CS 61 Scribe Notes - 09/17/2013

Jonah Kallenbach, Keenan Monks, Matt Rauen

1 Overview

1. Data representation for complex types: arrays, structures, unions
2. Storage recap
3. Memory errors

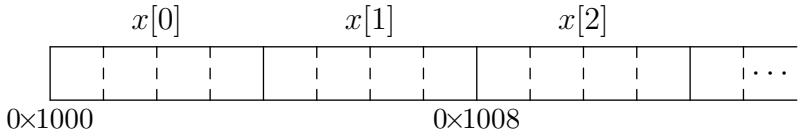
2 Arrays

The C abstract machine treats a difference in addresses as a difference in indices. So in an `int array`, although 8 bytes separate `&x[0]` and `&x[2]`, we would get that `&x[2] - &x[0]`=2. It first subtracts the indices, and then divides by `sizeof(int)`. If we treat the pointers as `char`*, it will instead divide by `sizeof(char)`= 1, and we get the actual difference in addresses. For example, in the following program, we would expect an output of 2.

```
#include <inttypes.h>
#include "hexdump.h"

int main() {
    int x[5] = {0,1,2,3,100};
    hexdump(stdout, &x, sizeof(x));
printf(%d\n,(int)(&x[2]-&x[0]))
}
```

Here is a depiction of the layout in memory, with some sample addresses.



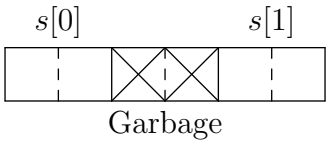
Also, it is impossible for two live objects to have the same address. One cool property of C is that we are permitted to compare the addresses that pointers point to. Comparing pointers can give useful information. For example, if we know that a pointer points after the beginning and before the end of an array, we can deduce that it points within the array. The way C computes addresses is `&a[i]` = `&a[0]` + `i`*`sizeof(a[i])`. This works because every element of an array must have the same type and size. Therefore, you cannot have an array of arbitrary arrays, and if you cannot take the size of the elements of the array, you get an incomplete type error. We can't have an array of types we don't understand or unknown sizes.

3 Structs

In a `struct`, the components are not all necessarily the same type or size. Because of this, padding is added to make all addresses differ by a multiple of a certain number of bytes. This value is known as the *alignment*. We say that the *alignment* of type T is A when addresses of objects of type T must be multiples of A. Here is a table showing the size and aligmnets of some common types.

Type	Size	Alignment
`char`	1	1
`short`	2	2
`int`	4	4
`int16_t`	2	2
`int64_t`	8	4
`char*`	4	4
`int x[8]`	32	4

The alignment of a variable must be at most its size, otherwise when you divide by the size of the variable, you would get incorrect indices. For example, the alignment of a short is 2, since if it were bigger, memory would look something like this:

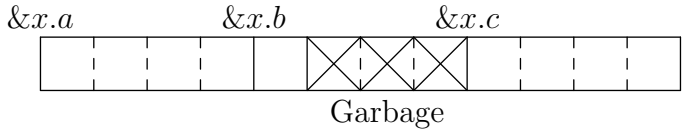


Dividing the difference in addresses by `sizeof(short)` would make the C abstract machine think that there is an extra short in the array. Additionally, the alignment of an array is the same as the alignment of a component of the array. The order of the elements of the struct affects the size it takes up. For example, consider the following two structs.

```
struct
{
    int a;
    char b;
    int c;
    char d;
}
```

```
struct
{
    int a;
    int b;
    char c;
    char d;
}
```

Both of these structs have an alignment of 4. The struct on the left pads after the first char and the second, so it occupies a total of 16 bytes. The first twelve bytes are structured like this:



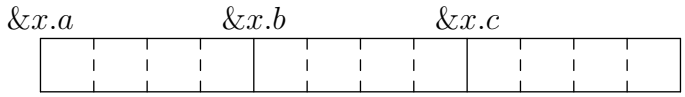
The padding just consists of garbage values. On the other hand, the struct on the right only pads once after the second char, so it occupies 12 bytes total. The alignment of a struct is equal to the maximum alignment of the components. If you have a basic struct like

```
struct {
    int x;
}
```

this will be indistinguishable from a single `int`. If you have the struct

```
struct {
    int x;
    int y;
    int z;
}
```

then this will be indistiguishable from an array of three `int`s.



4 Union

All of the objects within a union type overlap with one another. Therefore, the size of a union is the maximum size of its constituents. In the union given by

```
union s {
    char x;
    char b;
    char d;
    char e;
    char f;
    //int f;
}
```

the size is 1, and uncommenting out the `int` will make it size 4.

5 Stacksmash

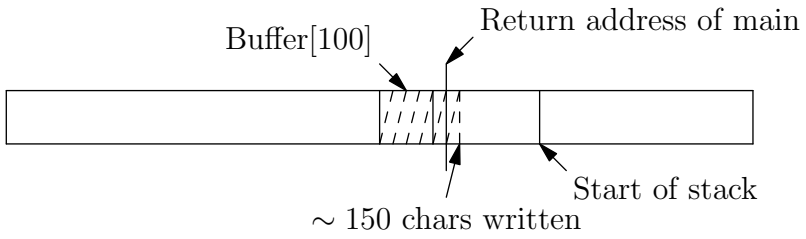
NEVER use `gets()`, it is deprecated. Look at the following code:

```
// stacksmash.c
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

int main() {
    char buffer[100];
    if (gets(buffer))
        printf("Read %d character(s)\n", strlen(buffer));
    else
        printf("Read nothing\n");
}
```

The `gets()` function does nothing to check for buffer overflows. It will just continue writing past the buffer as long as there are additional characters to write. This is problematic because you could overwrite the return address on the stack, which is in `0xbfffffff`-land.



Some useful command line arguments for testing include:

1. `head -n 10` prints the first 10 lines of the output stream
2. `head -c 10` prints the first 10 characters of the output stream
3. `tr -d '\n'` strips all newlines from output stream

If you want to test buffer overflow bugs, you can try inputting strings of variable length, and seeing what length first causes the crash. You can also use Valgrind to look for memory errors. Using a function like `fgets()` will limit the amount of data that can be written into the buffer.