7.9. structs in Assembly

A struct is another way to create a collection of data types in C. Unlike arrays, structs enable different data types to be grouped together. C stores a `struct` like a single-dimension array, where the data elements (fields) are stored contiguously.

Let's revisit `struct studentT` from Chapter 1:

```c
struct studentT {
    char name[64];
    int  age;
    int  grad_yr;
    float gpa;
};

struct studentT student;
```

Figure 1 shows how `student` is laid out in memory. Each x_i denotes the address of a particular field.

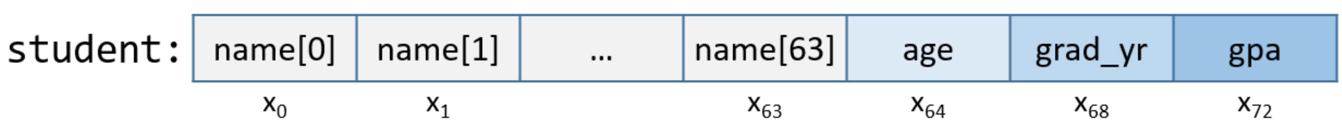

Figure 1. The memory layout of a struct studentT

The fields are stored contiguously next to one another in memory in the order in which they are declared. In Figure 1, the `age` field is allocated at the memory location directly after the `name` field (at byte offset x_{64}) and is followed by the `grad_yr` (byte offset x_{68}) and `gpa` (byte offset x_{72}) fields. This organization enables memory-efficient access to the fields.

To understand how the compiler generates assembly code to work with a `struct`, consider the function `initStudent`:

```c
void initStudent(struct studentT *s, char *nm, int ag, int gr, float g) {
    strncpy(s->name, nm, 64);
    s->grad_yr = gr;
    s->age = ag;
    s->gpa = g;
}
```

The `initStudent` function uses the base address of a `struct studentT` as its first parameter, and the desired values for each field as its remaining parameters. The following listing depicts this function in assembly:

```
Dump of assembler code for function initStudent:
0x4006aa <+0>:  push  %rbp                      #save rbp
0x4006ab <+1>:  mov   %rsp,%rbp                 #update rbp (new stack frame)
0x4006ae <+4>:  sub   $0x20,%rsp                #add 32 bytes to stack frame
0x4006b2 <+8>:  mov   %rdi,-0x8(%rbp)           #copy 1st param to %rbp-0x8
(s)
0x4006b6 <+12>: mov   %rsi,-0x10(%rbp)          #copy 2nd param to %rpb-0x10
(nm)
0x4006ba <+16>: mov   %edx,-0x14(%rbp)          #copy 3rd param to %rbp-0x14
(ag)
0x4006bd <+19>: mov   %ecx,-0x18(%rbp)          #copy 4th param to %rbp-0x18
(gr)
0x4006c0 <+22>: movss %xmm0,-0x1c(%rbp)         #copy 5th param to %rbp-0x1c
(g)
0x4006c5 <+27>: mov   -0x8(%rbp),%rax           #copy s to %rax
0x4006c9 <+31>: mov   -0x10(%rbp),%rcx          #copy nm to %rcx
0x4006cd <+35>: mov   $0x40,%edx                #copy 0x40 (or 64) to %edx
0x4006d2 <+40>: mov   %rcx,%rsi                 #copy nm to %rsi
0x4006d5 <+43>: mov   %rax,%rdi                 #copy s to %rdi
0x4006d8 <+46>: callq 0x400460 <strncpy@plt> #call strcnpy(s->name, nm,
64)
0x4006dd <+51>: mov   -0x8(%rbp),%rax           #copy s to %rax
0x4006e1 <+55>: mov   -0x18(%rbp),%edx          #copy gr to %edx
0x4006e4 <+58>: mov   %edx,0x44(%rax)           #copy gr to %rax+0x44 (s-
>grad_yr)
0x4006e7 <+61>: mov   -0x8(%rbp),%rax           #copy s to %rax
0x4006eb <+65>: mov   -0x14(%rbp),%edx          #copy ag to %edx
0x4006ee <+68>: mov   %edx,0x40(%rax)           #copy ag to %rax+0x40 (s-
>age)
0x4006f1 <+71>: mov   -0x8(%rbp),%rax           #copy s to %rax
0x4006f5 <+75>: movss -0x1c(%rbp),%xmm0         #copy g to %xmm0
0x4006fa <+80>: movss %xmm0,0x48(%rax)          #copy g to %rax+0x48
0x400700 <+86>: leaveq                          #prepare stack to exit
function
0x400701 <+87>: retq                            #return (void func, %rax
ignored)
```

Being mindful of the byte offsets of each field is key to understanding this code. Here are a few things to keep in mind.

- The `strncpy` call takes the base address of the `name` field of `s`, the address of array `nm`, and a length specifier as its three arguments. Recall that because `name` is the first field in the `struct studentT`, the address of `s` is synonymous with the address of `s->name`.

```
0x4006b2 <+8>:  mov   %rdi,-0x8(%rbp)        #copy 1st param to %rbp-0x8
(s)
0x4006b6 <+12>: mov   %rsi,-0x10(%rbp)       #copy 2nd param to %rpb-0x10
(nm)
0x4006ba <+16>: mov   %edx,-0x14(%rbp)       #copy 3rd param to %rbp-0x14
(ag)
0x4006bd <+19>: mov   %ecx,-0x18(%rbp)       #copy 4th param to %rbp-0x18
(gr)
0x4006c0 <+22>: movss %xmm0,-0x1c(%rbp)      #copy 5th param to %rbp-0x1c
(g)
0x4006c5 <+27>: mov   -0x8(%rbp),%rax        #copy s to %rax
0x4006c9 <+31>: mov   -0x10(%rbp),%rcx       #copy nm to %rcx
0x4006cd <+35>: mov   $0x40,%edx             #copy 0x40 (or 64) to %edx
0x4006d2 <+40>: mov   %rcx,%rsi              #copy nm to %rsi
0x4006d5 <+43>: mov   %rax,%rdi              #copy s to %rdi
0x4006d8 <+46>: callq 0x400460 <strncpy@plt> #call strcnpy(s->name, nm,
64)
```

- This code snippet contains the previously undiscussed register (`%xmm0`) and instruction (`movss`). The `%xmm0` register is an example of a register reserved for floating-point values. The `movss` instruction indicates that the data being moved onto the call stack is of type single-precision floating point.

- The next part of the code (instructions `<initStudent+51>` thru `<initStudent+58>`) places the value of the `gr` parameter at an offset of 0x44 (or 68) from the start of `s`. Revisiting the memory layout in Figure 1 shows that this address corresponds to `s->grad_yr`:

```
0x4006dd <+51>: mov   -0x8(%rbp),%rax        #copy s to %rax
0x4006e1 <+55>: mov   -0x18(%rbp),%edx       #copy gr to %edx
0x4006e4 <+58>: mov   %edx,0x44(%rax)        #copy gr to %rax+0x44 (s-
>grad_yr)
```

- The next section of code (instructions `<initStudent+61>` thru `<initStudent+68>`) copies the `ag` parameter to the `s->age` field of the `struct`, which is located at an offset of 0x40 (or 64) bytes from the address of `s`:

```
0x4006e7 <+61>: mov   -0x8(%rbp),%rax        #copy s to %rax
0x4006eb <+65>: mov   -0x14(%rbp),%edx       #copy ag to %edx
0x4006ee <+68>: mov   %edx,0x40(%rax)        #copy ag to %rax+0x40 (s-
>age)
```

- Lastly, the `g` parameter value is copied to the `s→gpa` field (byte offset 72 or 0x48) of the `struct`. Notice the use of the `%xmm0` register since the data contained at location `%rbp-0x1c` is single-precision floating point:

```
0x4006f1 <+71>: mov   -0x8(%rbp),%rax        #copy s to %rax
0x4006f5 <+75>: movss -0x1c(%rbp),%xmm0      #copy g to %xmm0
0x4006fa <+80>: movss %xmm0,0x48(%rax)       #copy g to %rax+0x48
```

7.9.1. Data Alignment and structs

Consider the following modified declaration of `struct studentT`:

```c
struct studentTM {
    char name[63]; //updated to 63 instead of 64
    int  age;
    int  grad_yr;
    float gpa;
};

struct studentTM student2;
```

The size of the `name` field is modified to be 63 bytes, instead of the original 64. Consider how this affects the way the `struct` is laid out in memory. It may be tempting to visualize it as in Figure 2.

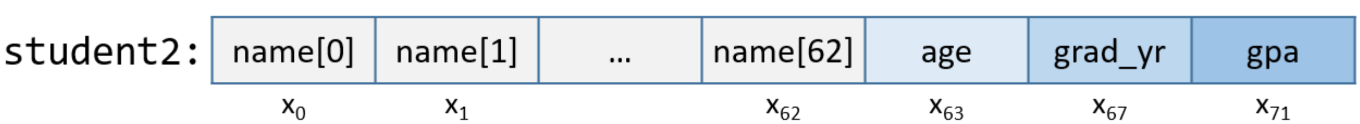

Figure 2. An incorrect memory layout for the updated struct studentTM. Note that the struct's "name" field is reduced from 64 to 63 bytes.

In this depiction, the `age` field occurs in the byte immediately following the `name` field. But this is incorrect. Figure 3 depicts the actual layout in memory.

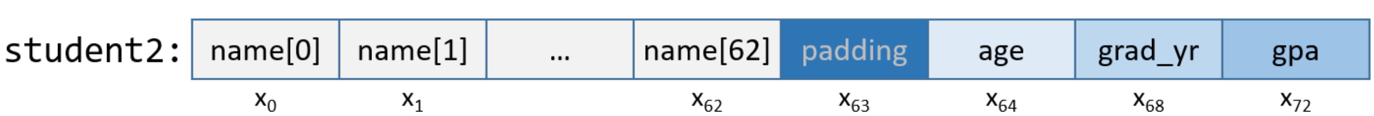

Figure 3. The correct memory layout for the updated struct studentTM. Byte x_{63} is added by the compiler to satisfy memory alignment constraints, but it doesn't correspond to any of the fields.

x64's alignment policy requires that two-byte data types (i.e., `short`) reside at a two-byte-aligned address, four-byte data types (i.e., `int`, `float`, and `unsigned`) reside at four-byte-aligned addresses, whereas larger data types (`long`, `double`, and pointer data) reside at eight-byte-aligned addresses. For a `struct`, the compiler adds empty bytes as **padding** between fields to ensure that each field satisfies its alignment requirements. For example, in the `struct` declared in Figure 3, the compiler adds a byte of padding at byte x_{63} to ensure that the `age` field starts at an address that is at a multiple of four. Values aligned properly in memory can be read or written in a single operation, enabling greater efficiency.

Consider what happens when a `struct` is defined as follows:

```c
struct studentTM {
    int  age;
    int  grad_yr;
    float gpa;
    char name[63];
};

struct studentTM student3;
```

Moving the `name` array to the end ensures that `age`, `grad_yr`, and `gpa` are four-byte aligned. Most compilers will remove the filler byte at the end of the `struct`. However, if the `struct` is ever used in the context of an array (e.g., `struct studentTM courseSection[20];`) the compiler will once again add the filler byte as padding between each `struct` in the array to ensure that alignment requirements are properly met.

Contents

7.9. structs in Assembly

7.9.1. Data Alignment and structs