pointers (continued), arrays, and strings

Last week

We have seen pointers, eg of type **`char *p`**
with the operations **`*`** and **`&`**

- These are tricky to understand, unless you draw pictures

pointer arithmetic

You can use **+** and **–** with pointers.

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

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 access the array

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);
```

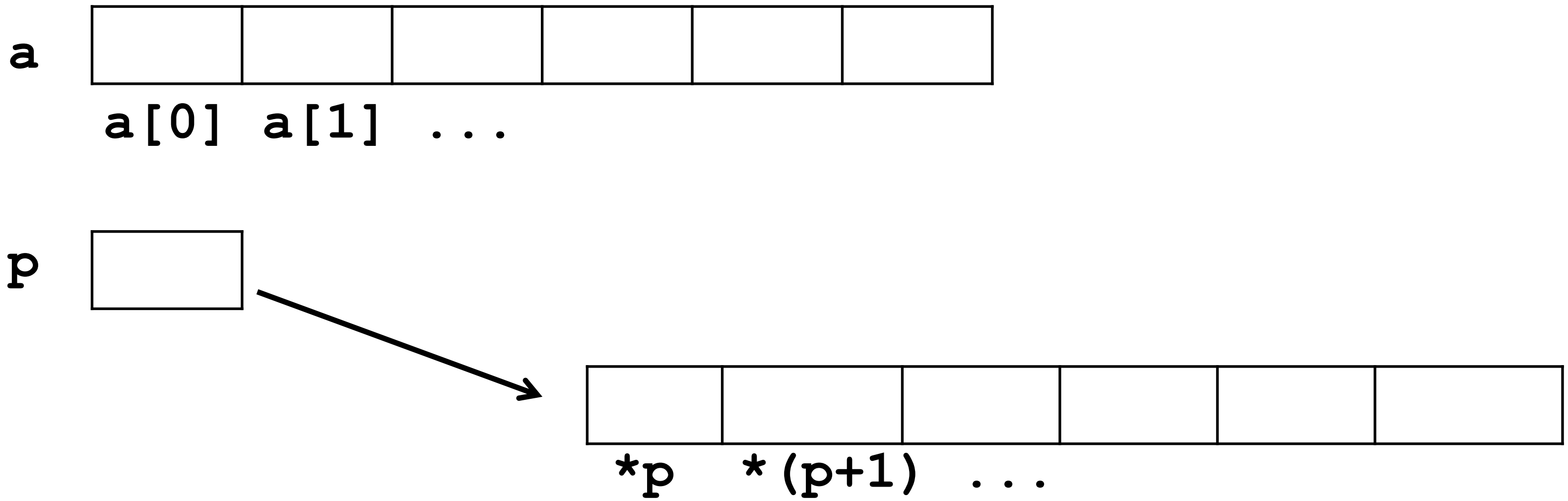
This will print

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

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])`**

Instead of **`p!=&(a[10])`** we could also write **`p != ((int*)&a)+10`**

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...

Recap – so far

We have seen pointers, eg of type **`char *p`**
with the operations **`*`** and **`&`**

- These are tricky to understand, unless you draw pictures

We can have aliasing, where two names, say **`*p`** and **`c`**, can refer to the same variable (location in memory)

We can use pointer arithmetic, and eg write **`*(p+1)`**,
and use this to access arrays

- Confusingly, the meaning of addition for pointers depends on their type, as **`+1`** for pointers of type **`int*`** really means **`+sizeof(int)`**

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

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

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

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. 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:

1. going outside the array bounds will be detected at runtime (eg Java)
2. which will be resized automatically if they do not fit (eg Python)
3. 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