Accessing memory

Hacking in C
Thom Wiggers

iCIS | Digital Security
Radboud University

Table of Contents

Table of Contents

Allocation of multiple variables

Consider the program

```
main(){
  char x;
  int i;
  short s;
  char y;
  ....
}
```

What will the layout of this data in memory be?

Assuming 4-byte ints, 2-byte shorts, and little endian architecture

Printing addresses where data is located

We can use & to see where data is located

```
char x; int i; short s; char y;

printf("x is allocated at %p \n", &x);
printf("i is allocated at %p \n", &i);
printf("s is allocated at %p \n", &s);
printf("y is allocated at %p \n", &y);

// Here %p is used to print pointer values
```

Compiling with or without -O2 may reveal different alignment strategies

Data alignment

Memory as a sequence of bytes

...		**x**	$\mathbf{i}_0$	$\mathbf{i}_1$	$\mathbf{i}_2$	$\mathbf{i}_3$	$\mathbf{s}_0$	$\mathbf{s}_1$	**y**			...

But on a 32-bit machine, the memory is a sequence of 4-byte words.

x	$\mathbf{i}_0$	$\mathbf{i}_1$	$\mathbf{i}_2$
$\mathbf{i}_3$	$\mathbf{s}_0$	$\mathbf{s}_1$	**y**
			...

Now the data elements are not nicely aligned with the words,
which will make execution slow, since CPU instructions act on words.

Data alignment

Different allocations, with better/worse alignment

x	i_0	i_1	i_2
i_3	s_0	s_1	**y**
			...

Lousy alignment, but uses minimal memory

x			
i_0	i_1	i_2	i_3
s_0	s_1		
y			

Optimal alignment, but wastes memory

s_0	s_1	**x**	**y**
i_0	i_1	i_2	i_3
			...

Possible compromise

Data alignment

Compilers may introduce padding or change the order of data in memory to improve alignment.

There are trade-offs here between speed and memory usage.

Most C compilers can provide many optional optimizations. E.g., use

```
man gcc
```

to check out the many optimization options of gcc.

Table of Contents

Arrays

An array contains a collection of data elements with the same type.
The size is fixed after defining an array.

```
int test_array[10];
int a[] = {30, 20};
test_array[0] = a[1];

printf("oops %d \n", a[2]);

// will compile & run
```

Array bounds are **not** checked.

Anything may happen when accessing outside array bounds (*undefined behaviour*).

The program may crash, usually with a segmentation fault (segfault).

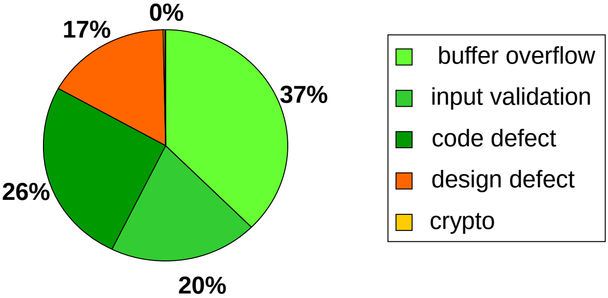
Array bounds checking

The historic decision **not** to check array bounds is responsible for in the order of 50% of all the security vulnerabilities in software, in the form of so-called buffer overflow attacks.

Other languages took a different (more sensible?) choice here. E.g. ALGOL60, defined in 1960, already included array bound checks.

Typical software security vulnerabilities

Security bugs found in Microsoft's first security bug fix month (2002)



Array bounds checking

Tony Hoare in Turing Award
speech on the design principles of ALGOL 60

"The first principle was security. . . A consequence of this principle is that every subscript was checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished us to provide an option to switch off these checks in the interests of efficiency. Unanimously, they urged us not to — they knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law."

[C.A.R.Hoare, The Emperor's Old Clothes, Communications of the ACM, 1980]

Overrunning arrays

Consider the program

```
int y = 7;
char a[2];
int x = 6;
printf("oops %d \n", a[2]);
```

What would you expect this program to print?

If the compiler allocates `x` directly after `a`, then (on a little-endian machine) it will print `6`.
There are no guarantees! The program could simply crash, or return any other number, re-format the hard drive, explode, . . .

By overrunning an array we can try to reverse-engineer the memory layout.

Arrays and alignment

The memory space allocated for an array is guaranteed to be contiguous, i.e. `a[1]` is allocated right after `a[0]`.

For good alignment, a compiler could again add padding at the ends of arrays.
E.g. a compiler might allocate 16 bytes rather than 15 bytes for

```
char text[15];
```

Array variables are references

If you take the following program

```
#include <stdio.h>
int main(void) {
    int arr[10] = {0};
    printf("arr = %p\n", arr);
}
```

You'll see that the output is

```
arr = 0x7ffc1fccb620
```

This is because the `int[]` type is actually a pointer!

Arrays are passed by reference

Arrays are always passed by reference.

For example, given the function

```
void increase_elt(int x[]) {
  x[1] = x[1]+23;
}
```

What is the value of `a[1]` after executing the following code?

```
int a[2] = {1, 2};
increase_elt(a);
```

`a[1] == 25`

Recall call by reference from Imperative Programming, OOP, whereever.

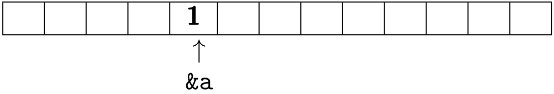
(Actually, we are still just passing by value, but we're passing the pointer to the array by its value!)

Table of Contents

Last week on pointers

Remember we could get the address of a variable using the & operator.

```
int a = 1;
printf("a is stored at %p\n", &a);
```



This allowed us to create a variable that stores this reference:

```
int* a_ptr = &a;
```

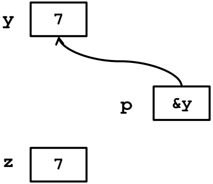
With the special type `int*`.

If we wanted to use this reference to write or read a, we needed to dereference.

```
*a_ptr = 42;
printf("a = %d\n", *a_ptr);
```

Confused? Draw some pointy arrows!

```c
int y = 7;
int *p = &y; // pointer p now points to cell y
int z = *p;  // give z the value of what p points to
```



Style debate: `int* p` or `int *p`?

What can be confusing in

```
int *p = &y;
```

is that this is an assignment to `p`, not `*p`

Some people prefer to write

```
int* p = &y;
```

Some C purists will argue this is C++ style.

Downside of writing `int*`:

```
int* x, y, z;
```

declares `x` as a pointer to an `int` and `y` and `z` as `int`!

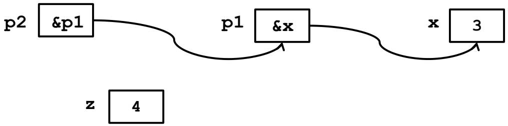
We must go deeper

```c
int x = 3;
... p1 = &x;
... p2 = &p1;
int z = **p2 + 1;
```

What will the value of `z` be?
What should the types of `p1` and `p2` be?

Pointerpointers

```
int x = 3;
int  *p1 = &x;   // p1 points to an int
int **p2 = &p1; // p2 points to a _pointer_ to an int
int z = **p2 + 1;
```



On breaking symmetry

So, `&` takes the address of a variable, and `*` undoes it:

```
int x = 1; *&x == x
```

However. . .

```
int x = 1; &*x  // SEGMENTATION FAULT!
```

There exists a type for which this *does* work!

Pointer puzzle

```
int y = 2;
int z = 3;
int* p = &y;
int* q = &z;
(*q)++;
*p = *p + *q;
q = q + 1;
printf("y is %d\n", y);
```

What is the value of `y` at the end?
6
What is the value of `*p` at the end?
6
What is the value of `*q` at the end?
We don't know! `q` points to some memory cell after `z` in the memory

Table of Contents

Pointer maths

As you've seen, it's perfectly possible to plusplus your pointer.
However, what that means exactly depends on the type of the pointer.
Doing `a_ptr + 1` will get you the address of the next location.
In other words: it will add `sizeof(a)` to `a_ptr`!
So for `char* x; int* y`
`x + 2` means *add 2 * sizeof(char)* $\rightarrow +2$
`y + 2` means *add 2 * sizeof(int)* $\rightarrow +8$

Pointers into array

You're probably used to iterating over arrays

```
uint8_t x[100] = {0};
for (size_t idx = 0; idx < 100; idx++) {
  printf("x[%zu] = %hhd\n", idx, x[idx]);
}
```

However, we can also do this via pointers!

```
uint8_t x[100] = {0};
for (uint8_t* x_ptr = &x, size_t idx = 0; idx < 100; idx++)
  printf("x[%zu] = %hhd\n", idx, *(x_ptr + idx));
}
```

In fact, `x[idx]` is equivalent to `*(x + idx)`!

In fact, you could even write `idx[x]` or `7[x]`!

Looping via pointer

Another way to iterate over your array:

```
uint8_t x[100] = {0};
for (uint8_t* x_ptr = &x; x_ptr < &(x[100]); x_ptr++) {
  printf("x[%zu] = %hhd\n", idx, *(x_ptr + idx));
}
```

This may be a bit of a silly example, but some buffers lend themselves to this more...

Potential of pointers: inspecting raw memory

To inspect a piece of raw memory, we can cast it to a `unsigned char*` or `uint8_t*` and then inspect the bytes

```
#include <stdio.h>
#include <stdint.h>
#include <math.h>
int main(void) {
    double f = M_PI;
    uint8_t* p = (uint8_t*)&f;
    printf("The representation of double %lf is:\n\t0x", f);
    for (size_t i = 0; i < sizeof(double); i++, p++) {
        printf("%hhx", *p);
    }
    printf("\n");
}
```

```
The representation of double 3.141593 is:
        0x182d4454fb21940
```


This of course completely relies on undefined behaviour (reading out of

Turning pointers into numbers

`intptr_t` defined in `stdint.h` is an integral type that is guaranteed to be wide enough to hold pointers.

```
int *p; // p points to an int
intptr_t i = (intptr_t) p; // the address as a number
p++;
i++;
// Will i and p be the same?
```

- No! `i++` increases by 1, `p++` with `sizeof(int)`
- There is also an unsigned version of `intptr_t`: `uintptr_t`
- Useful, for example, if you want to compute the distance between two addresses.
  - Subtracting pointers that point to different objects is UB!

Table of Contents

C-strings

- You may have noticed that there is no `string` type!
- Even though we've been using strings…
- A string is just an array of `char`s…
- extremely important: …terminated by a `NULL` byte (`'\0'`)!
- Note that only strings are null-terminated!
- So `"Thom"` has a length of 5!
- The type of a `"string literal"` is `const char*`.
  - It points to a location in the memory of the compiled binary!
  - (The standard says `char[]`, and not `const`, but it's UB to modify it)
  - There is a warning that helps find this bug `-Wwrite-strings`
    - It is not enabled by `-Wall`.

Looping over C-strings

We might use pointers to loop over a string:

```
const char* text = "It's pining for the fjords.";
for (char* letter = text; letter != '\0'; letter++) {
  printf("%c", letter);
}
printf("\n");
```

What if you somehow forget to set or broke the `NULL` byte...

Looping over C-strings: strlen

We might use pointers to count the characters in a string:

```
const char* text = "My hovercraft is full of eels.";
size_t len = 0;
for (char* letter = text; letter != '\0'; letter++) {
  len++;
}
printf("text is %zu chars\n", len);
```

We've just reinvented `strlen`!

How this goes horribly wrong

How does this go horribly wrong?

```
const char* ximinez = "Nobody expects the Spanish "
                      "Inquisition!";
char copy[100];
memcpy(copy, ximinex, strlen(ximinez));
printf("%s\n", copy);
```

There is no terminating `'\0'`!

Solutions:

- `copy[strlen(ximinez)+1] = '\0'`
- `memcpy(copy, ximinez, strlen(ximinez)+1);`
- `strcpy(copy, ximinez);`
  - Of course, `strcpy` was designed for this...

How `strcpy` breaks

```
const char *ximinez = "Our chief weapon is surprise,"
                      "fear and surprise";
char palin[10];
strcpy(palin, ximinez);
```

That doesn't fit: buffer overflow.

Solution: use `strncpy(dst, src, n)`: copies at most `n` characters.

How `strncpy` still breaks

`strncpy(dst, src, n)` does not make sure the target is `NULL`-terminated!

If there is no `NULL` byte within the `n` bytes of `src`, `dst` will not be `NULL`-terminated.

Yet more patchwork:

```
const char* ximinez = "Our chief weapon is surprise, "
                      "surprise and fear, "
                      "fear and surprise.";
char palin[10];
strncpy(palin, ximinez, 9);
buf[9] = '\0';
```

Spot the bug

```
int main(int argc, char* argv[]) {
  char buf[10];
  // copy name of program to buf
  strcpy(buf, argv[0], strlen(argv[0]));
}
```

We are taking the size of the source buffer here.

Even the name of the program is user input!

`ln -s program my_longer_name_that_crashes_this`

(`ln -s target linkname` creates a *symbolic link*.)

Wait, what's the deal with `argv` anyway

- The signature of the `main` function in C is `int main(int argc, char* argv[])`.
  - Alternatively, `char** argv`
  - We may leave out the `int argc, char* argv[]` part.
- The returned `int` is the status code
  - Anything that's not zero means error
- `argc` is the argument count.
- `./main arg1 arg2 arg3` means `argc == 4`.
- "`./main`" is always the first argument!
- `argv` is an array of character pointers
- Alternatively, a pointer to some pointers

Handling command line arguments

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    printf("argc = %d\n", argc);
    for (int idx = 0; idx < argc; idx++) {
        printf("Argument %d: \"%s\"\n", idx, argv[idx]);
    }
}
```

```
$ ./commandlineargs a b c d
argc = 5
Argument 0: "./commandlineargs"
Argument 1: "a"
Argument 2: "b"
Argument 3: "c"
Argument 4: "d"
```

Safer strings and array?

There is no reason why programming language should not provide safe versions of strings (or indeed arrays).
Other languages offer strings and arrays which are safer in that, for example:

1. Going outside the array bounds will be detected at runtime
2. Which will be resized automatically if they do not fit
3. The language will ensure that all strings are null-terminated

More precisely, the programmer does not even need to know how strings are represented, and whether null-terminator exists and what they look like: the representation of strings is completely transparant/invisible to the programmer.
Moral of the story: if you can, avoid using standard C strings.
E.g. in C++, use `std::string`; in C, use safer string libraries.
An extension to C11 defines for example
`strncpy_s(dst, dstsize, src, srcsize)`.

Table of Contents

Homework

- Homework: continue with stuff from last week