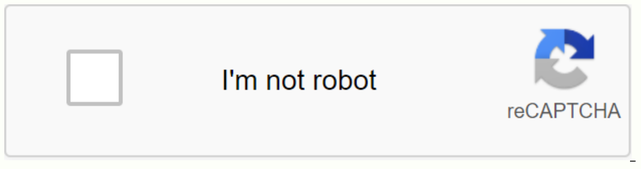

Continue

Generics in c

Captions: PamperedChefGenerics are syntax components of the programming language that can be used for different types of objects. Typically, generics take form classes or functions that take type(s) as a parameter. Generics are also commonly referred to as templates, and C++ is officially called templates. Typically, programming languages that don't have generics or templates have received criticism. Having the opportunity to share the procedural code between types greatly improves quality and development. This article describes several methods and methods to achieve a template as functionality in a pure C.What is the type? Generics and templates allow types to use the same code. But, what really represents a type in a language like C? Unlike other object-oriented programming languages, C has no native concept of building and destroying objects. There is also no native inheritance concept embedded in the language. C examines the types of memory partitions with a specific known format of their data. Officially, the types are divided into two categories, scalar and aggregate types. The scalar type contains only one data item, such as int or char*. Most often it is an integer or types of pointers. Aggregate types have one or more data values, such as an array or pod. Void type* is a special type of pointer that can indicate any type, scalar or aggregate. In fact, emptiness * can sometimes be used as a form of general code. Quite a few features in string.h use emptiness* to read and write pieces of memory of different sizes. Therefore, the type in C should have: known sizeA known formatTwa types can have the same size, but not the same format. Such as, on most platforms, float and int. Usually they are the same size, and, as such, sizeof(int) == sizeof(float) is stored correctly. However, they have a completely different format, and thus will behave differently when causing certain functions. Static templatesSoon type of templates that can be implemented in C are static templates. Static templates are created during compilation and do not perform size execution checks because they translate this responsibility to the compiler. The static templates used in C are similar to C++ templates because they depend on actual type participants, such as struct. In C, the only native tool for creating static templates is to use macros. Note: Using a static word means only during compilation. This is not related to the static keyword C, which means that the function has an internal connectionFor the beginning let's observe the simplest form of a static template, the macro definition: the above def macro can work with any type that can be initialized with operator = . It can also work by doing: This template allows the code needed to identify variables with the primary type to be generalized and abstract. The goal is to make layerable and similar between types. However, this example is trivial in the sense that it simplifying simply writing out a statement. The real power comes from code that performs more complex tasks. ForeachIn C, for loops can cause a lot of code to be very type specific, resulting in redundant code and huge source files. With static templates in the form of macros, we can make for loops much smoother in C. First, consider the standard for the cycle:There are four different syntax elements. Initialize the variable, the termination condition, the incremental step, and finally the actual block of code that will be executed every step of the cycle. Given these four elements, we can build a macro that acts as a thought loop. This template works for primitive type ranges where there is a known type, starting value, target value, and required function to be applied to each member of the range. Name_foreach_var chosen to purposefully reduce the possibility of colliding with one of the macro arguments. Note that fn should not be a function. You can, in this case, pass in a macro for fn, also called a higher order macro: For cycle templates, you can also use loops that iterate the contents of a cumulative type, such as an array. The use of aggregate types can sometimes be even more direct forward with macros because we can get more information from the aggregate type than the scalar type. Using the operator size, we can determine the number of elements in the array. Operator size always accepts the size of the value in bytes. This means that the transmission of the array will estimate the total size of this array. This does not work with pointer types because the size of any pointer will always be the same as sizeof(void*). This information, which can be obtained from an array, can be used to create a powerful foreach template: In the FOREACH macro above, the only two arguments required are the array itself, arr, and some can be applied to each element in arr. Type size_t can be used to access any array element. We don't need to also go through in type arr because we don't need it, we can let the FN handle it. FunctionsStatic templates can also be implemented in C to implement entire functions. This approach to generics and templating is necessary when the desired behavior is more complex than what can be captured in a cycle-oriented template. Typically, in C, parameters taking the form of several different types will be given a type of void*, which is another size_t parameter that represents the memory size of the void * indicates. This function definition style is used in a standard library, for example with memcpy and memset functions. The problem with using invalid * is the lack of type validation. Any address of any type C can be stored in the void *, for example,These are all valid definitions. The compiler cannot interpret an incorrect type that is passed invalid*, intended for another type. Thus, functional templates provide limit the amount of types taken in general To begin with, let's look at an example of comparing two aggregate types by value of a specific field for equality: The above template is a macro that creates a function definition every time it is called. This function occupies two constances of the specified type and returns the result of the field comparison. The name of the function produced is called the type added to the end. We need every definition of this equality template function to have its own name because C does not allow functions to have the same name but conflicting types. This means that, with this template, doingWould actually FIELD_EQ_point_t and FIELD_EQ_foo_t respectively. The advantage here is that we can create the same function for multiple types without writing out these functions multiple times. Here's an example of both using a functional template and calling a manufactured function:This function production template can be used to create the same C APIs between different types. Structuresin C structures, called structures short, are cumulative types that can contain heterogeneous, named data fields. Compared to other languages, struct is like an object with no methods, no constructor and no destructor. Pieces are pieces of formatted data consisting of several scalar types. Structures differ from objects in other languages, because there is no built way to achieve polymorphism. There is also no native way to achieve inheritance, for example, in the desired case of parental current and baby current. Interpreted programming languages, such as Python, use the object's head approach. This means that any structure that is considered a language object has a special set of fields at the beginning of its definition that have information related to the identity of structures. This allows all structures to be thyed to the general base type as long as they transfer these fields before any other specific fields are defined. The Python C API achieves this by using a macro called PyObject_HEAD. This macro contains fields that identify the type of shake, its size, and many others. A more basic example would look like this:In the above obj_t is the basic shaking of all obj_*_t related structures. It is defined purely by using a macro OBJECT_HEAD, holding down information about the type struct. This macro is a template, any other OBJECT_HEAD with its part, because its definition can be thyed obj_t. This can be seen in a obj_int_t type pod. Create types of struct that can be watered down to obj_t can be used using a higher-order macro template: these two macros, OBJECT_START, and OBJECT_END allow for more standard definitions of template structure types. They ensure that any defining statements between the original and final macros will form a obj_t compatible type. It also ensures that the type name will exist under struct <name>and <name>namespaces. The conclusion that implements the templates in C can&/name> Code C is more readable, less redundant and less error-prone. This allows efficient development without having to include or switch to C++ or languages with built-in system templates, if desired. Using generics and templates in C can also make applications more secure, and prevent incorrect access to memory. Memory.