Memory Allocation for Structure

If we create an object of some structure, then the compiler allocates contiguous memory for the data members of the structure. The size of allocated memory is at least the sum of sizes of all data members. The compiler can use padding and in that case there will be unused space created between two data members. The padding is done for the alignment of data members which makes the access to the member faster. However you can control the padding behavior and can stop the compiler to generate extra spaces.
The data members of a structure are accessed with the help of the base address of the structure and the offset of the data member in the structure object.

Lets see this in an example.
C code:

```
struct data_struct
{
    int a;
    int b;
};
struct data_struct global_data;
int main()
{
    struct data_struct local_data;
    global_data.a = 10;
    global_data.b = 15;
    local_data.a = 25;
    local_data.b = 20;
    return 0;
}
```

Generated assembly code:

```
    .comm   global_data,8,4
    .text
.globl main
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $10, global_data
    movl    $15, global_data+4
    movl    $25, -8(%ebp)
    movl    $20, -4(%ebp)
    movl    $0, %eax
    leave
    ret
```

Offset of data members of the structure

```
a ==> 0
b ==> 4
```

The members of global_data are accessed as:

```
global_data.a ==> global_data+0 or simply global_data
global_data.b ==> global_data+4
```

Similarly the local_data member will be accessed as

```
local_data.a ==> -8(%ebp)
local_data.b ==> -4(%ebp)
```

The local_data is allocated on the stack on the memory range -8(%ebp) to (%ebp)
Goto here (memorymanagement.html) for details of local variables allocation.

Structure and padding

The compiler often allocates some empty space between two members of a structure to make accessing each member faster. This is called padding. The size of alignment is mostly dependent on a processor architecture.

Lets take and example

```
struct data_struct
{
    char a;
    int b;
};
```

If we get the size of the structure using sizeof operator on i386, it will come to 8. But the sizeof(char) is one and sizeof(int) is 4 so total of 5 bytes are required but the compiler allocated 8 bytes. Actually it allocated 4 bytes for the char member too.

Now declare the structure as:

```
struct data_struct
{
        char a;
        int b;
} __attribute__((packed));
```

Then sizeof(struct data_struct) on i386 will come to 5. The __attribute__((packed)) has forced the compiler not to do padding and waste space. This attribute can be also applied on individual data members of the struct.

For example:-

```
struct data_struct
{
    char a;
    int b __attribute__((packed));
};
```

This declaration says do not do padding between data member a and b. The __attribute__ can be also used to increase the alignment boundary. Look at the compiler documentation for details on this.

Function returning structure

We have seen previously that the function returns through eax register for basic data types. But structure can not be returned by the eax register because the size of the eax register is only 4 bytes but the structure can be of any size. To return structure, compilers can use different strategies.

Let's see how gcc works with an example:
The C code:

```
struct data_struct
{
    int a;
    int b;
} ;
struct data_struct fun()
{
    struct data_struct data;
    data.a = 20;
    data.b = 25;
    return data;
}
int main()
{
    struct data_struct data = fun();
    return 0;
}
```

Generated assembly code:

```
    .text
.globl fun
fun:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    8(%ebp), %ecx
    movl    $20, -8(%ebp)
    movl    $25, -4(%ebp)
    movl    -8(%ebp), %eax
    movl    -4(%ebp), %edx
    movl    %eax, (%ecx)
    movl    %edx, 4(%ecx)
    movl    %ecx, %eax
    leave
    ret       $4
.globl main
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $20, %esp
    leal    -8(%ebp), %eax
    movl    %eax, (%esp)
    call    fun
    subl    $4, %esp
    movl    $0, %eax
    leave
    ret
```

We can re-write the equivalent C code of the generated assembly code. This code will look like:-

```
void fun(struct data_struct *ptr)
{
    struct data_struct data;
    data.a = 20;
    data.b = 25;
    ptr->a = data.a;
    ptr->b = data.b;
}
int main()
{
    struct data_struct data;
    fun(&data);
    return 0;
}
```

The compiler has passed a pointer to the function of the data type which will be returned. Here are two things to notice

- The function which was returning some struct, was made to void type. It will not return via eax register.
- The function got an extra argument of the type pointer to the structure which it was returning in the original code.

‹ Returning Value From Function (/cin/return.html)
up (/cin/cin.html)
Mixing C and Assembly › (/cin/mixing.html)