[1] difference between struct and union
source:
http://www.thecrazyprogrammer.com/2015/03/difference-between-structure-and-union.html

Struct
The memory a struct var occupies is the sum of memory that all its members occupy;

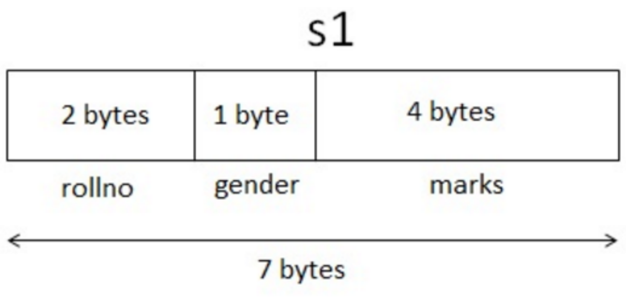
After the assginments, all members are stored;

```
struct student
{
	int rollno;
	char gender;
	float marks;
}s1;

struct student* p = &s1;

p-> rollno = 1;
p-> gender = ‘F’;
p-> marks = 46;

sizeof(student) = sizeof(int)+sizeof(char)+sizeof(float)
```



Memory Allocation in Structure

Union
The memory a union var occupies is the memory that its longest member occupies;

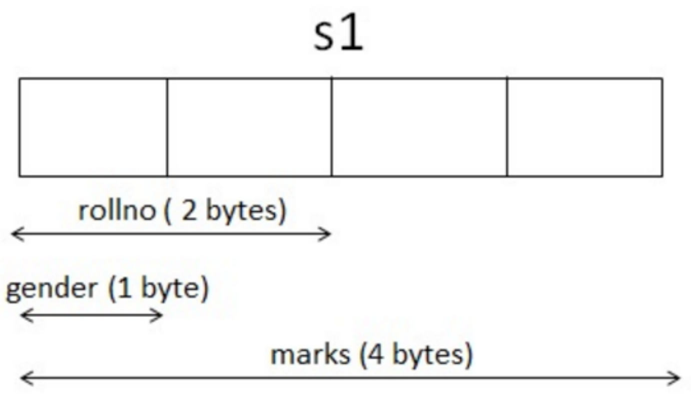
After assginments, only the last member is stored. All previous members are overridden;

```
union student
{
	int rollno;
	char gender;
	float marks;
}s1;


struct student* p = &s1;

p-> rollno = 2;      (the value of rollno will be overridden)
p-> gender = 'M';  (the value of gender will be overridden)
p-> marks = 20;   (the value of marks is stored)

sizeof(student) = max{ sizeof(int), sizeof(char), sizeof(float) }
```



Memory Allocation in Union

[2] difference between stack and heap
source:
http://www.programmerinterview.com/index.php/data-structures/difference-between-stack-and-heap/

Where are the stack and heap stored?
They are both stored in the computer's RAM (Random Access Memory). For a refresher on RAM and virtual memory, read this article: How Virtual Memory Works

How do threads interact with the stack and the heap? How do the stack and heap work in multithreading?

In a multi-threaded application, each thread will have its own stack. But, all the different threads will share the heap. Because the different threads share the heap in a multi-threaded application, this also means that there has to be some coordination between the threads so that they don't try to access and manipulate the same piece(s) of memory in the heap at the same time.

Can an object be stored on the stack instead of the heap?

Yes, an object can be stored on the stack. If you create an object inside a function without using the "new" operator then this will create and store the object on the stack, and not on the heap. Suppose we have a C++ class called Member, for which we want to create an object. We also have a function called somefunction( ). Here is what the code would look like:

Code to create an object on the stack:

```
void somefunction( )
{
/* create an object "m" of class Member
    this will be put on the stack since the
    "new" keyword is not used, and we are
   creating the object inside a function
*/

  Member m;

} //the object "m" is destroyed once the function ends
```

So, the object "m" is destroyed once the function has run to completion – or, in other words, when it "goes out of scope". The memory being used for the object "m" on the stack will be removed once the function is done running.

If we want to create an object on the heap inside a function, then this is what the code would look like:

Code to create an object on the heap:

```
void somefunction( )
{
/* create an object "m" of class Member
```

```
    this will be put on the heap since the
    "new" keyword is used, and we are
   creating the object inside a function
*/

  Member* m = new Member( ) ;

  /* the object "m" must be deleted
      otherwise a memory leak occurs
  */

  delete m;
}
```

In the code above, you can see that the “m” object is created inside a function using the “new” keyword. This means that “m” will be created on the heap. But, since “m” is created using the “new” keyword, that also means that we must delete the “m” object on our own as well – otherwise we will end up with a memory leak.

How long does memory on the stack last versus memory on the heap?

Once a function call runs to completion, any data on the stack created specifically for that function call will automatically be deleted. Any data on the heap will remain there until it’s manually deleted by the programmer.

Can the stack grow in size? Can the heap grow in size?

The stack is set to a fixed size, and can not grow past it’s fixed size (although some languages have extensions that do allow this). So, if there is **not** enough room on the stack to handle the memory being assigned to it, a ***stack overflow***occurs. This often happens when a lot of nested functions are being called, or if there is an infinite recursive call.
If the current size of the heap is too small to accommodate new memory, then more memory can be added to the heap by the operating system. This is one of the big differences between the heap and the stack.

How are the stack and heap implemented?

The implementation really depends on the language, compiler, and run-time – the ***small*** details of the implementation for a stack and a heap will always be different depending on what language and compiler are being used. But, in the big picture, the stacks and heaps in one language are used to accomplish the same things as stacks and heaps in another language.

Which is faster – the stack or the heap? And why?

The stack is much faster than the heap. This is because of the way that memory is allocated on the stack. Allocating memory on the stack is as simple as moving the stack pointer up.

How is memory deallocated on the stack and heap?

Data on the stack is ***automatically*** deallocated when variables go out of scope. However, in languages like C and C++, data stored on the heap has to be deleted**manually** by the programmer using one of the built in keywords like **free, delete, or delete[ ]. Other languages like Java and .NET use garbage collection to automatically delete memory from the heap, without the programmer having to do anything.**.

What can go wrong with the stack and the heap?

If the stack runs out of memory, then this is called a *stack overflow* – and could cause the program to crash. The heap could have the problem of *fragmentation*, which occurs when the available memory on the heap is being stored as noncontiguous (or disconnected) blocks – because *used* blocks of memory are in between the *unused* memory blocks. When excessive fragmentation occurs, allocating new memory may be impossible because of the fact that even though there is enough memory for the desired allocation, there may not be enough memory in one big block for the desired amount of memory.

Which one should I use – the stack or the heap?

For people new to programming, it’s probably a good idea to use the stack since it’s easier.
Because the stack is small, you would want to use it when you know exactly how much memory you will need for your data, or if you know the size of your data is very small. It’s better to use the heap when you know that you will need a lot of memory for your data, or you just are not sure how much memory you will need (like with a dynamic array)