也可以只授权给特定实例。

为了引用（类或函数）模版的一个特定实例，我们必须首先声明模版自身。

编程：一对一友好关系

编程：通用和特定的模版友好关系

1.3 模版参数

我们通常将类型参数命名为T，但实际上我们可以使用任何名字。

就像我们能为函数参数提供默认实参一样，我们也可以提供默认模版实参。

template <typename T,typename F = less<T>>

int compare(const T &v1, const T &v1, F f = F())

与默认实参一样，对于一个模版参数，只有当它右侧的所有参数都有默认实参时，它才可以有默认实参。

模版默认实参与类模版

template <class T = int> class Numbers{};

1.4 成员模版

一个类可以包含本身是模版的成员函数。这种成员被称为成员模版。成员模版不能是虚函数。

1.5 控制实例化

在大型系统中，在多个文件中实例化相同模板的额外开销可能非常严重。

在新的标准中，可以通过显示实例化来避免这种开销。

extern template class Blob<string>;

template int compare(const int&, const int&);//

对每个实例化声明，在程序中每个位置必须有其显式的实例化定义。

1.6 效率与灵活性

对模版设计者所面对的设计选择，标准智能指针类型给出了一个很好的展示。

2 模版实参推断

2.1 类型转换与模版类型参数

顶层const无论是在形参中还是实参中，都会被忽略。

将实参传递给带模版类型的函数形参时，能够自动应用的转换类型只有const转换以及数组或函数到指针的转换。

2.2 函数模版显式实参

long lng;

compare(lng,1024)//错误

compare<long>(lng,1024)//正确

compare<int>(lng,1024)//正确

2.3 尾置返回类型与类型转换

template <typename It>

auto fcn(It beg, It end) -> decltype(\*beg){}

remove\_reference<decltype(\*beg)>::type

将获得beg引用的元素的类型：decltype(\*beg)返回元素类型的引用类型。remove\_reference<decltype(\*beg)>::type脱去引用，剩下元素类型本身。

在函数中返回元素值的拷贝：

template<typename It>

auto fcn2(It beg,It end)->typename remove\_reference<decltype(\*beg)>::type

参考文档：标准类型转换模版

2.4 函数指针和实参推断

当我们用一个函数模版初始化一个函数指针或为一个函数指针赋值时，编译器使用指针的类型来推断模版实参。

template<typename T> int compare(const T&,const T&)

int (\*pf1)(const int&, cosnt int&) = compare;

当参数是一个函数模版实例的地址时，程序上下文必须满足：对每个模版参数，能唯一确定其类型或值。

2.5 模版实参推断和引用

（待续 2.5 2.6 2.7）

2.6 理解std::move

2.7 转发

3 重载与模板

正确定义一组重载的函数模板需要对类型间的关系及模板函数允许的有限的实参类型转换有深刻的理解。

当有多个重载模板对一个调用提供同样好的匹配时，应选择最特例化的版本。

对于一个调用，如果一个非函数模板与一个函数模板提供同样好的匹配，则选择非模板版本。

在定义任何函数之前，记得声明所有重载的函数版本。这样就不必担心编译器由于未遇到你希望调用的函数而实例化一个并非你需要的版本。

4，可变参数模板

一个可变参数模板就是一个接受可变数目参数的模板函数或模板类。

可变数目的参数被称为参数包，存在两种参数包：模板参数包，表示零个或多个模板参数；函数参数包，表示零个或多个函数参数。

template<typename T,typename... Args>

void foo(const T &t, const Args& ... rest)

Args是一个模板参数包，rest是一个函数参数包

例子：见书本

当我们需要知道包中有多少元素时，可以使用sizeof...运算符。sizeof...(Args)

4.1 编写可变参数函数模板

可变参数函数通常是递归的，第一步调用处理包中的第一个实参，然后用剩余实参调用自身。

编程：可变参数模板进行打印

当定义可变参数版本的print时，非可变版本的声明必须在作用域中。否则，可变参数版本会无限递归。

4.2 包扩展

对于一个参数包，除了获取其大小外，我们能对它做的唯一的事情就是扩展它。

当扩展一个包时，我们还要提供用于每个扩展元素的模式。

扩展一个包就是将它分解为构成的元素，对每个元素引用模式，获得扩展后的列表。

我们通过在模式右边放一个省略号(...)来触发扩展操作。

例子：见书本

扩展的模式会独立地应用于包中的每个元素。

4.3 转发参数包

（待续）

5 模板特例化

一个特例化版本就是模板的一个独立的定义，在其中一个或多个模板参数被指定为特定的类型。

定义函数模板特例化

template <>

int compare(const char\* const &p1, const char\* const &p2){}

一个特例化版本本质上是一个实例，而非函数名的一个重载版本。因此，特例化不影响函数匹配。

除了特例化函数模板，我们还可以特例化类模板。

编程：为标准库hash模板定义一个个特例化版本，保存Sales\_data对象。

待续