How to Use Function Template Parameter Packs in C++

C++ template parameter packs (also known as variadic templates) were introduced in the C++11 standard and are powerful tools that let functions accept an arbitrary number of arguments. Without further ado, jump into your favourite IDE,...

Method 1 of 2:

Using Non-Type Function-Template Parameter Packs

1. **Set up the main function.** No header files are required to construct a variadic template. For demonstration purposes in this article, the **iostream** header file will be included.

   ```
   #include using namespace std; int main() { }
   ```

2. **Construct a non-type templated function.** The code below shows a function accepting an **int** template argument.

   ```
   #include using namespace std; templateint N> void addOne() { cout  N+1
    endl; } int main() { addOne5>(); // Expected output: 6 }
   ```

3. **Expand the template to a parameter pack.** Consider what happens if you want to add more arguments to the function, so that you can call the function with any number of arguments. You *could* provide default arguments, e.g.

   ```
   templateint N, int N2 = 0, int N3 = 0> void addOne() { cout  N+1  " "
   N2+1  " "  N3+1  endl; }
   ```

   1. But this is tedious. What's more, this will always output 3 numbers, regardless of the arguments passed. A parameter pack easily resolves this:

   ```
   templateint ...Ns> void addOne() { for (int i : {Ns...}) cout  i+1
   endl; }
   ```

   1. The ellipsis "packs" **int** values into **Ns**. It is then "unpacked" into an initialiser-list in a for-range loop. The loop will output each value plus one as it iterates.
      1. You can use **Ns** to denote the presence multiple **N** template parameters.
4. **Call the function.** Pass an arbitrary number of **int**s to the template and watch as you have successfully constructed and called your first variadic template function.

```
int main() { addOne5, 1, 4, -20>(); // Expected Output:// 6// 2// 5
// -19return 0; }
```

Method 2 of 2:

Using Type Function-Template Parameter Packs

1. **Set up up the main function and construct a simple templated function.** Include any necessary header files. The template function used in this example will be a print function, output a variable of any type.

   ```
   #include using namespace std; templateclass T> void print(T arg) { cout
     arg  endl; } int main() { print("Hello wikiHow!");
   // Expected Output:  // Hello wikiHow! }
   ```

   1. Due to type deduction, you don't need to pass any type-arguments to the template.
2. **Expand the template to a parameter pack.** Similar to packing non-type template parameters, you use an ellipsis to pack **types**. The sample code below shows how function arguments can be white-space-delimited along a single line.

   ```
   // parameter pack templateclass T, class ...Ts> void print(T arg, Ts...
    args) { cout  " "  arg; print(args...); }
   ```

   1. It's impossible to unpack **args** into an initialiser-list unless the values passed into **print** all have the same type. However, you can rely on recursion to handle the job. One argument is deducted on each recursive iteration and printed.
   2. You can use **Ts** to denote multiple **T**. Note that these **Ts** could be of any type.
   3. You may also choose to unpack the values into an **std::tuple**.
3. **Provide a base case.** Without the base case, compilers will throw errors about a missing **print()** function (with zero arguments). It is important to provide a base case to both terminate the recursion and handle any cleanup (in this case, printing a newline). When the number of arguments in **print** reaches 0, the base case will be called. A suitable base case that will terminate the recursion may be:

   ```
   // base case void print() { cout  endl; }
   ```

4. **Test the function.** Ensure that the function works correctly as intended.

   ```
   int main() { print(2019, ':', "Happy", 14, "th", "Birthday", "wikiHow!"
   ); // Expected Output:  // 2019 : Happy 14 th Birthday wikiHow!}
   ```

   1. To use **print** on STL containers or user-defined classes, you will need to overload **operator**.

You finished reading the article "**How to Use Function Template Parameter Packs in C++**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.