Memory Management

Samuel Navarro

August 21, 2019

Contents

Pointers literally points to a memory address. References are aliases for other objects. In practice, compilers typically implement references using the memory address of an object so *pointers* and *references* have a complementary role.

Pointers point to a memory address and *references* refer to whatever is stored in that memory address but conceptually, a *reference* for a variable is just an alias for that variable.

Bjarne on RAII is the idea that when you come in scope (a function or whatever) you initialize an object, hold those resources, and at the end of the scope they get released automatically.

In general, memory management in C++ can be split into two major categories by their storage in the host machine. When we analyze this classification, we have two main concepts: stack and heap.

The Stack

1. only local variables
2. resizing of variables cannot be done
3. efficiently managed space by CPU, memory does not become fragmented
4. very quick access
5. there is a limit on stack size (OS-dependent)
6. explicit de-allocation of variables is not mandatory

The Heap

1. memory must be managed
2. memory size is not limited
3. access is relatively slower
4. realloc() is used for the resizing of variables
5. variables can be accessed globally
6. efficient use of space is not guaranteed, memory can become fragmented over time when blocks of memory are allocated, then freed

Bjarne on Stack and Heap What is the difference between the stack and the heap?

When you call a function it increases the stack with a stack frame that represent the function. You call another function and you go up, when you return you go back again.

But if you want data that lives from one function to another you have to have it somewhere else. *(In principle you can refer stack down in the frame but you can't refer up)* so you have memory on the side, give me some of that and I'll keep it and when I finish I will give it back. That's known as the heap or the dynamic memory or the free store. You get it with an operation call `new`. And then, eventually you have a `delete` that releases this memory (it gets recycled but explicitly).

1 Pointers

A **pointer** is a primitive data type that keeps track of a memory address. It's designated with the dereference operator `*`. A **pointer** provides access to 2 pieces of information, a memory address and a value.

Bjarne on pointers: All the data in memory are sequences of objects and you can refer to them by pointers, which are machine addresses with a type. In memory **pointers** and **references** look the same.

A **pointer** you then associate, *at compile time*, the type with it, so that you know that you are pointing to `int, float`, etc.

2 References

References are declared by the ampersend operator `&`.

As a rule of thumb, they are safer than pointers because they are less prone to memory leaks.

Some of the typical deficiencies of references in compare with pointers are:

- When reference is created it cannot be re - initialized
- References cannot be initialized to NULL value, which is often used to indicate that pointers aren't showing to any memory location
- A reference must be initialized with valid data - meaning that they are only declared when they have valid memory address to reference

When declare a reference to specific data in C++, we are inheriting all related data for that specific variable. So, besides the basic value of that

```
int var = 10;
// we are requesting value of variable var:
std::cout<<var<<std::endl;
// 10
// we are requesting memory address of variable var:
std::cout<<&var<<std::endl;
// e.g. => 0x7ffe318d531c
```

Listing 1: Operation of Referencing

data, we are inheriting information about memory location of that specific value. Having that in mind, when we perform any data manipulation of our created reference, we are affecting our initial variable. Explanation of that behaviour comes from a fact that reference is just a representation of a variable upon which we declared our reference. In specific point of view, we can imagine references as primitive pointers, because they also carry information about data memory location. Besides that, reference is also a operator, which can be used for operation of referencing. Referencing data in C++ states that we are requesting memory location of wanted data.

References are much more efficient for usage in C++, because they are preventing memory leakage, they are safer from a point of memory management and they are easier to use. Reference are popular as input and return values in functions. With this technique, we are preventing copying of our data from one part of program to another and we are certain that any changes in a function will affect our original data.

Bjarne on References: A lot of the times you want to point to something. One of the reasons is that you don't want to move the stuff around all the time. In C++ **pointers** and **references** are roughly the same except that is easy to use a reference. You don't have to say, go ahead and get it. You just use it.

Another reason is that a **reference** cannot be made to refer to a different object.

2.1 References example

We convert the `string` to c-string a lot of the times (Why? - the reason the instructor give is that for example Windows api even requires you to use c-strings).

This line:

```
const char* ptr = str.c_str();
```

Gives us the address of the first element in the string because the c string is basically a character array.

The addSpaces function makes operations on the string just by knowing its address.

3 New and Delete

C++ comes with two types of memory allocation: static and dynamic.

Static memory allocation occurs at compile time. Dynamic memory allocation occurs at run time; the compiler does not know whether or how much data will be required, so the program itself has to allocate this memory when it runs. For example, you are programming a game and you don't know how many players there will be ahead of time. You may use dynamic memory to create objects for each players as they come to the game.

Dynamic memory is allocated on the heap, which is often referred as the *free store.*

Bjarne on new and delete: My rule of thumb is don't use `new` and `delete` in application code. (?) `new` and `delete` belongs in the implementations of you abstractions. So, if you use vector and string, you don't use new and delete because that is hidden inside the abstraction.

3.1 Description of New

Before any action, couple of requests needs to be fulfilled for successful allocation. First of all we need to have sufficient memory and we need to know which type of data (what size of memory) we are reserving for our usage. After these requirements are fulfilled we can perfom memory allocation. Next action is allocation of memory and return of memory address in heap. General purpose syntax is:

```
pointer = new type-specifier
```

In this context we can see that pointer is used to access this newly created data. We are using pointers because this performed actions returns memory

```cpp
// Example program
#include <iostream>

int main()
{
    double *ptr = new double(23);
    double *arr_ptr =  new double[4];
    for( int i = 0; i<3; i++) {
        *ptr = ++ *ptr;
        std::cout<< "Address: "<<ptr<<" Value: "<<*ptr<<std::endl;
        std::cout<< "Address array: "
                                <<arr_ptr<<" Value array: "
                                <<arr_ptr[i]<<std::endl;
    }
}
// OUTPUT:
/*
Address: 0x1a809e0 Value: 24
Address array: 0x1a80a00 Value array: 0
Address: 0x1a809e0 Value: 25
Address array: 0x1a80a00 Value array: 0
Address: 0x1a809e0 Value: 26
Address array: 0x1a80a00 Value array: 0
*/
```

Listing 2: New and Delete Code

address. Technique for usage of memory addresses in C++ is usage of pointers. Next in line is type-specifier. This can be any user defined or default defined data type (classes, structures, char, int, float, double). Besides that, data types could be array like structures, where we can defined arrays of mentioned data types.

The point of this example is to understand that the variable is in the same address but it's value is changing.

The dereference operator * is used to get or access the value.

3.2 Delete Operator

When you use `delete` operator on a given memory address, this means that compiler is going to this address in memory and deleting its content where the process is finished when this part of memory is marked as unoccupied (free).

```
delete pointer-variable;
```

Releasing block of memory, occupied by array - like data structure is done similarly as complementary allocation. We use square brackets (" [ ] "), to signal that we are deallocating block of memory which allocated on a given address.

```
delete[] pointer-variable;
```

Bjarne on Memory Leaks: You get something from the heap and then you throw your pointer away. Now, nobody know that this `int` is still sitting over there.

4 Memset and Malloc

4.1 Memset

The prototype of memset is:

```
void * memset ( void * ptr, int value, size_t num );
```

where ptr is memory address of memory we want to initialize, value is value which will be used to set this memory space. The value is passed as an int, however it is first converted to unsigned char and then placed by the function into the memory block. num is used to signal how many bytes we are initializing.

Memset is usually used for initialization of values. When we take into consideration that we can use following data structure:

And if declare instance of Point data type `Point p1;`

Then, char values in this structure are going to be undefined or have some values from the last usage of that part of the stack. They are un predictable. Having that in mind we can use memset to set default values for this data type. We need to have in mind that this concept and approach is also used

```cpp
// Illustration of memory allocation and deallocation
int main() {
int *ptr = nullptr;
/*
When using new, best practice in coding states that we need
to enclose it within a try-catch block. The new operator throws
an exception and does not return a value. To force the new
operator to return a value, you canuse the nothrow
qualifier as shown below:
*/
ptr = new(std::nothrow) int;
if (!ptr) {
 std::cout << "Mem alloc failed!" << std::endl;
}
else {
    // assigning value to newly allocated address
    *ptr = 31;
    // checking our pointer state:
    std::cout<<" Address is: " << ptr << std::endl;
    std::cout<<" Value is: " << *ptr << std::endl;
}

// We are creating pointer to array of integers
int *arr_ptr = new(std::nothrow) int[3];
// We are storing new values to created array
// 0 1 4
for (int i=0; i<3; i++)
    arr_ptr[i] = (i*i);

// Writing our arr_ptr pointer info:
for (int i=0; i<3; i++) {
    // notice notation for retrieval of memory address
    // in array pointers ( we are using references)
    std::cout<<" Address is: " << &arr_ptr[i] << std::endl;
    std::cout<<" Value is: " << arr_ptr[i] << std::endl;
}

// Before our program is finished, we have responsibility
// to deallocate all of our allocated memory:

// integer pointer
delete ptr;
// array of integers pointer - block of memory
delete[] arr_ptr;

return 0;
}
```

Listing 3: Delete Operator

```
struct Point{

  char x;
        char y;
};
```

Listing 4: Memset example

because memset is significantly faster function then user defined ones. So our code could look something like this:

Note: we need to include $< cstring >$ library or $< string.h >$ for usage of memset

```
memset(&p1, '-', sizeof(p1));
```

4.2 Malloc

Its function is to allocate a block of specific **size**, measured in bytes of memory, where result of this operation (function) is pointer to the beginning of the block. Data stored in this memory block is not initialized, which means that this space has non determinate values. Prototype of this function is:

```
void* malloc(size_t size);
```

Size is unsigned integer value which represent memory block in bytes. It is important to note that in case of failed allocation return value is null pointer.

malloc is not something that we really use. `malloc` is basically a forerunner to `new`. We'll use `new`. We'll not use `malloc`. But `malloc` will be in legacy code.

One thing to consider is that we need to write for loops to check the size:

5 Scope

In general, scope can be viewed as the extent up to which some resource can be used and worked with. In programming also scope of a variable is

```cpp
// Included library for memset
#include <cstring>
// included library for malloc
#include <cstdlib>
int main() {
    int *ptr = NULL;
    // Notice how pointer is allocated to single int memory space
    // but later is redicerted to show to memory block
    ptr = new(std::nothrow) int;
    //  We can initialize memory block using malloc
    //  declaring memory space for array of 4 integers
    ptr = (int*)std::malloc(4*sizeof(int));
        for (int i=0; i<4; i++){
        std::cout << "Address: "
        << &ptr[i] << " Value: "<< ptr[i] << std::endl;
    }
    // every memory place is used for initialization of specific value
    // here we are using letters for values, but casted to ints
    for( int i=0; i<4; i++){
        // sending address by reference
        std::memset(&ptr[i],('A' + i), sizeof(int));
    }
    // Printing state of our array
    // we are converting int values to char to see what
    // is happening with our values.
    for (int i=0; i<4; i++){
        std::cout << "Address: " << &ptr[i]
        << " Value: "<< char(ptr[i])
        << std::endl;
    }
    /* OUTPUT:
    Address: 0x1031c20 Value: 0
    Address: 0x1031c24 Value: 0
    Address: 0x1031c28 Value: 0
    Address: 0x1031c2c Value: 0
    Address: 0x1031c20 Value: A
    Address: 0x1031c24 Value: B
    Address: 0x1031c28 Value: C
    Address: 0x1031c2c Value: D
    */
    return 0;

                AUNQUE FALTA delete

}
```

Listing 5: Malloc Example

```
// We need to cast the return value of malloc because
// the return value of malloc is a pointer to the
// first spot of the block. It'll contain
// a pointer which will be of type void.
// So I need to cast it to int
ptr = (int*)std::malloc(5*sizeof(int));

for (int i=0; i<500; i++){
 // ESTO ES LO QUE TENEMOS QUE REVISAR
 // PORQUE EL TAMAÑO ERA 5 * sizeof(int)
}
```

Listing 6: Malloc Checks

```
#include <iostream>
// Global variable declaration:
int glob;
int main () {
    glob = 10;
    glob ++;
    std::cout << glob << std::endl;
    // OUTPUT: 11
    return 0;
}
```

Listing 7: Global Variable

defined as the extent of the program code within which the variable can we accessed or declared or worked with. A scope can be defined by many ways : it can be defined by namespace, functions, classes and just .

Global Variable

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

There are also **Local Variables and Namespaces**

Namespaces were introduced in C++, as a new feature not inherited from C. Namespaces are used for definition of scopes, where we can create our on namespaces within same program, which have they own scope.

This is a general example of **Scope**:

```cpp
// Variable created inside namespace
namespace myNamespace
{
    int val = 123;
}

// Global variable
int val = 100;

int main() {
    // Local variable
    int val = 200;

    // These variables can be accessed from
    // outside the namespace by using the scope
    // operator :: -> also known as scope resolution operator

    // namespace variable value
    std::cout << myNamespace::val << std::endl;
    // global variable value
    std::cout << ::val << std::endl;
    // local variable value
    std::cout << val << std::endl;

    // OUTPUT:
    // 123
    // 100
    // 200
    return 0;
}
```

Listing 8: Namespaces

```
int val;
//this variable val is defined in global namespace
// which means, its scope is global. It exists
// everywhere.

namespace _namespace
{
     int val;
    // it is defined in a local namespace called '_namespace'
    // outside _namespace it doesn't exist.
}
void _func()
{
   int val;
    // scope is the function itself.
    // outside the function, a doesn't exist.
   {
        int val; //the curly braces defines this scope
   }
}
class _class
{
   int val;
    // scope is the class itself.
    // outside class _class, it doesn't exist.
};
```

Listing 9: Scope