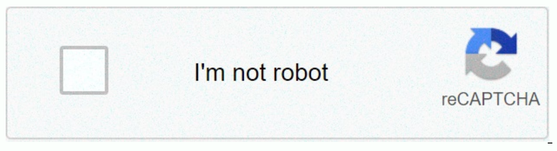

Continue

C template functor example

April 18, 2008 by Alex | Finally, in the previous chapter, revised by Alex on January 23, 2020, i learned how to write functions and classes to make it easier to write, secure, and maintain programs if you need a function template. Functions and classes are powerful and flexible tools for effective programming, but in some cases they can be somewhat limited by the requirement of C++ to specify all parameter types. For example, let's say you wrote a function that calculates up to two numbers. Int max (int x, int y) {Return (x > y) ? x: y;} This feature will have a great effect on integers. What happens later when I realize that the maximum () function must be East Sea in doubles? Traditionally, the answer is to create a new version that works with maximum (overloaded) functionality and double: the code for implementing a double version of Double Max (double x, double y) {return (x > y) is exactly the same as the int version of the maximum ()! In fact, this implementation will work for all sorts of different types: chars, ints, doubles, if you have overloaded > operators, and even classes! However, Because C++ requires variables to create a specific type, it is stuck in writing one function for each type you want to use. The type of parameters can be maintenance headaches and wasteful time, and redundant code should be specified in a different taste of the same function that violates the general programming guidelines that should be minimized as much as possible. Wouldn't it be nice if you could write one version of the maximum () version that you can work with with all types of parameters? Welcome to the world of templates. What is a function template? If you look up the word template in a dictionary, you can find definitions similar to the following: One type of template that is easy to understand is the template for the stencil. A stencil is a shaped object(for example, a piece of cardboard), such as a character J. Place the stencil on top of another object and then spray the paint through the hole to create a stencil pattern in a variety of colors very quickly! You only need to create a specified stencil once - you can use it as many times as you want to create a stencil pattern for any color. Better, you don't have to decide the color of the stencil pattern you want to create until you decide to actually use the stencil. In C++, a function template is a function that acts as a pattern for creating other similar functions. The basic idea of a function template is to create a function without specifying the exact type (s) of some or all of the variables. Instead, it defines a function using a placeholder type called a template type parameter. Once we have You created functions using these placeholder types, and you effectively created function stencils. When calling a template function, the compiler stenciates a copy of the template and replaces the placeholder type with the actual variable type in the parameters of the function call. Using this methodology, the compiler can create multiple flavors of the function in one template! We'll take a closer look at this in the next lesson. At this point, if you create a function template in C++, you may actually be wondering how to create a function template with C++. It turns out it's not all that difficult. Let's look at the int version of the back up () of: int max (int x, int y) {return (x > y) ? x: y;} There are three locations in which a particular type is used: parameters x, y, and return values all specify that they must be integer. To create a function template, replace this particular type with placeholder types. In this case, only one template type parameter is required because there is only a type (int) that needs to be replaced. You can name the placeholder type of almost anything you want, unless it's a reserved word. However, in C++, it is customary to name the template type of character T (short in the case of type). Here's a new feature with placeholder type: T max (T x, T y) {return (x > y) ?x: y;} This is a good start - but it's not compiled because the compiler doesn't know what T is! To do this, you need to tell the compiler two things: first, template definition, and t, placeholder type. You can do both in one line using what's called template parameter declaration: template & typename t=>/// This is a template parameter declaration (T x, T y) return (x> y) ? x: y;} Believe it or not, that's all we need. Compiled! Now let's take a closer look at the template parameter declaration. Start with a keyword template - the following tells the compiler that it will be a list of template parameters. We place all of our parameters (in angled brackets).lt> To create template type parameters, use a keyword type name or class. In this context, there is no difference between the two keywords, so the keyword you use is up to you. If you use class keywords, the type passed doesn't really need to be a class (it can be a base variable, pointer, or any other matching item). Then name the type (typically T). If the template function uses multiple template type parameters, they can be separated into commas: template & typename t1,= typename= t2=>//// It is common to see that the template function is another single capital character name, such as T1 and T2, or S, for classes that use more than one type here. Last note: The function arguments passed to type T can be class types, and generally it is not recommended to pass classes by value, so it's not a good idea to pass classes by value.The parameters and return type of template function configuration reference: template <typename t=&configurator T&& Max (const T&x, const T&y) {Return (x> y) ? x: y;} It is very simple to use a function template using a function template. Here is the full program using our template feature: 123456789111211313141516171819202 1 #include <iostream>template configurator T<typename t=>&&& Max (const T&x, const T&y) return (x> y) ? x: y;} int main () {int i = up (3, 7); // 7 std::: cout), & i=> > 'lt; double= d=max (6.34, 18.523;= returns= 18.523= std:::cout=> > 'lt; '> ';=char= ch=max ='a', '6'); = returns= 'a'= std::cout=> > ch=gt;gt; "return='&gt='>0 = this= will= print:= 7= 18.523= a= note= that= all= three= of these= calls= to= max=have=parameters= of= different= types!= because we've= called= the= function= with= 3= different= types,=the= the compiler= will= the use= the = the use= Template= definition= to= create= 3= different= versions= of this= function:= one= with= int= parameters= (named=>lt;int>double>double;name) and char parameter (maximum name) >char>returns. As long as the compiler can infer the template from the parameter type, it is not necessary to explicitly specify the template type in the function name (for example, the maximum part). As you can see, template functions can save a lot of time and work in different types because you only need to write one function. Once you get used to writing function templates, you don't write any more than functions with real types. Template functions can be more secure because you don't have to copy them and change the format yourself whenever you need a function to work with a new type! There are some drawbacks to template functions, and we will be dismissed without mentioning them. First, some older compilers don't have much template support. However, this disadvantage is no longer as problematic as it used to be. Second, template functions often generate insanely-looking error messages that are much more difficult to decipher than regular functions (you can see an example of this message in the next lesson). Third, template functions can increase compilation time and code size because a single template can be realized and recompiled from multiple files (there is a way to fix this file). However, these drawbacks are quite trivial compared to the power and flexibility templates you bring into your programming toolkit! Note: The standard library already comes with a template maximum (algorithm header), so you don't have to write it yourself unless you want it. If you write your own, you will not know whether the compiler wants the maximum version of max () or last name:max(), so if you use the statement namespace std; you record the possibility of naming a conflict. For the rest of this chapter, we'll continue to look at the theme of the template. Template. > int>/int>/char&/char&/double>/int>lt;typename;/iostream>lt;typename;/typename