1 Introduction

What makes C++ (and C) different from many other modern programming languages is the fact the memory management is left to a large extent in the hands of the programmers themselves.

If you have a background in Java, Python or C# (to name a few) then you are likely used to not having to worry about when an object is destroyed since these languages are *Garbage Collected*. In such languages you can just create objects, usually with an operator called `new`, and then leave it up to the garbage collector to determine when that object should be destroyed. This means that you don't ever have to worry about destroying objects, nor do you have to consider whether objects still exist or not since the garbage collector will keep objects alive as long as they are *reachable* in your code. This is not the case in C++.

C++ prioritizes efficiency and control over ease-of-use. A garbage collector makes code easy to write and manage since there is no need to keep track of how the memory is used. But this does not come for free. A garbage collector is a very complicated system that requires a lot of overhead, both in memory and in CPU cycles. Because of this (and other historic reasons) C++ has opted to not use a garbage collector and instead leave the issue of memory management for the programmer to solve themselves.

This might seem like a bad thing, but not using a garbage collector comes with many benefits. It allows the programmer to tailor their memory management for their specific problem or program. This can greatly benefit performance of said program since it does not require a large garbage collection system to run in the background. This frees up more CPU cycles to be used by the actual program.

Another benefit is the fact that the programmer has full control of when an object is created *and* destroyed. This can allow for greater fine-tuning, but can also allow the program to have better understood behavior which makes debugging a lot easier.

Because of this it is important to understand how memory works, how it is managed and how to access it. Knowing this will also help you make better decisions in your code. This tutorial aims to be a (somewhat) complete guide to memory and memory management.

2 What is memory?

There are two things we usually refer to when we say "memory". The first one is data stored on a harddrive. This is usually structured into *files* that are managed by the file system. This is *not* what we mean when we say memory.

Whenever we say "memory" in the context of C++ (and programming in general) we refer to the *Main Memory* (or *RAM*) where data related to a currently running program is stored. Things like variables, data related to function calls and the machine code for the currently running program are stored here.

We can think of memory as a sequence of *bytes*. A *byte* can be thought of as an integer that can store a value between `0` and `255` (it consists of 8 *bits*). So memory is essentially an "array" of bytes. This means that each byte has an index which is called its *address*.

On 32-bit systems an address takes 32-bits (4 bytes) while on a 64-bit system an address takes (you guessed it) 64-bits (8 bytes). We are going to assume that an address takes 64-bits to represent, but keep in mind that this can differ between different platforms.

So an address can refer to 2^{64} different bytes (i.e. 16 billion gigabytes, which is a lot). This is more than enough to refer to all available RAM on a given computer. Note that for 32-bit systems we can only refer to $2^{32} = 4$ gigabytes. This is not enough for modern computers which is why 64-bit systems are the current standard.

3 So how is memory used by the program?

The operating system will distribute the memory to all programs running on the computer. Each program only has access to a its own portion of the memory. On older systems programs had access to all the available memory, but this lead to a lot of security issues. Because of this modern operating systems are designed to only allow a program to access its own assigned memory. The operating system is responsible for making sure that each currently running program has access to memory. It also controls how much memory they have access to and so on.

If a program tries to access (either by writing to or reading from) memory outside of its assigned portion, the operating system will immediately kill the program and give an error. This error is commonly called `segmentation fault` on Linux.

Each program need access to at least some memory since all local variables and each function call requires memory. So if the program runs out of memory it will result in severe errors. So we need to understand how this works to ensure that the programs we write plays nicely with the operating system.

3.1 Wait... What do you mean "function calls requires memory"?

The fact that a function call requires memory might seem a bit unintuitive since functions are just code, but it is true. Think of it like this: when a function gets called the program starts executing code from somewhere else (i.e. the code inside the function). But once the function is complete the program has to jump back to the position in the code where the function was called. Consider the following example:

```
1  #include <iostream>
2
3  void fun()
4  {
5    std::cout << "Function called!" << std::endl;
6  }
7
8  int main()
9  {
10   std::cout << "Program start..." << std::endl;
11   fun();
12   std::cout << "Program end!" << std::endl;
13 }
```

The program starts execution on line `10` and will continue executing the code inside `main()` until it reaches line `11`. Here the program will abruptly jump to line `5` and start executing the `fun()` function. Once `fun()` is done, the code needs to jump back to where it came from, meaning it has to go back to line `11` and continue execution from that point onward.

For the program to be able to do this it has to remember where it needs to jump after a function call is done. This is more clearly demonstrated in an example where a function is called more than once, like this:

```
1  void fun()
2  {
3    std::cout << "Function called!" << std::endl;
4  }
5
6  int main()
7  {
8    std::cout << "First call: " << std::endl;
9    fun();
10   std::cout << "Second call: " << std::endl;
11   fun();
12 }
```

Here `fun()` is called twice, once on line `9` and once on line `11`. Each of these calls requires the program execution to continue in different places. Hopefully this clarifies that each time a function gets called the program *have* to remember where to jump back, since this location might differ from call to call. The only way it can remember this is by using the memory.

3.2 OK... But what about local variables?

Local variables that are declared inside a function also have to be stored in memory. We cannot store one set of local variables per function, we need to do it *per function call*. Why is that?

A function might call itself (called *recursion*) and in that case each time it calls itself it will need to have a new set of variables since those values might be different for each call. Due to recursion it might also be the case that once a function call returns it might end up in a previous call to the same function, which of course have different values for the local variables.

This is more easily explained with an example. Consider this recursive function:

```
1  int factorial(int n)
2  {
3    if (n == 0)
4      return 1;
5    else
6      return n * factorial(n - 1);
7  }
```

Note that parameters that are passed to functions are considered local variables.

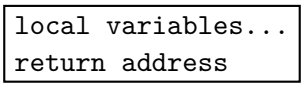
If we now call `factorial(3)` the program will jump to the `factorial` code, and set `n` to be `3`. Since `n != 0` the program will take the else-branch and call factorial *again* but this time with `n = 2`. Once this function call is done the program has to jump back to the previous call to `factorial` where `n = 3`. Because of this each call to a function must store its own copy of `n`, because otherwise this information would be lost in the recursive calls.

So there are potentially a lot of things that needs to be stored for each *call* to a function, more specifically it needs to store where the function was called from, and all of its parameters and local variables. Most platforms solves this by using a *stack* structure inside the memory.

4 The stack

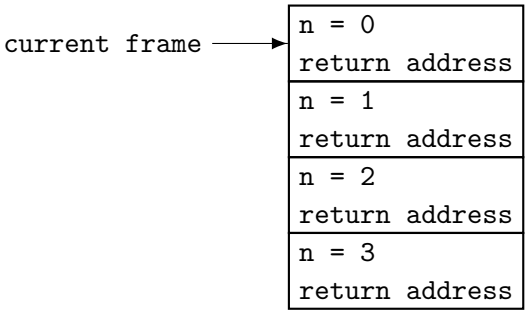
Most (almost all) platforms uses a similar way to structure the memory of a program in order to solve the problems presented in the previous section(s).

Each function call will store what is called a *Stack Frame* in the memory. A Stack Frame is just a sequence of memory where all local variables are stored as well as the *return address* (the place in the code where the function should jump to once it is done). In this tutorial we will represent Stack Frames like this:

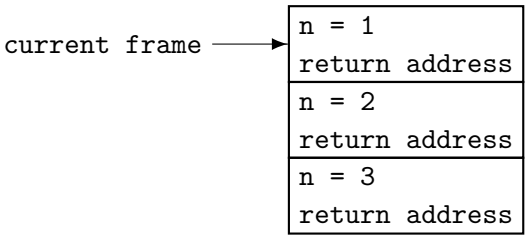


The Stack simply refers to the portion of a programs memory that contains these stack frames. They are always stored in sequence in the memory, which means that whenever a function gets called its stack frame is placed on the next available position in the memory.

As an example let us consider the `factorial` function defined in the previous section. If we call `factorial(3)`, once we reach the call where `n = 0` we will have the following stack frame:



When `factorial(0)` returns, the program execution will jump to the place in the code specified by the current stack frames return address, and then the previous stack frame will be the active one instead. Meaning it would look like this:



Most variables we create will be stored on the stack in a stack frame for some function. Remember that `main` is a function which of course also has a stack frame. In fact all code that gets executed in C++ must reside in a function call.

We say that local variables have *automatic storage duration*, meaning that the variable will be destroyed when the stack frame it resides in is destroyed (which is handled automatically when we return from a function).

More specifically: A variable with automatic storage duration will only exist for as long as its enclosing scope is being executed (curly brackets).

4.1 But what about global variables?

One can argue that global variables are the complete opposite of local variables, Global variables are not bound to a specific function call. Their value is the same across the whole code base and if we change it somewhere that change should be reflected everywhere. Meaning we cannot store global variables in a stack frame the same way we do with local variables.

Here is an example of a global variable:

```
1  // global variable
2  int global { 0 };
3
4  void fun(int n)
5  {
6    global += n;
7    std::cout << global << std::endl;
8  }
9
10 int main()
11 {
12   std::cout << global << std::endl; // will print 0
13   fun(3); // will print 3
14   fun(2); // will print 5
15   fun(0); // will print 5
16 }
```

Notice that the value of the global variable is retained between each function call, and is modified by each call so clearly it is not related to the local stack frames for each function call.

Side note: `global` is not truly a global variable since it is only available in this implementation files (`.cc` or `.cpp` file). To make it truly global (i.e. to make it available in all implementation files) we have to declare it in a header file (`.h` or `.hpp` file) as such: `extern int global;`

And then define it in *one* implementation file, like this: `int global { 0 };`

Then, and only then, will it be available for everyone that includes the header file that contains the declaration. This is related to linking, which is a part of the compilation process and not relevant for memory so we will leave it there.

4.2 So how do we store global variables?

Exactly where and how global variables are stored in memory varies from platform to platform. For the purposes of this tutorial it is not important to know exactly where in memory global variables are stored. However it is important for us to know that global variables have something called *static storage duration*.

Static storage duration means that the variable will be destroyed at the end of the program (after we have returned from `main`). Most variables that have static storage duration are also created at the start of the program (before `main` starts executing). There is however an exception, and that is something called *static local variables*.

4.3 What are static local variables?

To explain what static local variables are, let us look at an example:

```
1  void fun(int n)
2  {
3    static int variable { 0 };
4    variable += n;
5    std::cout << variable << std::endl;
6  }
7
8  int main()
9  {
10   fun(3); // will print 3
11   fun(2); // will print 5
12   fun(0); // will print 5
13 }
```

This has a very similar behavior to the previous example, but there are two major differences:

— `variable` is created once we call `fun` for the first time (while global variables are created before `main` is executed).

— `variable` is *only* accessible inside the `fun` function.

So a static local variable is like a global variable that is only available from its local scope (`fun` in this example) and is created the first time that scope is being executed.

5 Pointers

Now that we understand the stack and how variables are stored it is time to move on with our discussion on memory. Recall that the smallest building block of memory is a byte. Also remember that each byte can be uniquely identified by their *address*. A variable can also be identified by an address. However, a variable can take up more than one byte in memory. For example variables of type `int` typically occupies 4 bytes in memory. Each byte in an `int` has its own address which theoretically means that all four of those addresses can be used to identify the variable. But this way of thinking will quickly become problematic. To see why consider the following two questions:

— What is the value of the variable located at address `x`?

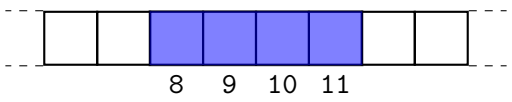
— Which byte in the variable is located at address `x`?

To answer these questions we need to know two things:

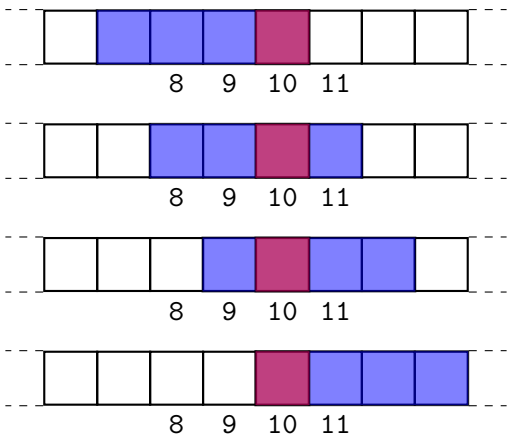
— where the variable begins

— How many bytes the variable occupies

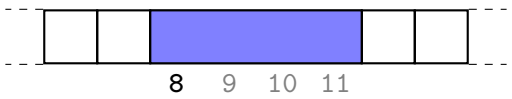
To demonstrate what is meant here, suppose we have an `int` variable stored at address 8, 9, 10 and 11, like this:



Now suppose we refer to the `int` stored at address `10`, we know that `int` values are 4 bytes large but how are we supposed to know where the *actual* value begins? We might be in any of these situation:



To solve this problem the terminology is adjusted so that the expression "`int` variable stored at address `8`" means that the variable *begins* at address `8`, so we uniquely identify a variable by its *address* (i.e. where in memory it begins) and its *size* (i.e. how many bytes it occupies). So we adjust our model to be similar to this diagram instead:



5.1 Storing addresses

Because of the existence of addresses there are generally *two* ways to uniquely identify variables: using their name *or* their address. But what data type is an *address*? Remember that an address itself is usually not enough, we also need to know what type of data is stored at said address. This is where the concept of a *pointer* comes in. A pointer is simply a data type which stores an address and keeps track of what type is stored there. It generally looks like this: `int*`, where the `*` indicates that this is a type that stores an address, and the `int` signifies which data type is stored at that location. You can create a pointer like this:

```
int* pointer;
```

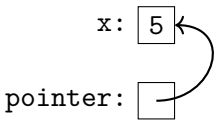
Remember that we store addresses in this variable, so the natural question should be: where do we get addresses from? How do we retrieve an address of a variable? There are many situations where an address might appear, but the most simple case is when we try to find the address of a specific variable. This is done with the `&` operator. This works like this:

```
1 int x { 5 };          // create a variable
2 int* pointer { &x }; // store a pointer to 'x'
```

But just storing an address seems pretty pointless. It would be quite unnecessary to store it unless we can actually access whatever is stored at the address. Luckily this is covered with the use of the `*` operator which is a *unary prefix operator*, meaning you put the `*` before the pointer (just like the `&` operator). See the following example:

```
1 int x { 5 };          // create a variable
2 int* pointer { &x }; // store a pointer to 'x'
3
4 // retrieve the value at the address stored in 'pointer'
5 // as an int and print it (meaning we'll print 5)
6 std::cout << *pointer << std::endl;
```

Programmers don't always think of pointers in terms of addresses, instead it is usually helpful to think of them as *indirections*. The term *indirection* in Computer Science refers to an abstraction where *something* refers to *something else*. In this instance it means that a pointer refers to another variable somewhere else. It is common to represent pointers as arrows between objects. Something like this:



This allows us to think of pointers as something that *refers* to another object, which is its primary use. This type of indirection can be achieved in other ways, for example by using *references* or *iterators* but most of these indirections are secretly pointers themselves.

We see through the use of a pointer we can refer to `x` in multiple ways: either by just referring to its given name `x`, *or* through its address which we've stored in `pointer`. But

so far we've only seen how to *read* the value of `x` through the pointer, but how do we *write*?

5.2 Assignment and indirection

Since pointers themselves are variables that store addresses this means that we can *change* what address is currently stored. We do this similarly to any other variable using the `=` operator. See the following example:

```
1 int x { 5 };
2 int y { 3 };
3 int* pointer { &x };
4
5 std::cout << *pointer << std::endl; // print 5
6 pointer = &y; // change the stored address
7 std::cout << *pointer << std::endl; // print 3
```

But what if we want to modify the object stored at the address? Well, then we once again have to use the so-called *dereference operator* (i.e. the `*` operator). The act of *dereferencing* a pointer means to retrieve the object stored at the other end of a pointer, so to modify the object we point to we first dereference the pointer and then do the modification. See the following:

```
1 int x { 5 };
2 int* pointer { &x };
3
4 std::cout << *pointer << std::endl; // print 5
5 *pointer = 3; // dereference and then assign
6 std::cout << *pointer << std::endl; // print 3
```

Note that dereferencing a pointer gives us *full* access to the underlying object, so we aren't limited to assignments. Anything you can do with an `int` (in this example) can be done with the dereferenced pointer. So:

```
1 int x { 5 };
2 int* pointer { &x };
3
4 *pointer = *pointer + 3;
5 ++(*pointer);
6 &(*pointer); // take the address of 'x'
```

The final expression above (line 6) demonstrates an interesting relationship between `&` and `*`, namely that they are each others inverses. The expression `&*pointer` gives you the address to `x`, while the expression `*&x` is equivalent to `x`. Meaning they *cancel* each other out.

`pointer` is an *indirection* so through it we still *indirectly* have access to the variable `x`, but it requires some extra steps (i.e. we need to dereference it).

5.3 Pointers are also variables

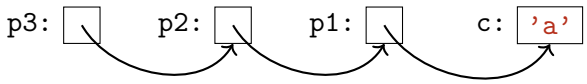
An interesting thing about pointers is that they themselves are variables, which means they also have an address. This means that you can have pointers *to other pointers*. Study the following example:

```
1 float f { 1.23f };
2 float* ptr1 { &f };
3 float** ptr2 { &ptr1 };
4
5 // all three of these print the same thing
6 std::cout <<   f    << " = "
7           <<  *ptr1 << " = "
8           << **ptr2 << std::endl;
```

Here we see that you can have pointers to anything, including other pointers by adding a `*` to the end of the type. Each `*` represents a layer of indirection. So if we have something like this:

```
1 char    c  { 'a' };
2 char*   p1 { &c };
3 char**  p2 { &p1 };
4 char*** p3 { &p2 };
```

Then the structure looks like this:



This diagram demonstrates the idea of layered indirections where we through `p3` have access to every other variable in the structure through the pointers. To retrieve them you just have to repeatedly apply the dereference operator, like this:

```
***p3 -> **(*p3) -> **p2 -> *(*p2) -> *p1 -> c
```

5.4 Pointing to nothing (`nullptr`)

One thing to note about pointers is that we can also have pointers that are "empty", meaning they point to no object what-so-ever. This is in fact the default-value for pointers, so if we initialize a pointer variable with empty braces then it will point to nothing.

However, a pointer variable must always store *some* address, which means that "pointing to nothing" must, in practice, still be represented as some type of address. This special "point to nothing" address is called `nullptr` and is *usually* (on most systems) represented with address `0x0`. A pointer that contains the `nullptr` address is called a *null pointer*.

To create a null pointer one can either default-initialize it or assign `nullptr` to it:

```
1 float* ptr1 {};
2 double* ptr2 {nullptr};
3
4 int x { 3 };
```

```
5  int* ptr3 { &x };
6
7  // make ptr3 a null pointer
8  ptr3 = nullptr;
```

One have to be very careful however when using null pointers, because it is *undefined behavior* to dereference the pointer (i.e. the expression `*ptr` is undefined behavior *if* `ptr` is `nullptr`). Usually, on most systems, this will result in a crash since modern operating systems does not allow read or write access to address `0x0`.

So if you need to dereference a pointer you must first make sure that it cannot be `nullptr` under any circumstance. Usually this is done by checking before dereferencing, *or* by making sure that there are no executions of the program where a pointer might end up as a null pointer.

5.5 Example

In this section we will study an example of *when* pointers (indirections) can be useful. This example will be slightly contrived just to keep things simple and to the point.

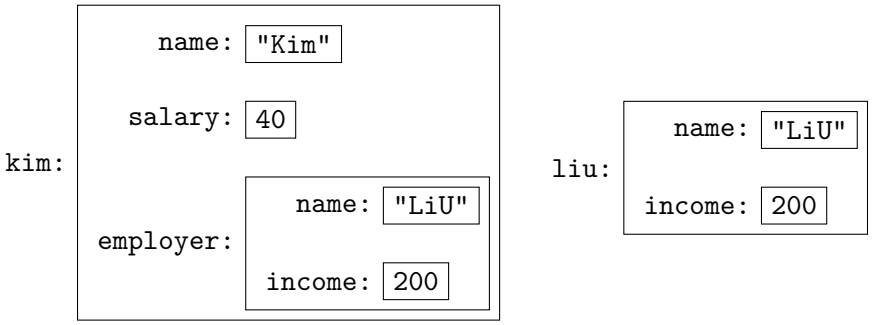
Suppose we are writing a simple database program where we store employees and organizations. Each employee store their name, their salary as well as their employer organization. Similarly, each organization store their name as well as their income. At first glance we might represent it like this:

```
1  struct Organization
2  {
3    std::string name;
4    int income;
5  };
6  struct Employee
7  {
8    std::string name;
9    int salary;
10   Organization employer;
11 };
```

But an interesting problem arises if we represent the employee like this. Consider this example:

```
1  int main()
2  {
3    Organization liu { "LiU", 200 };
4    Employee kim { "Kim", 40, liu };
5    cout << kim.employer.income << endl; // prints 200
6    liu.income = 250; // LiU got a grant so their income increased
7    cout << kim.employer.income << endl; // STILL prints 200, why?
8  }
```

Here we note that the employer organization stored in `kim` is **not** the same object as `liu`. This is because we are storing a *copy* of `liu` in `kim`, not `liu` itself. This would be even stranger if we had multiple employees at LiU, cause then each employee would have their own copy of `liu`. To aid in the understanding a diagram is provided:



So how do we solve it so that all employees of `liu` have direct access to the original `liu`? Well, if we store the *address* of the original `Organization` then we will always have access to it from inside an `Employee`. In particular the `Employee` object will see any potential changes that occurs in the `Organization` object, which is very useful for keeping our data synchronized over all `Employee` objects.

This would mean our structures would look like this instead:

```
1  struct Organization
2  {
3    std::string name;
4    int income;
5  };
6  struct Employee
7  {
8    std::string name;
9    int salary;
10   Organization* employer;
11 };
```

The main program from before must be modified now since an `Employee` no longer stores an `Organization` object directly, but instead stores a pointer, so now we initialize it with the *address* of our organization (i.e. `&liu`). We also need to dereference the `employer` pointer every time we want to access something inside it. Something like this:

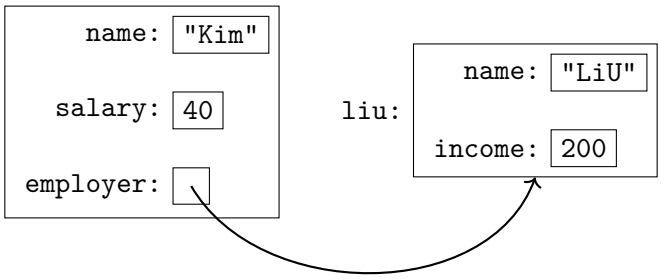
```
1  int main()
2  {
3    Organization liu { "LiU", 200 };
4    // note that we now pass the address of 'liu' to kim
5    Employee kim { "Kim", 40, &liu };
6    cout << (*kim.employer).income << endl; // prints 200
7    liu.income = 250;
8    cout << (*kim.employer).income << endl; // prints 250
9  }
```

The syntax `(*kim.employer).income` means that we take the pointer `employer` stored in `kim` and dereference it to get access to the `Organization` object it points to, then we access the field called `income` in said `Organization` object. This construction can however be simplified by writing: `kim.employer->income` where `->` now means that we take the pointer to the left (`kim.employer`) and access the `income` field at the other end of the pointer.

So the main program simplifies to:

```
1 int main()
2 {
3   Organization liu { "LiU", 200 };
4   // note that we now pass the address of 'liu' to kim
5   Employee kim { "Kim", 40, &liu };
6   cout << kim.employer->income << endl; // prints 200
7   liu.income = 250;
8   cout << kim.employer->income << endl; // prints 250
9 }
```

The structure above results in the following diagram, in particular note that there is only one `Organization` object now:



Note: We could also store the organization as a reference, but then an employee cannot change jobs because a reference can never change what it refers to.

6 Dynamic Memory

In the previous section we saw an example of using pointers as *indirection* to allow us to share the same data over multiple places in the code (in the example seen before we had several `Employee` objects that shared the same `Organization` objects).

But pointers has one other major use case. Note that an object can generally be uniquely identified in *two* different ways: using their assigned variable name, *or* by its address (and type). But names can only be assigned to variables that are declared *statically* (meaning during compile time), but sometimes we want to *dynamically* create objects (meaning while the program is running). In that cases we cannot create variable names, so then we can only access the created objects through pointers.

The typical example for when this becomes relevant is clearly demonstrated by studying so-called *linked structures*, for example linked lists.

6.1 Linked list

There are multiple strategies for storing sequences of data, a very common way is to use *arrays*: meaning we store the elements sequentially in memory. This means that each element is stored right next to the previous element. Arrays have many benefits, but we will not be discussing them here.

Another way to store sequences of data is to use a so called linked structure. Instead of enforcing that all elements are stored next to each other we can store them in a more fragmented manner where each element keeps track of where in memory to find the next element. So we *link* the elements together.

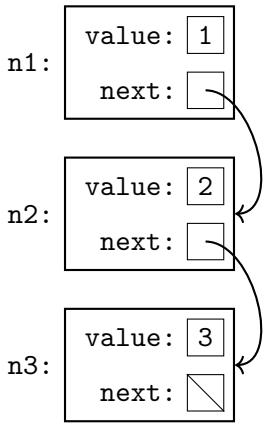
The most common linked structure is the (singly) linked list. Here each element is stored as a *node* which is a structure that looks like this:

```
1 struct Node
2 {
3   int value {};  // value currently stored
4   Node* next {}; // where to find the next node
5 };
```

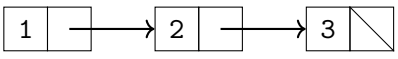
With this we can now create sequences of elements, for example like this:

```
1 // create all nodes
2 Node n1 { 1 };
3 Node n2 { 2 };
4 Node n3 { 3 };
5 // link the nodes together
6 n1.next = &n2;
7 n2.next = &n3;
```

This is, as we will see shortly, not the best way to construct linked lists but it works for our current intentions. Below is a diagram to help visualize what is happening:



Since we will be drawing these diagrams often it is common to simplify the visual representation of a linked structure to something like this:



Where the left box represents the `value` field, and the right box represent the `next` field.

Now usually when constructing larger components like lists and so on, we want to simply their usage. This is most commonly done by packaging the structure into a class with all the appropriate functions.

What we need to note about such a class is that it only have to store a pointer to the *first* node, because all subsequent nodes will be found by following the corresponding `next` pointers. This means that the class only need to store a pointer.

In this example we will implement a so-called stack using a linked list. Something like this:

```
1  class Stack
2  {
3  public:
4    void push(int value);
5    int pop();
6    int& top();
7    bool empty() const;
8  private:
9    Node* first {};
10 };
```

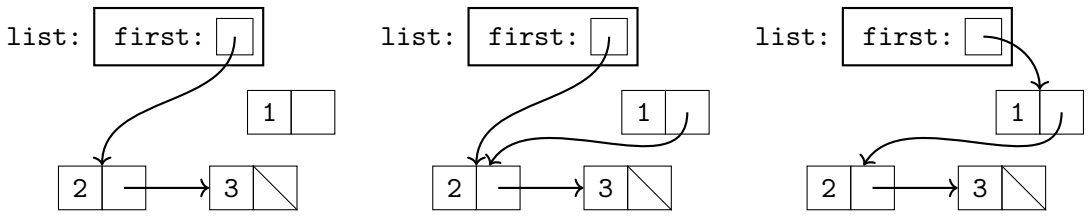
Where the `push()` function will add an integer to the *beginning* of the linked structure and `pop()` will remove the first value from the stack and return it. We also have an access function called `top()` which returns a reference to the first value. Finally we need to know whether the stack is empty or not, which we do with the `empty()` function.

In this example we will note that the stack is empty if `first` is `nullptr`.

We will start by looking at the implementation of the `void Stack::push(int value)` function. Logically we must do the following to implement it:

1. Create a node which has its `value` field set to the parameter.
2. Make the newly created nodes `next` field point to the start of the list.
3. Update the `first` pointer so that it points to the newly created node, thus making our new node the first in the list.

Here are diagrams to demonstrate the steps for a list already containing the values `2` and `3` where we push the value `1`:



(a) step 1 (b) step 2 (c) step 3

So how do we realize this with code? Well one might attempt something like this:

```
1 // NOTE: THIS CODE CONTAINS AN ERROR
2 void Stack::push(int value)
3 {
4   Node new_node { value };
5   new_node.next = first;
6   first = &new_node;
7 }
```

However, recall from section 4 that variables declared inside a function will be *destroyed* once the function returns, which means our newly created node cannot exist outside of the function scope. This will lead to strange memory errors since the node doesn't exist anymore and might have been overwritten with something else.

What we need to solve this problem is to have functionality which creates new values or objects without destroying them at the end of the function scope. Essentially, we want to be able to create objects that aren't destroyed automatically. This is what *dynamic memory* achieves.

6.2 Introduction to dynamic memory

In order to create objects that outlives the scope we must first establish *where* those object should end up. We cannot place them in the stack (section 4) since then we would "block" a variable slot, thus being unable to freely use the stack as intended. So we must place them elsewhere.

The operating system usually grants a program *more* memory than is required by the stack, so we can use that leftover memory to store our freestanding objects in a less structured way than the stack. This section of the memory is usually called *The Heap* (historically this was because this memory used to be structured as a heap data-structure, but that is usually not the case anymore). Another benefit of having a less structured section of the memory (compared to the stack) is that we can always request *more* memory from the operating system if it is needed and we can have "gaps" in the memory without any issues since the structure of the heap doesn't require the objects to be placed in sequence.

When putting objects in the heap we generally have to do two things:

1. *Allocate the memory*: this means finding a suitable position in the heap not already occupied by some other object where the object we want to create *fits* and reserving it (meaning no one else is now allowed to occupy that memory).

   For example: if I want to allocate an object that takes 16 bytes, then allocation means asking for an address inside the heap which has *at least* 16 consecutive bytes that are not occupied by any other object, we also reserve that memory so no other object can use those bytes.

2. *Construct the object*: Once we've received a suitable position in memory it is time to actually occupy that memory with an object which is initialized either by calling an appropriate constructor or by using aggregate initialization.

So here it is important to note the distinction between the two steps. The first step simply makes sure that we reserve a piece of the memory for our object, but after the completion

of said step we still don't have a valid object. The second step simply means initializing an object inside a portion of the memory.

These two steps can be performed independently of each other (we will see how to do that later on), but they will more commonly be done together using the operator called `new`, so that is where we will start our discussion.

6.3 `operator new`

By default the language has a builtin operator called `new` which allocates and constructs objects on the heap. It syntactically is used like this: `new Type{parameters}` by using *aggregate initialization* or, if you want to do *direct initialization*: `new Type(parameters)` (direct and aggregate initialization are discussed in seminar 1). Here `Type` can be *any* datatype, include user-defined types like classes, structs, unions and so on, while `parameters` represents parameters passed to either a constructor, aggregate initialization or a single parameter for *copy initialization*.

What is important to note here is that `new` will *allocate* the space for the object, then construct it and then finally return the *address* of the newly constructed object. So for example: `new int{3}` will allocate 4 bytes (if we assume `int` is 4 bytes), then initialize those bytes to be equivalent to the value `3` and then finally return a pointer of type `int*` to the `int` we just created.

Note: As previously discussed, these objects do not have names since they are created during execution of the program. To fully understand what this means, look at the following example:

```
1  int x { };
2  while (std::cin >> x)
3  {
4    std::string* ptr { };
5    if (x > 0)
6    {
7      // create a string of x number of '+' characters
8      ptr = new std::string(x, '+');
9    }
10 }
```

How many `std::string` objects are actually created? Well, it is impossible to say before we actually execute the program, because it depends on how many integers the user enters via `std::cin` and – more importantly – it also depends on what does integers where. Each execution of the program will likely lead to different results.

But how many *automatic* variables are created? Well, `x` is created once per execution of the program, and each iteration in the `while`-loop we will create and then destroy `ptr`.

So at any given point in the code the compiler *knows* how many automatic variable currently exist by just looking at the source code. However that is not true for dynamically constructed objects, since the number of objects in the heap directly depend on data that is unknowable by *just* reading the source code. This is why we cannot assign a name to

the dynamically constructed object, and instead refer to it through its address (specifically through a pointer).

6.4 When do dynamically allocated objects disappear?

The natural follow up question now is to ask about when dynamically objects disappear or get destroyed. The short answer is: when we tell them to.

Only objects created *statically* (i.e. automatic, global and static variables) are automatically destroyed in C++. *When* these variables are destroyed is well defined by the language. It is so well defined in fact that we can know – just by looking at the source code – at what point in the code they will be destroyed.

The same is not true for objects created *dynamically* (using for example `new`). These objects are *meant* to bypass the rules that statically created objects follow, they are *meant* to be used as objects that we *manually* control.

How we deal with dynamically memory differ however from language to language. As mentioned in section 1 some languages like Python or C# use a garbage collector. A garbage collector is a system which keeps track of all the allocated objects and regularly pauses the program to check if there are objects that are being unused (this explanation is *heavily* simplified). This system is independent from the compiler in the sense that when you *run* your program you will also automatically run the garbage collector.

As you might imagine, a garbage collector makes life easy for programmers because we don't have to worry about filling up our memory with unused objects since the garbage collector will regularly deal with them. But this doesn't come for free. The garbage collector *will* impact how fast your program can execute. The garbage collector will also make the behavior of your program *nondeterministic*, meaning we cannot predict by simply reading the program exactly what the computer will do at any given point in the execution because the garbage collector might briefly take control at any point.

Because of the performance loss and because C++ aims to be *deterministic* (meaning it will behave *exactly* the same each time you run the program with the same indata) it does *not* use a garbage collector.

Instead what is done in C++ is that we manually insert points in the code where objects are destroyed. I.e. when execution reach that point a particular object will be destroyed and that memory will be freed up to be used for something else. What is important to note is that this is somewhat similar to what the compiler does for automatic variables, since all automatic variables are destroyed at the end of the scope they were created in. But we must instead manually insert those points for our dynamically constructed objects.

Note that this is different from garbage collection because for automatic variables the compiler will insert their point of destruction automatically without ever having to run the program, while garbage collectors exclusively deal with the destruction of dynamically created objects *during* the execution of the program.

Exactly how this is done depends on *how* we dynamically constructed the object, but so far we've only seen how to dynamically create objects using `new`. Because of this we will begin by explaining and discussing the counter-part of `new` which is called `delete`.

6.5 `operator delete`

Recall that `new` will *allocate* and *construct* an object, and then return a *pointer* to said object. This object will exist until we *delete* it. The way we delete said object is to pass its *address* to the operator called `delete`.

Here is an example on what this looks like:

```
1  int x { };
2  while (std::cin >> x)
3  {
4    std::string* ptr { };
5    if (x > 0)
6    {
7      // construct object dynamically
8      ptr = new std::string(x, '+');
9    }
10
11   // ...do stuff...
12
13   if (x > 0)
14   {
15     // destroy the dynamically constructed
16     // object by explicitly passing its address
17     // to the delete operator.
18     delete ptr;
19   }
20 }
```

The `delete` operator will do *two* things:

1. *Destroy the object*: Do any cleanup work necessary for the object to safely be removed. This essentially means calling the *destructor* of the object.

   Note: the fundamental types does not have destructors, so for those particular objects this step is skipped.

2. *Deallocate the object*: Return the allocated memory to the pool of available memory. This operation can – depending on the implementation – be quite costly, but since we explicitly call `delete` we can control *when* we pay this cost.

Do note that these steps are the opposite of the two steps done by `new` and in the opposite order. So the full process is:

1. Allocate the memory (`new`)
2. Construct the object (`new`)
3. Use the object
4. Destroy the object (`delete`)
5. Deallocate the memory (`delete`)

6.6 Back to the stack

We will come back to discuss the mechanics of dynamic memory, but for now we have `new` and `delete` which is all we need to *correctly* implement the linked stack we began working on in section 6.1.

Remember that we tried to implement the `void Stack::push(int value)` using automatic variables which lead to issues since the automatic variable gets destroyed once we reach the end of the function. We can now solve this problem using `new`, like this:

```
1 void Stack::push(int value)
2 {
3   Node* new_node { new Node { value } };
4   new_node->next = first;
5   first = new_node;
6 }
```

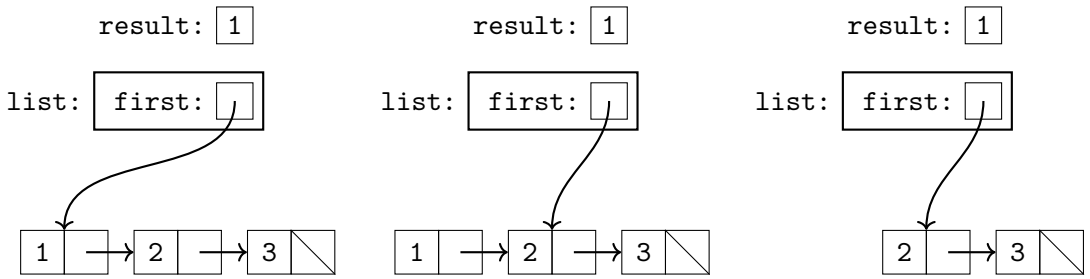
Using this implementation instead of the implementation in section 6.1, we see that the `Node` object will persist in memory after the function call (since we haven't manually destroyed it using `delete`).

With this we will now have a *fully* working `push()` function, meaning we can move on to its counter-part: `int Stack::pop()`.

The implementation must do the following:

1. Store the value of the current first node.
2. Set `first` to be the next node.
3. Destroy the previous first node.

Here are diagrams to show the steps in a list containing the values `1`, `2` and `3`:



(a) step 1 (b) step 2 (c) step 3

An attempt to implement this might look like this:

```
1 // NOTE: THIS CODE CONTAINS AN ERROR
2 int Stack::pop()
3 {
4   int result { first->value };
5   first = first->next;
6   return result;
7 }
```

However this doesn't delete the previous first node (i.e. it doesn't perform step 3). We need to call `delete` on the previous first node, otherwise it will exist in memory forever (or until the program ends), and since we overwrote the `first` pointer we have now lost the address to that node. This leads to a problem where we can no longer delete the node since we've lost where in memory it was. This is called a *memory leak*.

6.7 Memory leak

One of the major problems that gets introduced by using dynamic memory is the so-called *memory leak*. This occurs when we forget to delete a dynamically created object, meaning it occupies memory even when its long past its use. The most common cause of memory leaks is when losing track of an objects address, meaning there is no way we *could* delete that object.

Here are some suggestions for how to avoid memory leaks:

- Make sure that for each `new` call all execution paths lead to a `delete` call *eventually*.
- Delete objects as soon as possible after you are done with them.
- Before assigning anything to a pointer, think carefully whether you need to delete the object it pointed to previously.
- Use the destructor to delete all objects that were dynamically created during the lifetime of the class.
- Don't dynamically construct objects unless you absolutely have to. Remember that you can still get pointers to statically allocated objects using `operator&`.

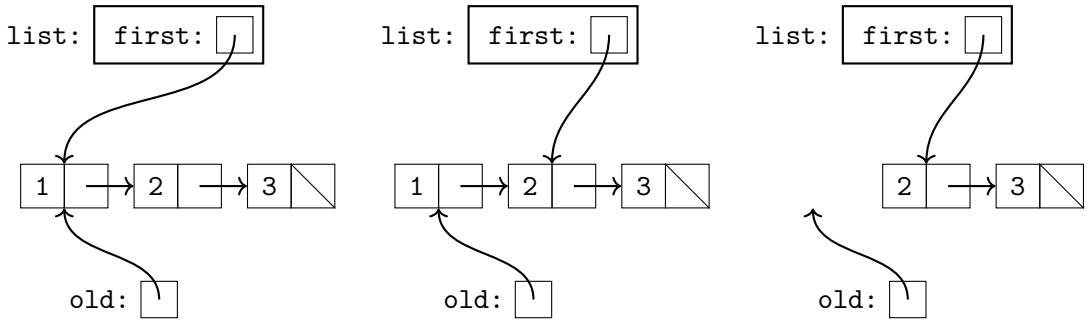
6.8 Fixing the `pop()` function

Now that we know more about memory leaks we can address the bad implementation of `Stack::pop()` given in section 6.8. The issue here is that we are assigning to `first` which means we lose whatever object it was pointing to previously. We fix that by taking a care around the assignment to `first`, like this:

```
1 int Stack::pop()
2 {
3   int result { first->value };
4   Node* old { first }; // remember old node
5   first = first->next;
6   delete old; // delete old node
7   return result;
8 }
```

Note that the order we do things here are important. In order to find what value `first` should point to we need the old node to exist so that we can find the *next* element. So once we've updated `first` then, and only then, can we delete the old first node. If we were to delete it first, then there are no guarantees that the `next` field still exists afterwards meaning we then would lose the *whole* list.

Below is a diagram showing what is happening:



(a) Remember old node (b) Point `first` to the next node (c) Delete old node

6.9 Completing the stack

To summarize what we have done in this chapter, below you can find the whole implementation of our linked stack:

```
1  class Stack
2  {
3  public:
4    void push(int value)
5    {
6      Node* new_node { new Node { value } };
7      new_node->next = first;
8      first = new_node;
9    }
10   int pop()
11   {
12     int result { first->value };
13     Node* old { first }; // remember old node
14     first = first->next;
15     delete old; // delete old node
16     return result;
17   }
18   int& top()
19   {
20     return first->value;
21   }
22   bool empty() const
23   {
24     return first == nullptr;
25   }
26 private:
27   Node* first {};
28 };
```

6.10 Dynamic Arrays

`new` is used to allocate singular objects *somewhere* in memory. However, it is quite common that we want to store several objects of the same type sequentially in memory, meaning we want to store them within consecutive addresses. This is something we generally cannot do with `new` since there are no guarantees that consecutive allocations will have consecutive addresses.

This concept of storing multiple objects of the same type in sequence is called an *array*. During the seminars we will see how to use *static* arrays (i.e. arrays that are allocated during compile time). In this document we will however see how to create *dynamic arrays* (i.e. arrays that are created during the execution of our program).

Dynamic arrays are allocated and constructed using the special operator called `new[]`. It is used like this:

```
int* array { new int[3] { 1, 2, 3 } };
```

The operation `new Type[N]` will allocate `N` objects of type `Type` in sequence in memory. The constructed objects are *guaranteed* to reside next to each other. It will appropriately initialize each object in order, either by using aggregate initialization of by calling appropriate constructors.

These dynamic arrays are then subsequently deallocated and destroyed using the `delete[]` operator. For example:

```
delete[] array;
```

Which is a different operator from `delete`. However the only difference is that `delete[]` deallocates arrays rather than singular objects. Note that `delete[]` can only delete arrays that were allocated with `new[]`.

6.11 `malloc()` and `free()`

As previously mentioned the `new` and `delete` operators *allocate* and *deallocate* memory as well as construct and destroy the objects residing in that memory. This is different from how it works in for example the programming language C.

In C there is no complex logic that gets executed when constructing or destroying objects, so there it is only necessary to allocate and deallocate memory, there is no need for construction and destruction. Sometimes this is true in C++ as well: we might need to allocate memory but we do not want to construct any objects there yet. Then we cannot use `new` and `delete`.

Since C++ was originally built on top of C, there are quite a lot of things that are shared between the languages. One such instance is that the way C deals with memory is also available in C++, which are through the functions `std::malloc` (called `malloc` in C) and `std::free` (called `free` in C).

`std::malloc()` takes the number of *bytes* you wish to allocate and returns the address of the first byte allocated. So if we for example want to allocate space for 4 `int` objects we could write: `std::malloc(4 * sizeof(int))`.

Note: `std::malloc()` does not set the bytes to anything in particular, whatever happens to reside in the memory is what you get. It is up to you to actually set those bytes.

`std::free()` takes an address to the beginning of a block of memory allocated using `std::malloc()` and deallocates it.

Note: `std::free()` will crash your program if you pass it anything that was *not* allocated using `std::malloc()` so be careful. Note that it doesn't touch the data itself, it only marks that memory as free to use for something else.

It is exceedingly rare that you actually *need* to use `std::malloc` and `std::free` in C++, the vast majority of the time `new` and `delete` are sufficient. In this course for example, we will *never* use `std::malloc` or `std::free`.