**Efficiency considerations and const references.**

Calling a function with parameters taken by value causes copies of the values to be made. This is a relatively inexpensive operation for fundamental types such as int, but if the parameter is of a large compound type, it may result on certain overhead.

For example, consider the following function:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | string concatenate (string a, string b)  {  return a+b;  } |  |

This function takes two strings as parameters (by value), and returns the result of concatenating them. By passing the arguments by value, the function forces *a* and *b* to be copies of the arguments passed to the function when it is called. And if these are long strings, it may mean copying large quantities of data just for the function call.

But this copy can be avoided altogether if both parameters are made *references*:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | string concatenate (string& a, string& b)  {  return a+b;  } |  |

Arguments by reference do not require a copy. The function operates directly on (aliases of) the strings passed as arguments, and, at most, it might mean the transfer of certain pointers to the function. In this regard, the version of concatenate taking references is more efficient than the version taking values, since it does not need to copy expensive-to-copy strings.

On the flip side, functions with reference parameters are generally perceived as functions that modify the arguments passed, because that is why reference parameters are actually for.

The solution is for the function to guarantee that its reference parameters are not going to be modified by this function. This can be done by qualifying the parameters as constant:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | string concatenate (const string& a, const string& b)  {  return a+b;  } |  |

By qualifying them as *const*, the function is forbidden to modify the values of neither *a* nor *b*, but can actually access their values as references (aliases of the arguments), without having to make actual copies of the strings.

Therefore, const references provide functionality similar to passing arguments by value, but with an increased efficiency for parameters of large types. That is why they are extremely popular in C++ for arguments of compound types. Note though, that for most fundamental types, there is no noticeable difference in efficiency, and in some cases, const references may even be less efficient!

Examples to understand:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | #include <iostream>  using namespace std;  void multiplication(double h, double k)  {  h = 2 \* h;  k = 2 \* k;  cout << "h = " << h << endl;  cout << "k = " << k << endl;  }  int main()  {  double h = 0.5, k = 4;  multiplication(h, k);  cout << "h = " << h << endl;  cout << "k = " << k << endl;  return 0;  } | h = 1  k = 8  h = 0.5  k = 4 | [Edit & Run](https://cplusplus.com/doc/tutorial/functions/) |

In above program the *multiplication* function takes variables *h* and *k* to its parameter by passing them by their value. Thus, before *multiplication* function begin it takes copies of *h* and *k* values.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17  18  19  20  21  22 | #include <iostream>  using namespace std;  void multiplication(double& h, double& k)  {  h = 2 \* h;  k = 2 \* k;  cout << "In the multiplication function:\n";  cout << "h = " << h << endl;  cout << "k = " << k << endl;  }  int main()  {  double h = 0.5, k = 4;  cout << "Before function call:\n";  cout << "h = " << h << endl;  cout << "k = " << k << endl;  multiplication(h, k);  cout << "After the function call in the main:\n";  cout << "h = " << h << endl;  cout << "k = " << k << endl;  return 0;  } | Before function call:  h = 0.5  k = 4  In the multiplication function:  h = 1  k = 8  After the function call in the main:  h = 1  k = 8 | [Edit & Run](https://cplusplus.com/doc/tutorial/functions/) |

However, in below program after the same function takes them as references. Thus, directly working on them or affecting them. Simultaneously the impact on the *h* and *k* variables are the same inside main function or outside main function. Where outside is the *multiplication* function.

Overall by not taking them as “pass by value” it saves memory. Because “pass by reference” does not require taking copy of the initialization of the variables, as it concurrently affects them wherever they are declared.

However, what if we want function to not take unwanted superfluous expensive memory by making copy of values at the same time not affecting them in main function? Answer: functions with const parameters where argument values are passed by reference!

1. First program won’t run because as the function is called and inside the parameter const arguments are passed by reference which makes unable to change the variables itself inside *multiplication* function. Consequently, the error occurs.
2. Second program however will run because it does not make any impact on variables globally which emphasize it functionality to get result by making operation with them inside *multiplication* function. Subsequently the function does not take any copies of the variables and sequentially does not use expensive memory, moreover it does not change the values of variables, namely in our case in main function.

1. Program

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | #include <iostream>  using namespace std;  void multiplication(const double& h, const double& k)  {  h = 2 \* h;  k = 2 \* k;  cout << "h = " << h << endl;  cout << "k = " << k << endl;  }  int main()  {  double h = 0.5, k = 4;  multiplication(h, k);  cout << "h = " << h << endl;  cout << "k = " << k << endl;  return 0;  } | Error on lines 5, 6 for making changes on constant values. | [Edit & Run](https://cplusplus.com/doc/tutorial/functions/) |

2. Program

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | #include <iostream>  using namespace std;  void multiplication(const double& h, const double& k)  {  cout << "h = " << 2 \* h << endl;  cout << "k = " << 2 \* k << endl;  }  int main()  {  double h = 0.5, k = 4;  multiplication(h, k);  cout << "h = " << h << endl;  cout << "k = " << k << endl;  return 0;  } | h = 1  k = 8  h = 0.5  k = 4 | [Edit & Run](https://cplusplus.com/doc/tutorial/functions/) |