Hacking in C

Alignment, arrays and pointers

Radboud University, Nijmegen, The Netherlands

Spring 2018

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 2 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` will reveal different alignment strategies

Data alignment

Memory as a sequence of bytes

...		**x**	$\mathbf{i}_4$	$\mathbf{i}_3$	$\mathbf{i}_2$	$\mathbf{i}_1$	$\mathbf{s}_2$	$\mathbf{s}_1$	**y**			...

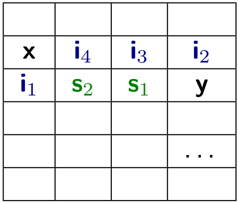
But on a 32-bit machine, the memory is a sequence of 4-byte words

x	$\mathbf{i}_4$	$\mathbf{i}_3$	$\mathbf{i}_2$
$\mathbf{i}_1$	$\mathbf{s}_2$	$\mathbf{s}_1$	**y**
			...

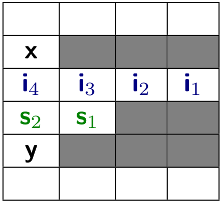
Now the data elements are not nicely aligned with the words,
which will make execution slow, since CPU instructions act on words.

Data alignment

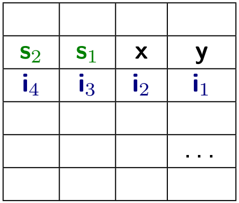
Different allocations, with better/worse alignment



Lousy alignment, but useses minimal memory



Optimal alignment, but wastes memory



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 optimisations. Eg use

```
man gcc
```

to check out the many optimisation options of gcc.

Arrays

Arrays

An array contains a collection of data elements with the same type. The size is constant.

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

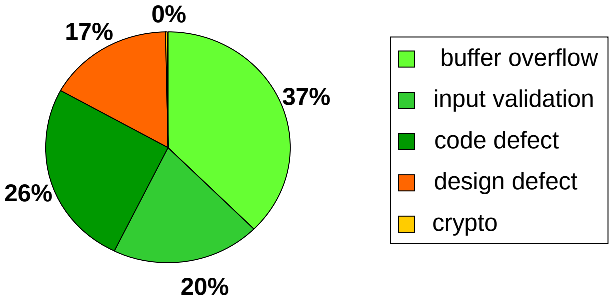
Array bounds checking

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)



Here *buffer overflows* are platform-specific.
Some of the *code defects* and *input validation* problems might also be.
Crypto problems are much more rare, but can be of very high impact.

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

What would you expect this program to print?

If the compiler allocates `y` directly after `a`, then 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 end of arrays.
E.g. a compiler might allocate 16 bytes rather than 15 bytes for

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

Arrays are passed by reference

Arrays are always passed by reference.

For example, given the function

```
void increase_elt(int x[0]) {
  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);
```

25

Recall call by reference from Imperatief Programmeren

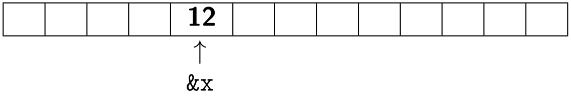
Pointers

Retrieving addresses of *pointers* using &

We can find out *where* some data is allocated using the & operation.
If

```
int x = 12;
```

then `&x` is the memory address where the value of `x` is stored,
aka a pointer to `x`.



It depends on the underlying architecture how many bytes are needed to represent addresses: 4 on 32-bit machines, 8 on a 64-bit machine.

Declaring pointers

Pointers are typed:
the compiler keeps track of what data type a pointer points to

```
int *p;   // p is a pointer that points to an int
float *f; // f is a pointer that points to a float
```

Creating and dereferencing pointers

Suppose

```
int y, z; int *p; // i.e. p points to an int
```

How can we create a pointer to some variable? Using **&**

```
y = 7
p = &y; // assign the address of y to p
```

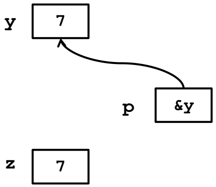
How can we get the value that a pointer points to? Using *****

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

Looking up what a pointer points to, with $*$, is called dereferencing.

Confused? draw pictures!

```
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
```



Read Section 9.1 of “Problem Solving with C++” for another explanation.

Pointer quiz

What is the value of `y`?

```
int y = 2;
int x = y;
y++;
x++;
```

3

What is the value of `y`?

```
int y = 2;
int *x = &y;
y++;
(*x)++;
```

4

Note that * is used for 3 different purposes, with 3 different meanings

1. In declarations, to declare pointer types

   ```
   int *p;  // p is a pointer to an int
            // i.e.  *p is an int
   ```

2. As a prefix operator on pointers

   ```
   int z = *p;
   ```

3. Multiplication of numeric values

   Some legal C code can get confusing, e.g.

   ```
   z = 3 * *p
   ```

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;
```

but 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`...

Still not confused?

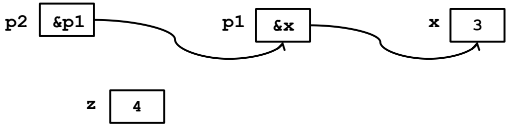
```
x = 3;
p1 = &x;
p2 = &p1;
z = **p2 + 1;
```

What will the value of `z` be?

What should the types of `p1` and `p2` be?

Still not confused? pointers to pointers

```
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;
```



Pointer test (Hint: example exam question)

```
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