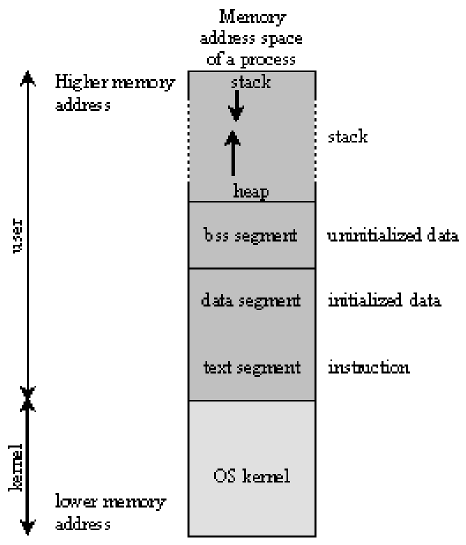

Figure 1: Process memory

0.1 C++ and memory management

In this lesson, you will learn how C++ manages memory, and you will gain understanding of how memory is managed inside your computer. Understanding these concepts is very important for programmers for several reasons:

- Allows you to optimize your program, avoiding to allocate useless memory.
- Understand what goes under the hood in dynamic programming languages.
- Can avoid security issues.
- Learn our first data structure!

0.1.1 Process memory

Every process inside a computer has a certain amount of memory, which is granted to the process by the operative system. Some of that memory will be reserved to keep some information about the process. That memory cannot be accessed, and is reserved to the operative system. Then, we have that is called the **stack**. Look at Figure 1 to get an idea of how process memory looks like:

Stack

The stack is used in order to keep track of the location where we are inside our process. Imagine that you have a certain number of tasks that you need to do. For every task, we make a post-it, with some information about the task. A task may require you to have to complete a different task before you may be able to solve the previous one, and for every task you need to store some information, relative to the task. How are we going to keep track of all the post-its that we will write?
An excellent way to manage this information is to use a pile. For every post-it that we make, we will push it on top our pile. If the task will require some other task to be completed, then we will begin working on that task, and push the relative post-it on top of our pile. Once a task is completed, then we remove the post-it from the top of our pile, and resume working on the task with the previous post it. When, the pile will be empty again, then our program will be over!
In computers programs, things work in the same way. Instead of tasks we have functions, and our pile is what we refer to as a stack, an abstract data structure. The information that we need in our process are variables, that we need to keep track of in order to complete our tasks. The post-its are the chunks of memory that we use to store that information inside our process. Lets take a look at an example:

```c
1  int task ( int sum )
2  {
3    int total = 0;
4    for ( int i =0; i < sum; i++ )
5      total +=i;
6    return total;
7  }
8
9  int main ( int argc, const char * argv[] )
10 { //  This is the entry point of our program.
11   int sum = 5;
12   int total = task( sum );
13   total     += task( sum ); // += will add to the existing value
14   return 0;
15 }
```

Now, lets take a look at what happens inside our stack when we execute the program. We start with main inside the stack. Inside the space for main, there will be enough space for the variables *sum*, and *total*. On line *13*, the variable sum will take the value 5. Then, the function *task* will be called.
New space will be asked for on the stack, so that we can fit the variables that we have inside *task*. The value of *sum* will be copied over, and the function *task* will have its

own copy. Likewise, *task* will have its own version of *total*. These are **local variables**, so they only exist in the scope of the function. You cannot access the local variables of *main* from the function *task*. Once the first call to *task* will be completed, we will copy the value of *total* to the main function, and the space on the stack that was reserved to *task* will be erased. Just like in our analogy we used to throw away post-its once they were completed. This is how the memory stack works!

Heap

There is one more complication to how memory management works. Imagine, we are in the middle of one of our functions/tasks. All of a student, we discover that we need more memory than we thought before. Where do we get the memory from? We cannot change the size of memory that we have on the stack, yet we need more memory. This is where dynamic memory comes in. If we do not know the amount of memory that we are going to use for a certain task, then we are going to use something called the **heap**. The heap is another memory segment in the process memory. It is used to allocate dynamic memory. You can allocate and free memory on the heap as you wish, without many limitations. In order to create memory on the heap in C++, you have to use the keyword new. Before we can look at an example, we have to look at something called pointers.

0.1.2 Pointers and reference

In C++, every variable has two attributes:

1. Type: E.x. int, bool.
2. The location of the variable in memory, which is a hexadecimal address. Eg. $0x301ca39d$.

When we use a variable, C++ goes into memory and looks at the value of that variable. Whenever we refer to a variable, C++ automatically lookup the value that it holds in memory. However, you can also lookup the address by using the **address** operator, **&**. Lets take a look at an example:

```
int look  = 5;
cout <<  look << endl; // will print 5
cout << &look << endl; // will print some address location.
```

Now, if you have an address, you can either store it, using another variable, or look up the memory inside the value. A variable that stores an address in memory is called a

pointer. You can lookup values inside pointers by using the **star operator**. Using the star operator to lookup the value of a memory address is called **dereferencing** a pointer. Lets continue looking at the previous example:

```
int * pointerToLook = &look;
cout << * pointerToLook << endl; // will print 5
cout <<   pointerToLook << endl; // will print some address. Equivalent to & look.
```

So, you can create a pointer variable using the star operator before the name of the variable, and then you can lookup the value inside memory using the star operator. Note that the two operatons, are reverse to each other. In fact, if you take the address and then dereference, it will be like doing nothing.

```
cout << *(&look ) == look << endl; // will print true
```

You have to be careful whenever you use the star operator. If you follow an address that doesn't exist in memory, then you will get a segmentation fault. That is, the computer will complain that the memory address that you have asked does not exist, therefore it will terminate the program. You have to be careful not to use uninitialized pointers, or pointer that point to NULL. Here is an example of segfaults:

```
int * a;
int * b = NULL;
* a = 3; // segfault
int c = * b; // seg fault
```

You can use pointers in two different ways. You can either use them to point at existing memory, or make them point at new memory you create for them. Both these cases can be very useful. Lets look at some examples:

```
int a = 5;
int * aReference = & a;
cout << a << endl; // prints 5
cout << * aReference << endl; // prints 5
* aReference = 8;
cout << a << endl; // prints 8
cout << * aReference << endl; // prints 8
a = 0;
```

```cpp
cout << a << endl; // prints 0
cout << * aReference << endl; // prints 0
int c = 39;
aReference = & c;
cout << a << endl; // prints 0
cout << c << endl; // prints 39
cout << * aReference << endl; // prints 39
```

Using pointers this way is very useful in case you want to keep allocating new memory for values. Say for example that you have a very large object, and you want to pass it inside a function. Copying the object would be very expensive. With pointers instead, you can simply pass in the address, and then you are done. There are three ways to pass in an object. By value, by reference and by pointer.
By value, it means that you copy over the value of the object you are passing in. This works very well with small object, such as primitive types. The object that is passed in will not change if you change it inside the function, because it is a distinct copy of the value that you have passed in.
Passing in values by reference and by pointers instead will change the value of the object that you pass inside. Both these two methods do not copy the object, but simply pass inside the function the address of the memory object you are passing inside the function. The two methods are fundamentally the same, they are only different in their syntax. Sometimes is more convenient to use reference, other pointer.

```cpp
int byValue ( int temp )
{
  temp += 5;
  return temp - 8;
}

int byReference ( int& temp )
{
  temp += 5;
  return temp - 8;
}

int byPointer ( int * temp )
{
  * temp += 5;
  return * temp -8;
}

int main ( int argc, const char * argv[] )
```

```
{
  int total = 5;
  cout << byValue     ( total )   << endl; // print 2
  cout << total                   << endl; // total not changed
  cout << byReference ( total )   << endl; // print 2
  cout << total                   << endl; // total changes to 10
  cout << byPointer   ( & total ) << endl; // print 7
  cout << total                   << endl; // total changes to 15
}
```

Now that we have a good understanding of pointers, lets see how to allocate new memory on a newly declared pointer.

0.1.3 Allocating memory in C++