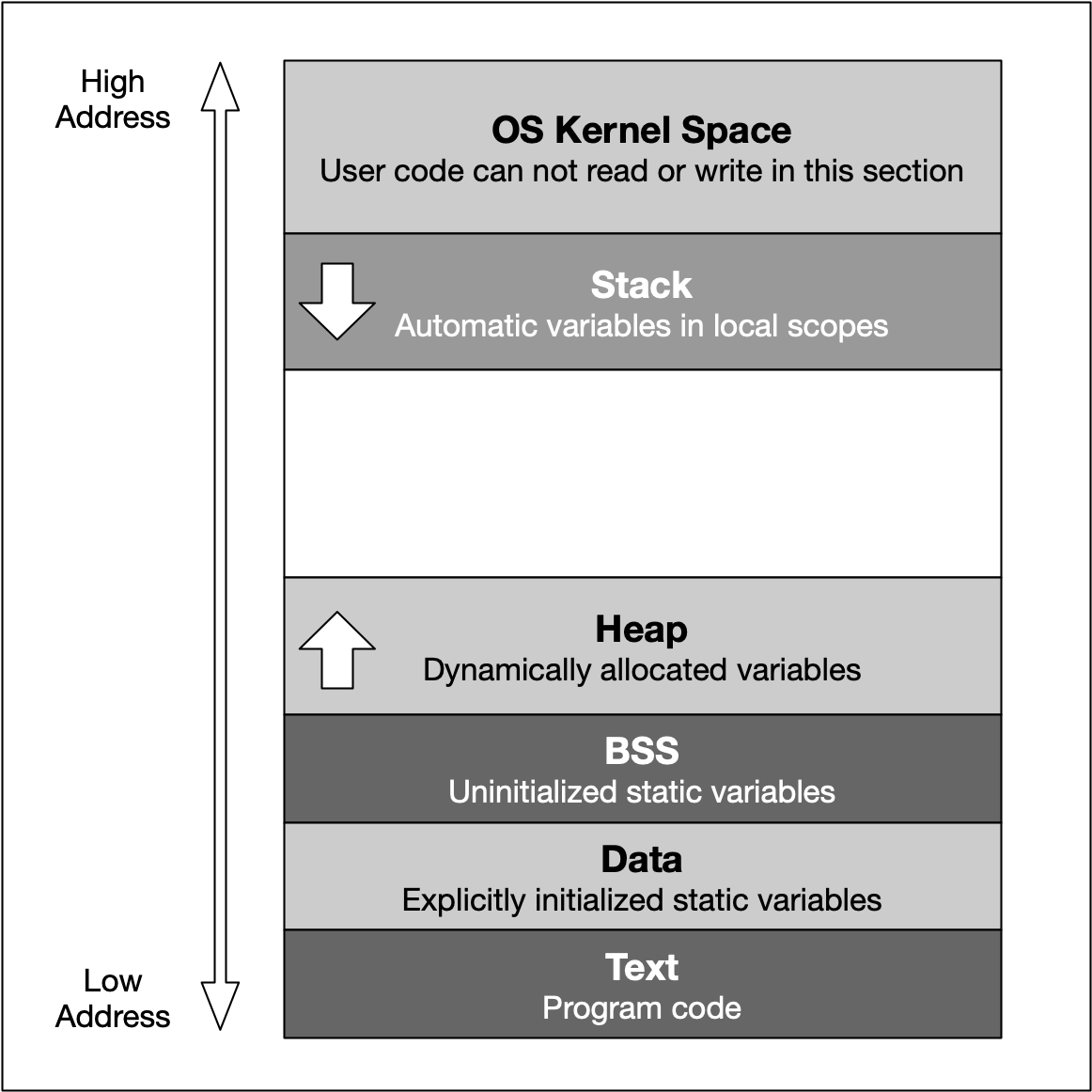
The Process Memory Model

As we have seen in the previous lesson, each program is assigned its own virtual memory by the operating system. This address space is arranged in a linear fashion with one block of data being stored at each address. It is also divided into several distinct areas as illustrated by the figure below:

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/789a1625-9b09-4615-9210-ddbc12e9247b/modules/b2145e6c-f349-4071-b1a5-682cda25eba8/lessons/3d93a126-42ab-4b02-8f13-b39eb91149c0/concepts/d1025d19-ba85-47e1-a60d-f7450cb6a4b5)](https://classroom.udacity.com/nanodegrees/nd213/parts/789a1625-9b09-4615-9210-ddbc12e9247b/modules/b2145e6c-f349-4071-b1a5-682cda25eba8/lessons/3d93a126-42ab-4b02-8f13-b39eb91149c0/concepts/d1025d19-ba85-47e1-a60d-f7450cb6a4b5)

The last address 0cFFFFFFFF converts to the decimal 4.294.967.295 , which is the total amount of memory blocks that can theoretically addressed in a 32 bit operating system - hence the well-known limit of 4GB of memory. On a 64 bit system, the available space is significantly (!) larger. Also, the addresses are stored with 8 bytes instead of 4 bytes.

From a programming perspective though, we are not able to use the entire address space. Instead, the blocks "OS Kernel Space" and "Text" are reserved for the operating system. In kernel space, only the most trusted code is executed - it is fully maintained by the operating system and serves as an interface between the user code and the system kernel. In this course, we will not be directly concerned with this part of memory. The section called 'text' holds the program code generated by the compiler and linker. As with the kernel space, we will not be using this block directly in this course. Let us now take a look at the remaining blocks, starting from the top:

1. The stack is a contiguous memory block with a fixed maximum size. If a program exceeds this size, it will crash. The stack is used for storing automatically allocated variables such as local variables or function parameters. If there are multiple threads in a program, then each thread has its own stack memory. New memory on the stack is allocated when the path of execution enters a scope and freed again once the scope is left. It is important to know that the stack is managed "automatically" by the compiler, which means we do not have to concern ourselves with allocation and deallocation.
2. The heap (also called "free store" in C++) is where data with dynamic storage lives. It is shared among multiple threads in a program, which means that memory management for the heap needs to take concurrency into account. This makes memory allocations in the heap more complicated than stack allocations. In general, managing memory on the heap is more (computationally) expensive for the operating system, which makes it slower than stack memory. Contrary to the stack, the heap is not managed automatically by the system, but by the programmer. If memory is allocated on the heap, it is the programmer’s responsibility to free it again when it is no longer needed. If the programmer manages the heap poorly or not at all, there will be trouble.
3. The BSS (Block Started by Symbol) segment is used in many compilers and linkers for a segment that contains global and static variables that are initialized with zero values. This memory area is suitable, for example, for arrays that are not initialized with predefined values.
4. The Data segment serves the same purpose as the BSS segment with the major difference being that variables in the Data segment have been initialized with a value other than zero. Memory for variables in the Data segment (and in BSS) is allocated once when a program is run and persists throughout its lifetime.

Memory Allocation in C++

Now that we have an understanding of the available process memory, let us take a look at memory allocation in C++.

Not every variable in a program has a permanently assigned area of memory. The term allocate refers to the process of assigning an area of memory to a variable to store its value. A variable is deallocated when the system reclaims the memory from the variable, so it no longer has an area to store its value.

Generally, three basic types of memory allocation are supported:

1. Static memory allocation is performed for static and global variables, which are stored in the BSS and Data segment. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
2. Automatic memory allocation is performed for function parameters as well as local variables, which are stored on the stack. Memory for these types of variables is allocated when the path of execution enters a scope and freed again once the scope is left.
3. Dynamic memory allocation is a possibility for programs to request memory from the operating system at runtime when needed. This is the major difference to automatic and static allocation, where the size of the variable must be known at compile time. Dynamic memory allocation is not performed on the limited stack but on the heap and is thus (almost) only limited by the size of the address space.

From a programmer’s perspective, stack and heap are the most important areas of program memory. Hence, in the following lessons, let us look at these two in turn.

Properties of Stack Memory

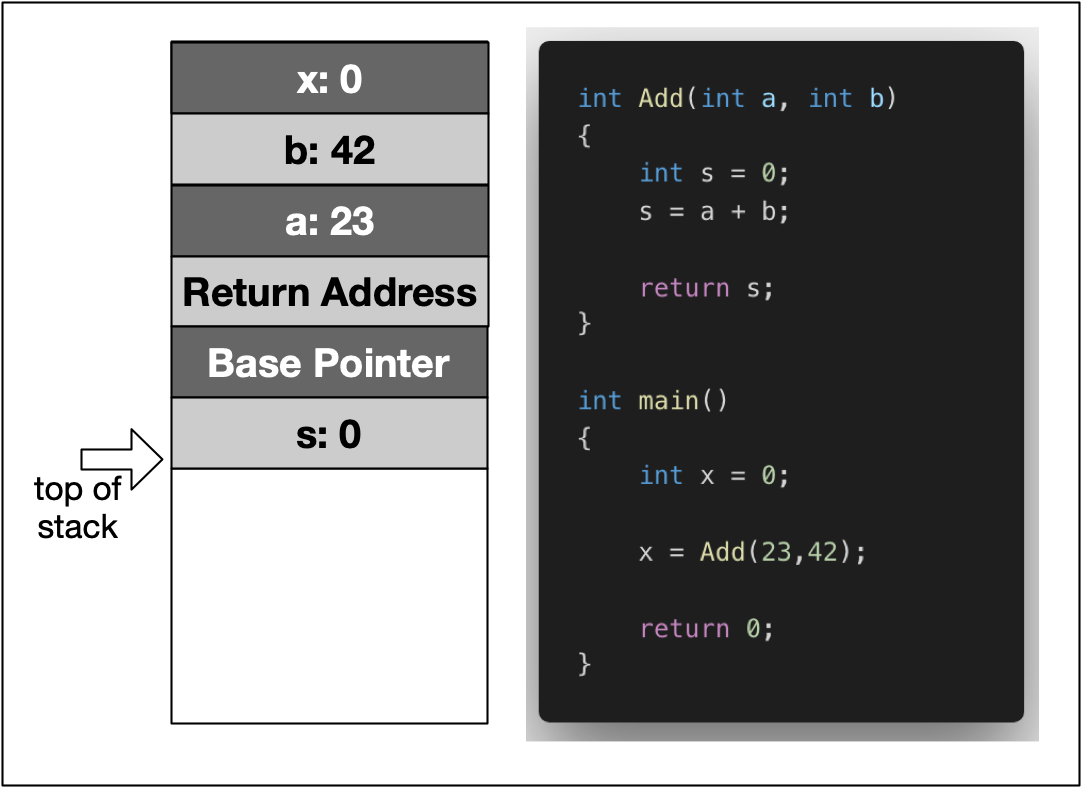
In the available literature on C++, the terms *stack* and *heap* are used regularly, even though this is not formally correct: C++ has the *free space*, *storage classes* and the *storage duration* of objects. However, since stack and heap are widely used in the C++ community, we will also use it throughout this course. Should you come across the above-mentioned terms in a book or tutorial on the subject, you now know that they refer to the same concepts as stack and heap do.

As mentioned in the last section, the stack is the place in virtual memory where the local variables reside, including arguments to functions. Each time a function is called, the stack grows (from top to bottom) and each time a function returns, the stack contracts. When using multiple threads (as in concurrent programming), it is important to know that each thread has its own stack memory - which can be considered thread-safe.

In the following, a short list of key properties of the stack is listed:

1. The stack is a contiguous block of memory. It will not become fragmented (as opposed to the heap) and it has a fixed maximum size.
2. When the maximum size of the stack memory is exceeded, a program will crash.
3. Allocating and deallocating memory is fast on the stack. It only involves moving the stack pointer to a new position.

The following diagram shows the stack memory during a function call:

[[](https://classroom.udacity.com/nanodegrees/nd213/parts/789a1625-9b09-4615-9210-ddbc12e9247b/modules/b2145e6c-f349-4071-b1a5-682cda25eba8/lessons/3d93a126-42ab-4b02-8f13-b39eb91149c0/concepts/9cc53eda-5e84-4463-b968-1b1324f6a7c3)](https://classroom.udacity.com/nanodegrees/nd213/parts/789a1625-9b09-4615-9210-ddbc12e9247b/modules/b2145e6c-f349-4071-b1a5-682cda25eba8/lessons/3d93a126-42ab-4b02-8f13-b39eb91149c0/concepts/9cc53eda-5e84-4463-b968-1b1324f6a7c3)

In the example, the variable x is created on the stack within the scope of main. Then, a stack frame which represents the function Add and its variables is pushed to the stack, moving the stack pointer further downwards. It can be seen that this includes the local variables a and b, as well as the return address, a base pointer and finally the return value s.

In the following, let us dig a little more deeply and conduct some experiments with variables on the stack.

Stack Growth and Contraction

In the first experiment, we will look at the behavior of the stack when local variables are allocated and a function is called. Consider the piece of code on the right.

Within the main function, we see two declarations of local variables i and j followed by a call to MyFunc, where another local variable is allocated. After MyFunc returns, another local variable is allocated in main. The program generates the following output:

1: 0x7ffeefbff688

2: 0x7ffeefbff684

3: 0x7ffeefbff65c

4: 0x7ffeefbff680

Between 1 and 2, the stack address is reduced by 4 bytes, which corresponds to the allocation of memory for the int j.

Between 2 and 3, the address pointer is moved by 0x28. We can easily see that calling a function causes a

significant amount of memory to be allocated. In addition to the local variable of MyFunc, the compiler needs to store additional data such as the return address.

Between 3 and 4, MyFunc has returned and a third local variable k has been allocated on the stack. The stack pointer now has moved back to a location which is 4 bytes relative to position 2. This means that after returning from MyFunc, the stack has contracted to the size it had before the function call.

The following diagram illustrates how the stack grows and contracts during program execution:

#include <stdio.h>

void MyFunc()

{

int k = 3;

printf ("3: %p \n",&k);

}

int main()

{

int i = 1;

printf ("1: %p \n",&i);

int j = 2;

printf ("2: %p \n",&j);

MyFunc();

int l = 4;

printf ("4: %p \n",&l);

return 0;

}

## Total Stack Size

When a thread is created, stack memory is allocated by the operating system as a contiguous block. With each new function call or local variable allocation, the stack pointer is moved until eventually it will reach the bottom of said memory block. Once it exceeds this limit (which is called "stack overflow"), the program will crash. We will try to find out the limit of your computer’s stack memory in the following exercise.

### Exercise: Create a Stack Overflow

Your task is to create a small program that allocates so much stack memory that an overflow happens. To do this, use a function that allocates some local variable and calls itself recursively. With each new function call, the address of the local variable shall be printed to the console along with the address of a local variable in main which has been allocated before the first function call.

The output of the program should look like this:

...

262011: stack bottom : 0x7ffeefbff688,

current : 0x7ffeef400704 262012: stack bottom : 0x7ffeefbff688, current : 0x7ffeef4006e4 262013: stack bottom : 0x7ffeefbff688, current : 0x7ffeef4006c4 262014: stack bottom : 0x7ffeefbff688, current : 0x7ffeef4006a4 262015: stack bottom : 0x7ffeefbff688, current : 0x7ffeef400684 262016: stack bottom : 0x7ffeefbff688, current : 0x7ffeef400664

The left-most number keeps track of the recursion depth while the difference between the stack bottom and the current position of the stack pointer lets us compute the size of the stack memory which has been used up already. On my MacBook Pro, the size of the stack memory is at 8MB. On Mac or Linux systems, stack size can be checked using the command ulimit -s:

imac-pro:~ ahaja$ ulimit -s

8192

On reaching the last line in the above output, the program crashed. As expected, the difference between stack bottom and current stack pointer corresponded to the maximum size of the stack: 0x7ffeef400664 - 0x7ffeefbff688 = 0xffffffffff800FDC = 8.384.548 bytes

From this experiment we can draw the simple conclusion that we do not want to run out of stack memory. This can happen quickly though, even on machines with large amounts of RAM installed. As we have seen, the size of the stack does not benefit from this at all but remains fixed at a very small size.

#include <stdio.h>

int id = 0;

void MyRecursiveFunc(int &i)

{

int j = 1;

printf ("%d: stack bottom : %p, current : %p\n",id++, &i, &j);

MyRecursiveFunc(i);

}

int main()

{

int i = 0;

MyRecursiveFunc(i);

return 0;

}

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
3. 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.
4. 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).

bool MyLocalFunction(int myInt)

{

bool isBelowThreshold = myInt < 42 ? true : false;

return isBelowThreshold;

}

int main()

{

bool res = MyLocalFunction(23);

return 0;

}

## 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

#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;

}

## 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 loo

#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;

}

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

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;

}

## 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

#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;

}

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.

#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;

}

### 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