When passing parameters to a function in C++, there is a variety of strategies a programmer can choose from. In this section, we will take a look at these in turn from the perspective of stack usage. First, we will briefly revisit the definition of scope, as well as the strategies call-by-value and call-by-reference. Then, we will look at the amount of stack memory used by these methods.

**Variable Scopes in C++**

The time between allocation and deallocation is called the **lifetime** of a variable. Using a variable after its lifetime has ended is a common programming error, against which most modern languages try to protect: Local variables are only available within their respective scope (e.g. inside a function) and are simply not available outside - so using them inappropriately will result in a compile-time error. When using pointer variables however, programmers must make sure that allocation is handled correctly and that no invalid memory addresses are accessed.

The example to the right shows a set of local (or automatic) variables, whose lifetime is bound to the function they are in.

When MyLocalFunction is called, the local variable isBelowThreshold is **allocated** on the stack. When the function exits, it is again **deallocated**.

For the allocation of local variables, the following holds:

1. Memory is allocated for local variables only after a function has been called. The parameters of a function are also local variables and they are initialized with a value copied from the caller.
2. As long as the current thread of execution is within function A, memory for the local variables remains allocated. This even holds true in case another function B is called from within the current function A and the thread of execution moves into this nested function call. However, within function B, the local variables of function A are not known.
3. When the function exits, its locals are deallocated and there is now way to them afterwards - even if the address were still known (e.g. by storing it within a pointer).

Let us briefly revisit the most common ways of passing parameters to a function, which are called pass-by-reference and pass-by-value.

### Quiz : How many local variables?[¶](https://r859981c930853xjupyterlykrdgbtw.udacity-student-workspaces.com/lab?#Quiz-:-How-many-local-variables?)

How many local variables are created within the scope of MyLocalFunction?

Two variables are created, namely myInt and isBelowThreshold.

**Variable Scopes in C++**

bool MyLocalFunction(int myInt)

{

bool isBelowThreshold = myInt < 42 ? true : false;

return isBelowThreshold;

}

int main()

{

bool res = MyLocalFunction(23);

return 0;

}

CONTINUE BELOW!!!!

## Passing Variables by Value

When calling a function as in the previous code example, its parameters (in this case myInt) are used to create local copies of the information provided by the caller. The caller is not sharing the parameter with the function but instead a proprietary copy is created using the assignment operator = (more about that later). When passing parameters in such a way, it is ensured that changes made to the local copy will not affect the original on the caller side. The upside to this is that inner workings of the function and the data owned by the caller are kept neatly separate.

However, there are two major downsides to this:

1. Passing parameters by value means that a copy is created, which is an expensive operation that might consume large amounts of memory, depending on the data that is being transferred. Later in this course we will encounter "move semantics", which is an effective way to compensate for this downside.
2. Passing by value also means that the created copy can not be used as a back channel for communicating with the caller, for example by directly writing the desired information into the variable.

Consider the example on the right in the pass\_by\_value.cpp file. In main, the integer val is initialized with 0. When passing it to the function AddTwo, a local copy of val is created, which only exists within the scope of AddTwo, so the add-operation has no effect on val on the caller side. So when main returns, val has a value of 2 instead of 4.

#include <iostream>

void AddTwo(int val)

{

val += 2;

}

int main()

{

int val = 0;

AddTwo(val);

val += 2;

std::cout << "val = " << val << std::endl;

return 0;

}

However, with a slight modification, we can easily create a backchannel to the caller side. Consider the code on the right.

In this case, when passing the parameter to the function AddThree, we are creating a local copy as well but note that we are now passing a pointer variable. This means that a copy of the memory address of val is created, which we can then use to directly modify its content by using the dereference operator \*.

#include <iostream>

void AddThree(int \*val)

{

\*val += 3;

}

int main()

{

int val = 0;

AddThree(&val);

val += 2;

std::cout << "val = " << val << std::endl;

return 0;

}

CONTINUE BELOW!!!

## Passing Variables by Reference

The second major way of passing parameters to a function is by reference. With this way, the function receives a reference to the parameter, rather than a copy of its value. As with the example of AddThree above, the function can now modify the argument such that the changes also happen on the caller side. In addition to the possibility to directly exchange information between function and caller, passing variables by reference is also faster as no information needs to be copied, as well as more memory-efficient.

A major disadvantage is that the caller does not always know what will happen to the data it passes to a function (especially when the function code can not be modified easily). Thus, in some cases, special steps must be taken to protect ones data from inappropriate modification.

Let us now look at an example of passing a variable by reference, shown in the code on the right.

To pass a variable by reference, we simply declare the function parameters as references using & rather than as normal variables. When the function is called, val will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument!

#include <iostream>

void AddFour(int &val)

{

val += 4;

}

int main()

{

int val = 0;

AddFour(val);

val += 2;

std::cout << "val = " << val << std::endl;

return 0;

}

### Quiz : Modifying several parameters[¶](https://r859981c930853xjupyterlykrdgbtw.udacity-student-workspaces.com/lab?#Quiz-:-Modifying-several-parameters)

An additional advantage of passing variables by reference is the possibility to modify several variables. When using the function return value for such a purpose, returning several variables is usually very cumbersome.

Your task here is to create a function AddFive that modifies the int input variable by adding 5 and modifies the bool input variable to be true. In the code to the right you will find the function call in main().

#include <iostream>

void AddFive(int &val, bool &success)

{

val+=5;

success = true;

}

int main()

{

int val = 0;

bool success = false;

AddFive(val, success);

val += 2;

std::cout << "val = " << val << ", success = " << success << std::endl;

return 0;

}

CONTINUE BELOW!!!

## Pointers vs. References

As we have seen in the examples above, the use of pointers and references to directly manipulate function arguments in a memory-effective way is very similar. Let us compare the two methods in the code on the right.

As can be seen, pointer and reference are both implemented by using a memory address. In the case of AddFour the caller does not even realize that val might be modified while with AddSix, the reference to val has to be explicitly written by using &.

If passing by value needs to be avoided, both pointers and references are a way to achieve this. The following selection of properties contrasts the two methods so it will be easier to decide which to use from the perspective of the use-case at hand:

* Pointers can be declared without initialization. This means we can pass an uninitialized pointer to a function who then internally performs the initialization for us.
* Pointers can be reassigned to another memory block on the heap.
* References are usually easier to use (depending on the expertise level of the programmer). Sometimes however, if a third-party function is used without properly looking at the parameter definition, it might go unnoticed that a value has been modified.

#include <iostream>

void AddFour(int &val)

{

val += 4;

}

void AddSix(int \*val)

{

\*val += 6;

}

int main()

{

int val = 0;

AddFour(val);

AddSix(&val);

std::cout << "val = " << val << std::endl;

return 0;

}

## Stack Usage of Function Calls

Now, we will compare the three strategies we have seen so far with regard to stack memory usage. Consider the code on the right.

After creating a local variable i in main to give us the address of the stack bottom, we are passing i by-value to our first function. Inside CallByValue, the memory address of a local variable j is printed to the console, which serves as a marker for the stack pointer. With the second function call in main, we are passing a reference to i to CallByPointer. Lastly, the function CallByReference is called in main, which again takes the integer i as an argument. However, from looking at main alone, we can not tell wether i will be passed by value or by reference.

On my machine, when compiled with g++ (Apple clang version 11.0.0), the program produces the following output:

stack bottom: 0x7ffeefbff698

call-by-value: 0x7ffeefbff678

call-by-pointer: 0x7ffeefbff674

call-by-reference: 0x7ffeefbff674

Depending on your system, the compiler you use and the compiler optimization techniques, you man not always see this result. In some cases

Let us take a look at the respective differences to the stack bottom in turn:

1. CallByValue requires 32 bytes of memory. As discussed before, this is reserved for e.g. the function return address and for the local variables within the function (including the copy of i).
2. CallByPointer on the other hand requires - perhaps surprisingly - 36 bytes of memory. Let us complete the examination before going into more details on this result.
3. CallByReference finally has the same memory requirements as CallByPointer.

### Quiz: Why does CallByValue require more memory?

In this section, we have argued at length that passing a parameter by reference avoids a costly copy and should - in many situations - be preferred over passing a parameter by value. Yet, in the experiment above, we have witnessed the exact opposite.

Can you explain why?

Let us take a look at the size of the various parameter types using the sizeof command:

printf("size of int: %lu\n", sizeof(int)); printf("size of \*int: %lu\n", sizeof(int \*));

The output here is

size of int: 4

size of \*int: 8

Obviously, the size of the pointer variable is larger than the actual data type. As my machine has a 64 bit architecture, an address requires 8 byte.

As an experiment, you could use the -m32 compiler flag to build a 32 bit version of the program. This yields the following output:

size of int: 4

size of \*int: 4

In order to benefit from call-by-reference, the size of the data type passed to the function has to surpass the size of the pointer on the respective architecture (i.e. 32 bit or 64 bit).

#include <stdio.h>

void CallByValue(int i)

{

int j = 1;

printf ("call-by-value: %p\n",&j);

}

void CallByPointer(int \*i)

{

int j = 1;

printf ("call-by-pointer: %p\n",&j);

}

void CallByReference(int &i)

{

int j = 1;

printf ("call-by-reference: %p\n",&j);

}

int main()

{

int i = 0;

printf ("stack bottom: %p\n",&i);

CallByValue(i);

CallByPointer(&i);

CallByReference(i);

return 0;

}