The programming language C (part 2)

alignment, arrays, and *pointers*

Last week

- The different data types in C

  `char, int, long, ...`

  in **`signed`** and **`unsigned`** versions
- Implicit conversions and explicit casts between these types
- 2-complement and big/little endian representations

Apart from the question of representation (*how* data is represented) there is also the question of allocation (*where* data is allocated)

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 allocated

We can use **&** to see if where compiler allocated data

```
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	i_4	i_3	i_2	i_1	s_2	s_1	y		...

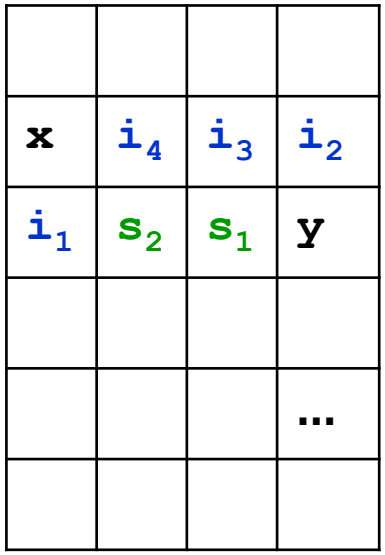
But on 32-bit machine, the memory be a sequence of 4-byte words

x	i_4	i_3	i_2
i_1	s_2	s_1	y
			...

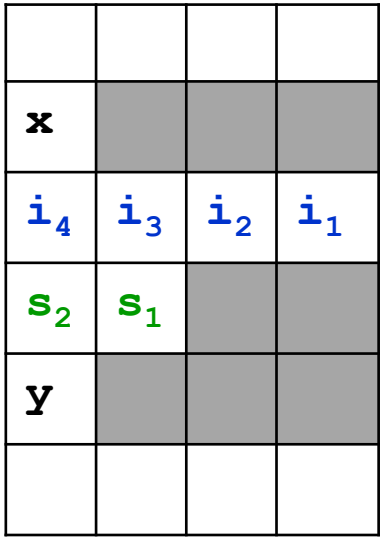
Now the data elements are not nicely aligned with the words,
which will make execution slow, since CPU instructions act on words.

data alignment

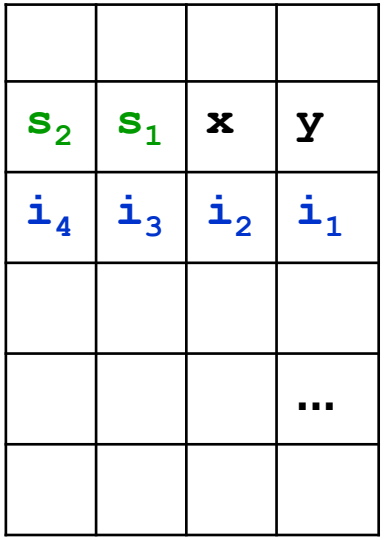
Different allocations, with better/worse alignment



lousy alignment, but using minimal memory



optimal alignment, but wasting 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 %i \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.

Eg ALGOL60, defined in 1960, already included array bound checks.

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 %i \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 a array is guaranteed to be contiguous

ie **`a[1]`** is allocated right after **`a[0]`**

For good alignment, a compiler could again add padding at the end of arrays.

eg a compiler might allocate16 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[])   { 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 or *pointers* using &

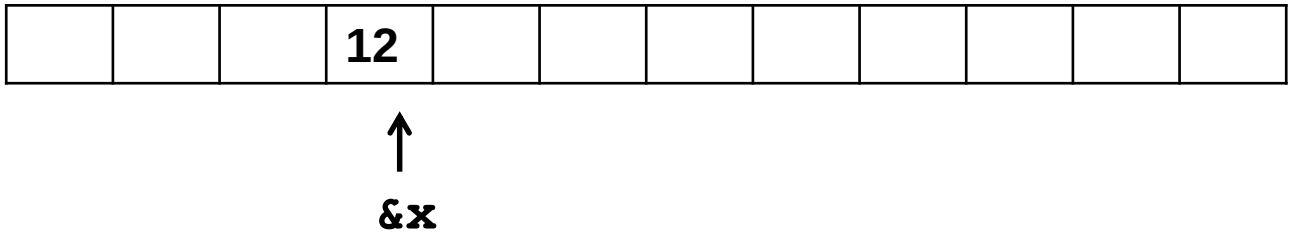
We can find out *where* some data is allocated using the & operation.

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 machine, 8 on 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; // ie. 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
```

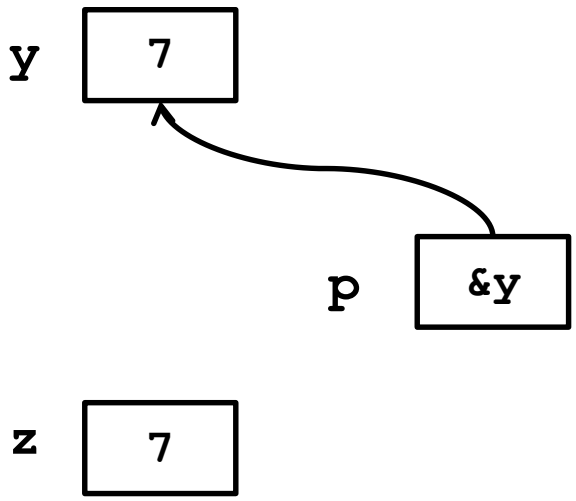
- How can we get the value that a pointer points to? Using *****

```
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

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

What is the value of **y**?

3

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

What is the value of **y**?

4

Note that **`*`** is used for 3 different purposes

1. in declarations, to declare pointer types

```
int *p;  //  p is a pointer to an int
         //  ie. *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, eg **`z = 3 * *p;`**

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

What can be confusing in

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

is that this an assignment to `p`, not to `*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 pointer to an `int` and `y` and `z` as `int`...

still not confused?

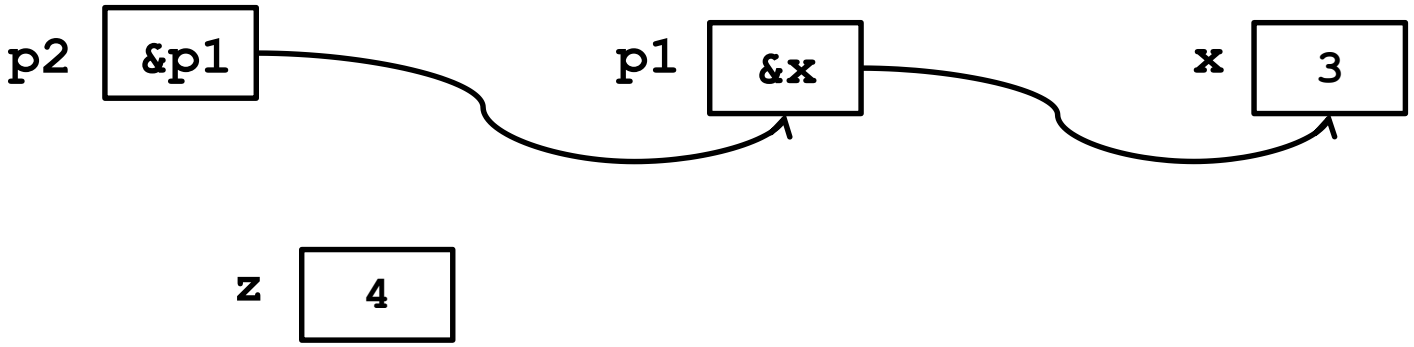
```
x = 3;
p1 = &x;
p2 = &p1;
z = **p2 + 1;
```

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 refresher (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 %i\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!!!!! It is the address where **`z`** *is allocated plus* **`sizeof(int)`**, *ie* **`&z + sizeof(int)`**

pointer arithmetic

Pointers can be added to and subtracted from.

The semantics depends on the *type of the pointer:*

adding 1 to a pointer will go to the “next” location,

given the size of the data type that it points to.

For example, if

```
int  *ptr;
char *str;
```

then

`ptr + 2` means `ptr + 2 * sizeof(int)`

`str + 2` means `str + 2`

because `sizeof(char)` is 1

pointer arithmetic for strings

What is the output of

```
char *msg = ”hello, world”;
char *t = msg + 6;
printf(”t points to the string %s.”, t);
```

This will print

```
t points to the string world.
```

using pointers as arrays

The way pointer arithmetic works means that

a pointer to the head of an array behaves like an array.

Suppose

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
int *p = (int*) &a; // the address of the head of a
                    // treated as pointer to an int
```

Now

`p+3`

points to

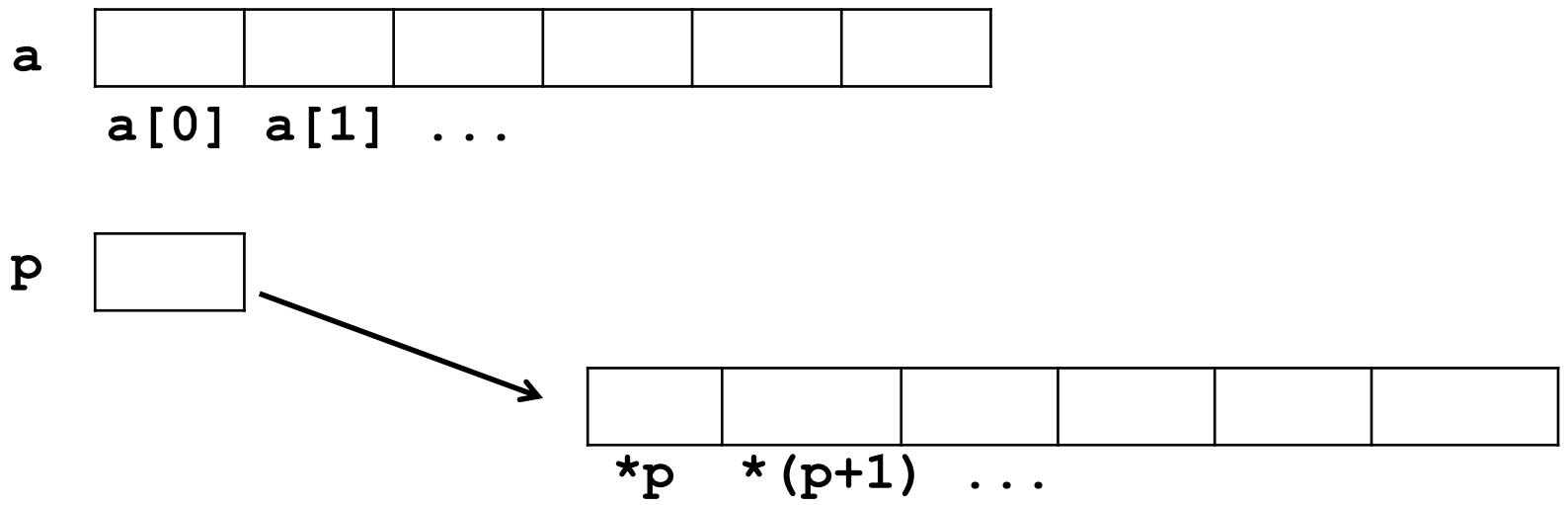
`a[3]`

so we use addition to pointer `p` to acces the array

arrays vs pointers

Arrays and pointers behave similarly, but are very different in memory

Consider `int a[];` `int *p;`



A difference: **`a`** will always refer to the same array,
whereas **`p`** can point to different arrays over time

using pointers as arrays

Supposes

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
```

Then

```
int sum = 0;
for (int i=0; i!=10; i++) {
  sum = sum + a[i];
}
```

can also be implemented using pointer arithmetic

```
int sum = 0;
for (int *p=(int*)&a; p!=&(a[10]); p++){
  sum = sum + *p;
}
```

but nobody in their right mind would ☺

This cast is needed because **a** is an integer array, so **&a** is a pointer to **int[]**, not pointer to an **int.** An alternative would be to write ***p = &(a[0])**

A problem with pointers: ...

```
int i; int j; int* x;
...
// lots of code omitted
i = 5;
j++;
// what is the value of i here?
(*x)++;
// what is the value of i here?
```

5

5 or 6, depending on whether `*x` points to `i`

A problem with pointers: ***aliasing***

Two pointers are called **aliases** if they point to the same location

```
int i = 5;
int* x = &i;
int* y = &i;
// x and y are aliases now
(*x)++;
// now i and *y have also changed to 6
```

Keeping track of pointers, in the presence of potential aliasing, can be really confusing, and really hard to debug...

The potential of pointers: inspecting raw memory

To inspect a piece of raw memory, we can cast it to a

`unsigned char*`

and then inspect the bytes

```
float  f = 3.14;
unsigned char *p = (unsigned char*) &f;
printf("The representation of float %f is", f);
for (int i; i <sizeof(float); p++;) {
    printf("%i", *p); i++;
  }
 printf("\n");
```

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 number
p++;
i++;
// Will i and p be the 'same'?
// No! i++ increases by 1, p++ with sizeof(int)!
```

There is also an unsiged version of **intptr_t**: **uintptr_t**

strings

strings

Having seen arrays and pointers, we can now understand C strings

```
char  *s = "hello world\n";
```

C strings are **`char`** arrays, which are terminated by a special null character aka null terminator, which is written as \0

Just like other arrays, we can use both the array type **`char[]`** and the pointer type **`char*`** for them.

There is some special notation for string literals, between double quotes, where this null terminator is implicit.

string problems

Working with C strings is highly error prone!

There are two problems:

1. as for any array, there are no array bounds checks;
   so it's the programmers responsibility not to go outside the array bounds
2. moreover, it is also the programmer's responsibility to make sure that the string is properly terminated with a null character.
   If a string lacks its null terminator, eg due to problem 1, then standard functions to manipulate strings will go off the rails.

safer strings and arrays?

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:

- going outside the array bounds will be detected at runtime (eg Java)
- which will be resized automatically if they do not fit (eg Python)
- the language will ensure that all strings are null-terminated (eg C++, Java, and python)

  More precisely, the programmer does not even have 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.

Eg in C++, use C++ type strings; in C, use safer string libraries.

a final string peculiarity

String literals, as in

```
char *msg = "hello, world";
```

are meant to be constant or read-only: you are not supposed to change the characters that make up a string literal.

Unfortunately, this does not mean that C will *prevent* this. It only means that the C standard defines changing a character in an string literal as having undefined behaviour ☹

Eg

```
char *t = msg + 6; *t = ';';
```

has undefined behaviour, ie. anything may happen

compilers can emit warnings if you change string literals, eg

```
gcc -Wwrite-strings
```

Recap

We have seen

- the different C types
  - primitive types

    `(unsigned) char, short, int, long, long, float ...`
  - implicit conversions and explicit conversions (casts) between them
  - arrays `int[]`
  - pointers `int*` with the operations `*` and `&`
  - C strings, as special `char` arrays
- their representations
- how these representations can be `broken', ie. how we can inspect and manipulate the underlying representation (eg. with casts)
- some things that can go wrong
  eg due to access outside array bounds or integer under/overflow