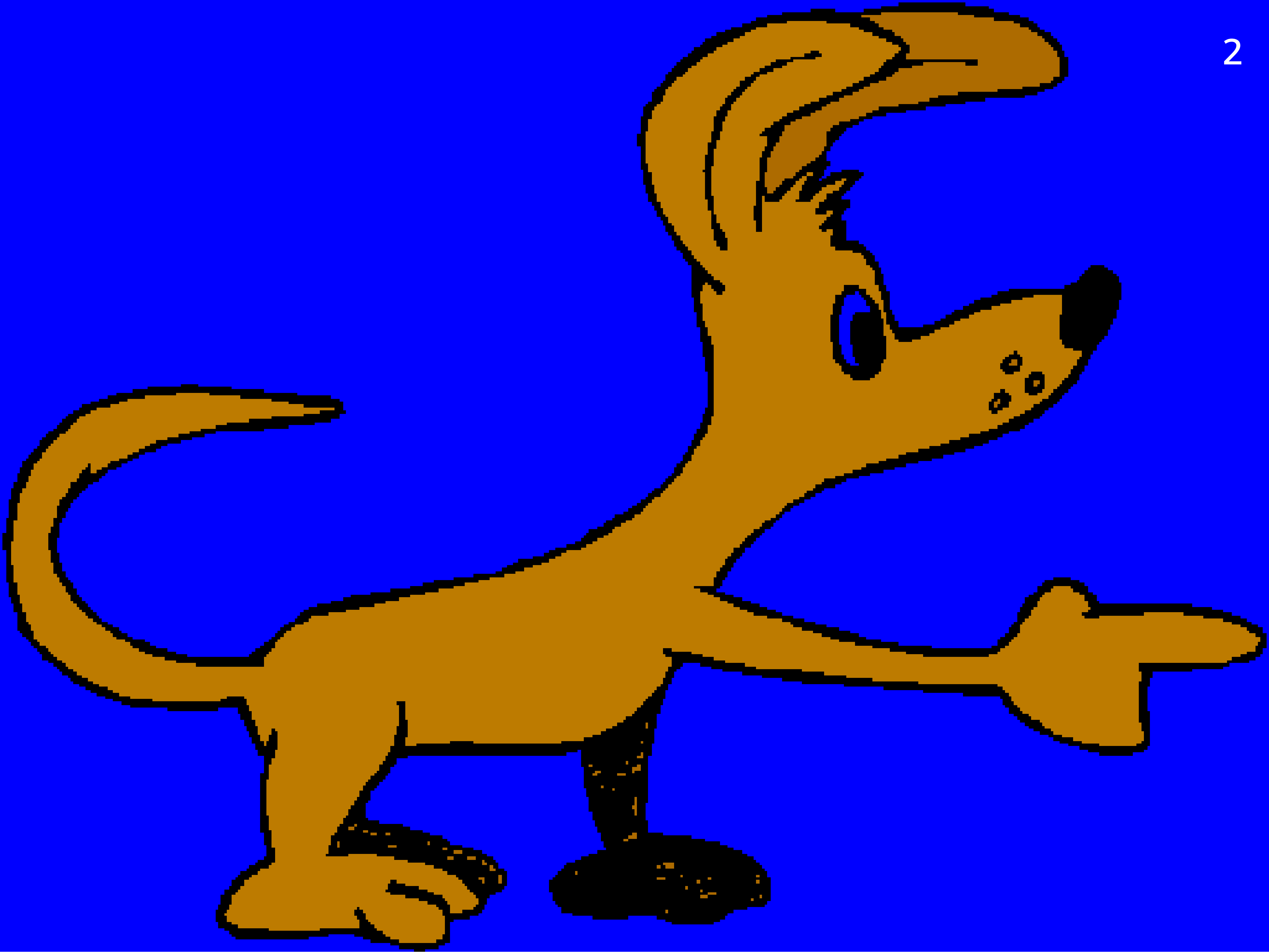
Pointers



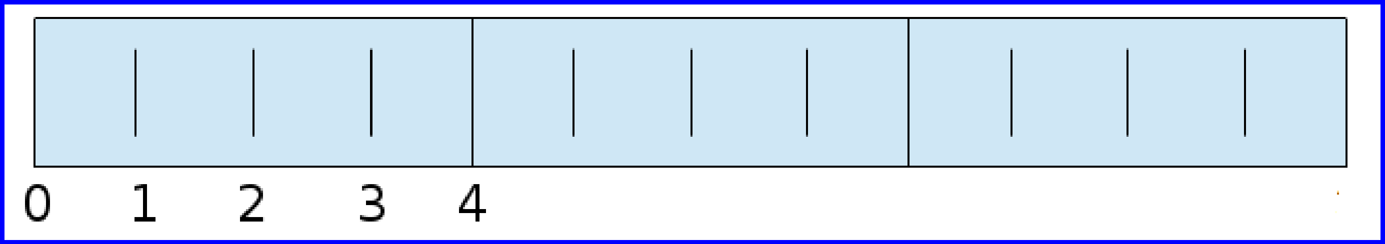
What are pointers for?

A pointer allows you to store something such as a string or array that has a different size at different times

This is most often used with dynamic allocation, which allows an item's lifetime to be more flexible than the duration of a single function call

Memory

A computer's memory is an array of bytes:



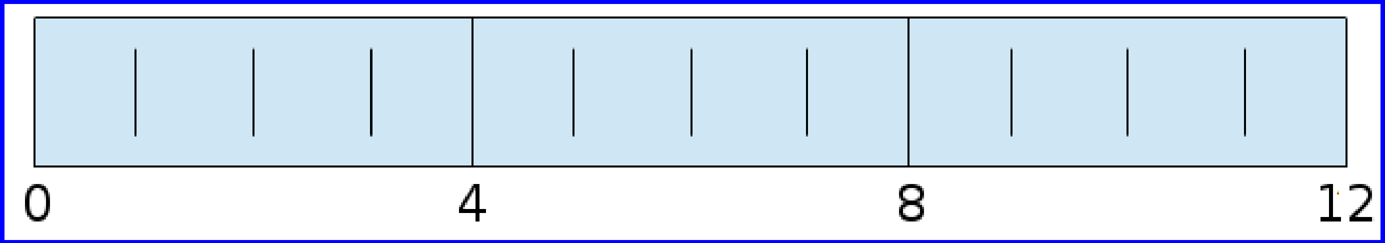
You can pick out one byte using its index

That's called the *address* of the byte

Real addresses are very occasionally bit-based or word-based, but the C language allows you to assume they are always byte-based

Multi-byte values

Suppose a section of memory holds `int`s



The address of an `int` is still a byte-address

Int addresses go up in 4's (assuming 4-byte `int`s)

But if the ints form an array, you want to index it by `0,1,2...`, not `0,4,8...`

Pointers

A *pointer* is an address in memory, together with the type and size of the item stored at that address

The type of a pointer to an `int` is `int *` ('int pointer')

C was carefully designed to give direct and total access to pointers, but without worrying about exactly what addresses are actually used at run time (which may be different from run to run and computer to computer)

Pointer variables

Pointers can be stored in memory, in variables

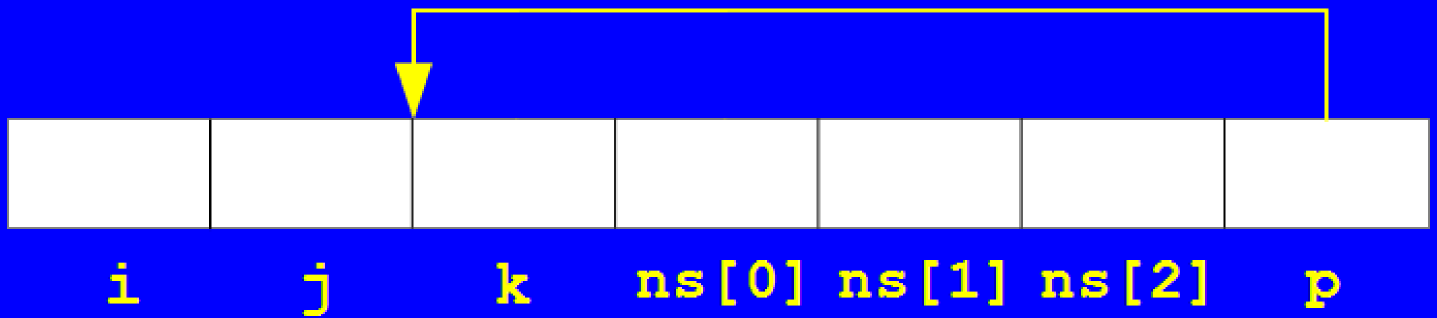
```
int i, j, k;
int ns[3];
int *p;
```

`p` is declared to be of type `int *` and must point to the beginning or end of an actual `int` in memory

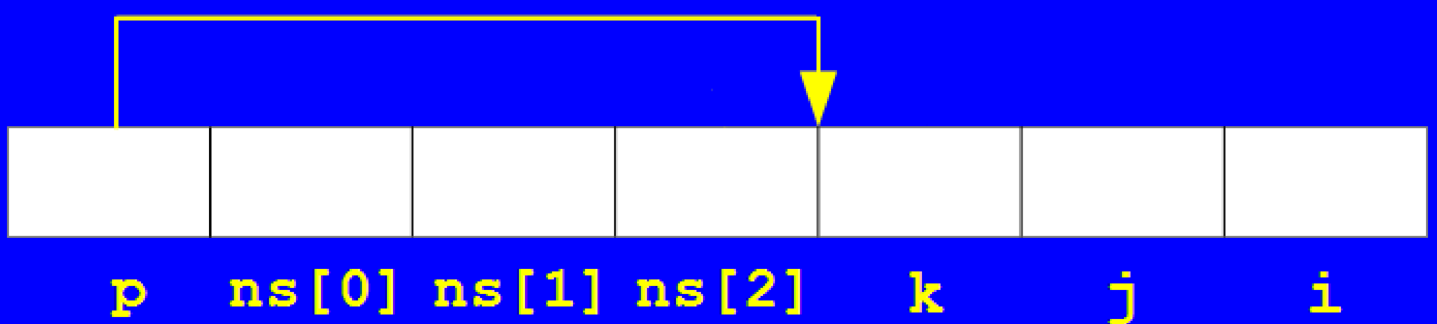
For example, `p` could point to the location of `i` or of `j` or of `k`, or to any of the elements of the array `ns`, or to the end of the `ns` array

Picturing pointers

It is important, when programming or debugging, to create pictures of pointers, in your head or on paper

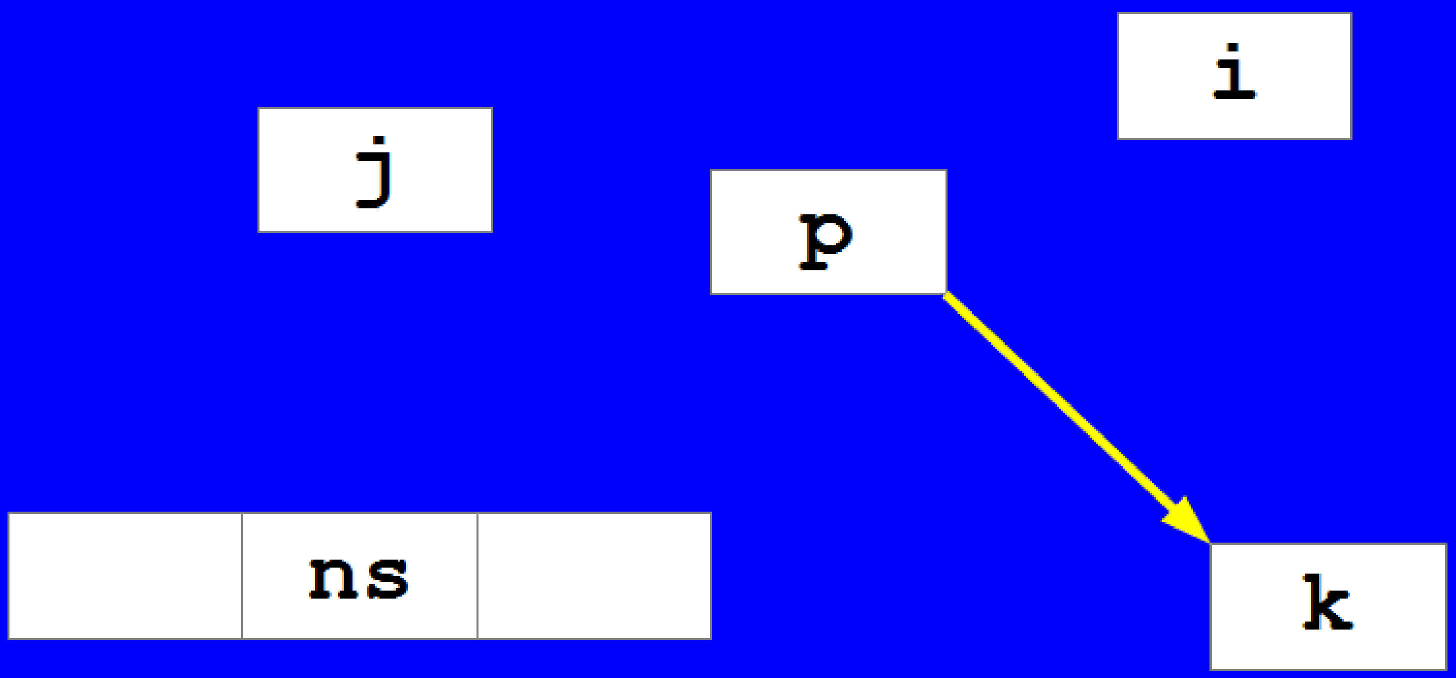


We don't know how memory is allocated, so the picture of p pointing to k could equally well be



Picturing pointers

Since we don't know (and don't need to know) where things are located in memory, we often picture them 'randomly' scattered:



Poor notation

Should you write `int *p` or `int* p` ?

In the first case, beware that `int *p = x;` means `int *p; p = x;` even though it looks like `int *p; *p = x;`

In the second case, beware that `int* p, q;` means `int* p; int q;` even though it looks like `int* p; int* q;`

It is a no-win situation, so let's follow the most common convention and write `int *p`

Reason for notation

Why are C's types written in this way?

With pointers, types can get very complicated, and the designers wanted types to be written the same way round as the operations performed on the variables, not the opposite way round

A declaration is written as an example of using the variable, plus the basic type you reach at the end

So `int *p` means "`p` is a variable to which you can apply the `*` operator, and then you reach an `int`"

Pointer arithmetic

If a pointer variable `p` of type `int *` points to an `int`, then the expression `p+1` points one `int` further on

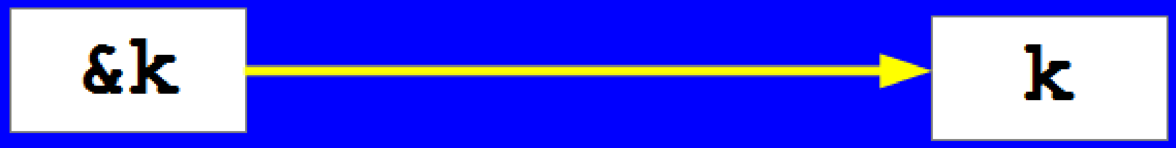
For example, if `p` points to `ns[1]` then `p+1` points to `ns[2]`

The C compiler uses the knowledge of the type of the item which a pointer points to, and its size, to make the arithmetic as convenient as possible

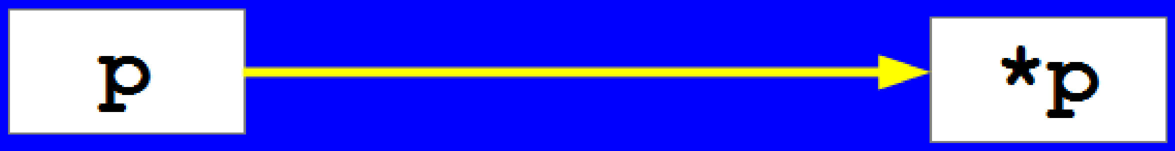
If you need to know a size (in bytes) yourself, apply the `sizeof()` pseudo-function to a variable or a type

Two operators

The & operator takes a variable, and creates a pointer to its memory location



The * operator takes a pointer, and follows it to find the value stored at that memory location



These go in 'opposite directions' along the pointer

The & operator

The & operator creates a pointer to a variable

```
/* Print a pointer. */                                pointer.c

#include <stdio.h>

int main() {
    int n;
    printf("pointer %p\n", &n);
}
```

The expression &n is often read "address of n", even though it should really be "pointer to n"

The * operator

The * operator finds the value which a pointer refers to

value.c

```
/* Print a value. */

#include <stdio.h>

int main() {
    int n = 42;
    int *p;
    p = &n;
    printf("value %d\n", *p);
}
```

Reminder

Here is a program which creates an array and puts a string in it

```
/* Demo: string in array */
#include <stdio.h>

int main() {
    char s[4];
    s[0] = 'c';
    s[1] = 'a';
    s[2] = 't';
    s[3] = '\0';
    printf("%s\n", s);
}
```

malloc and free

There are library functions `malloc` and `free` which allocate and deallocate new blocks of memory

```
/* Demo: string using malloc/free */
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *s = malloc(4);
    s[0] = 'c';
    s[1] = 'a';
    s[2] = 't';
    s[3] = '\0';
    printf("%s\n", s);
    free(s);
}
```

stdlib

The `stdlib` library contains the functions `malloc` and `free` so we need to include its header

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

Note: this provides the compiler with the declarations (signatures) of the library functions, so it knows how to generate calls

Note: the code of the library `stdlib`, as with `stdio`, is linked automatically with any program that needs it, so the `gcc` compiling line needs nothing special added

Calling malloc

The call to `malloc` allocates the memory

```
char *s = malloc(4);
```

The variable is declared as a pointer to the first element of an array, rather than an array

The argument to `malloc` is the number of bytes desired

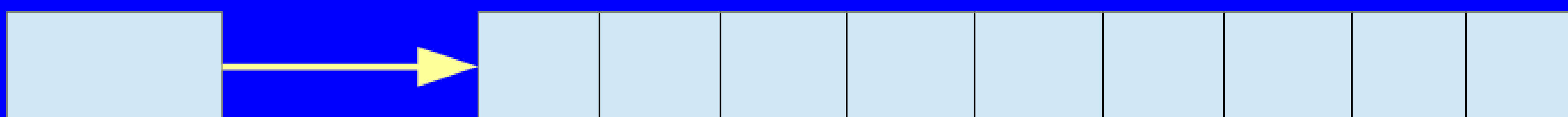
The return type of `malloc` is `void *` which means "pointer to something", compatible with all pointer types

Visualising malloc

You need to visualise the effect of malloc

```
char *s;
```


```
s = malloc(...);
```



Indexing

The new memory is indexed like an array

```
s[0] = 'c';
s[1] = 'a';
s[2] = 't';
s[3] = '\0';
```

The compiler allows array notation to be used on memory accessed via a pointer

In fact `s[i]` is just an abbreviation for `*(s+i)`

An array is not the ***same*** as a pointer to the start of an array, it is just treated the same by the compiler when it comes to indexing

Freeing

The new memory is freed explicitly when not needed any more

```
free(s);
```

The call is unnecessary ***in this case*** because the program is about to end, and all of its memory will be returned to the operating system

But it is good practice to free all dynamically allocated space, and to put the `free` 'near' the `malloc` call in such a way as to make it clear that the program has no memory leaks

The readline problem

Suppose you read a file in, one line at a time

You need a character array to store the characters in

You can reuse the array for each line, but how big should you make it?

In the old days, programmers would just guess a likely maximum size, e.g. 100 or 1000

This produces a program with the worst kind of bug - the kind which survives testing, bites users unpredictably, and leads to security loopholes

Flexible arrays

A flexible array is one which grows as needed (by reallocation and copying) to accommodate an initially unknown amount of data

Generally, the best strategy is to "start small and double"

Starting small means that if a program has lots of flexible arrays, they don't take up too much space

Doubling keeps the time cost of copying down, to match the amount of other activity

readline

Here's a 'proper' readline function:

readline.c

```
// Read line from a file, discard newline
char *readline(FILE *in) {
    int length = 0, capacity = 4;
    char *line = malloc(capacity);
    while (true) {
        fgets(line + length, capacity - length, in);
        if (feof(in)) break;
        length = strlen(line);
        char last = line[length-1];
        if (last == '\n' || last == '\r') break;
        capacity = capacity * 2;
        line = realloc(line, capacity);
    }
    line[strcspn(line, "\r\n")] = '\0';
    return line;
}
```

Setup

First, the variables are set up

```
int length = 0, capacity = 4;                readline.c
char *line = malloc(capacity);
```

`line` is the flexible array

`capacity` is its current size

`length` is the number of characters of the line read in so far (excluding the terminating null character)

Partial read

Here's the `fgets` call

```
fgets(line + length, capacity - length, in);
```

readline.c

The first argument is a pointer to the remaining space in the `line` array, after the characters which have already been read in (the next character may be null, but it will get overwritten)

The second argument is the remaining space in the array

End of line

How do you check whether `fgets` has finished reading the line?

```
length = strlen(line);                    readline.c
char last = line[length-1];
if (last == '\n' || last == '\r') break;
```

`fgets` includes the raw newline, so we need to check for both newline characters, to make the function reasonably platform independent

Reallocation

How do you change the capacity of the array?

```
capacity = capacity * 2;
line = realloc(line, capacity);
```

readline.c

The `realloc` function allocates a new array, copies the old array into the start of the new array, and deallocates the old array!

Matrix notation

Imagine writing a library module for matrices

It would be really nice to be able to write functions like this one for inverting an `nxn` matrix:

```
void invert(int n, double matrix[n][n]) { ... }
```

This uses VLA notation, which is very readable, avoiding pointer notation altogether

Inadequate tutorials

Most tutorials don't mention VLA notation

Of the rest, most give the impression that VLA notation cannot be used with dynamically allocated arrays

Most tutorials that discuss dynamic allocation don't mention multi-dimensional arrays

Of the rest, most either show one dimensional arrays with hand-calculated two-dimensional indexing

Or they show an array of dynamic pointers to rows

Little known

But here is a little-known approach

This `main` function uses the `invert` function we saw before

```
int main() {
    int n = 3;
    double (*p)[n][n];
    p = malloc(sizeof(*p));
    invert(n, *p);
}
```

Note `p` is a pointer to an array, which is very different from a pointer to the first element of an array

Limits

However, despite tricks, it is not possible to use pointer-free array notation in every circumstance

For example, it isn't possible to declare a structure with a variable-sized array embedded in it, because the C compiler insists on knowing, at compile time, how big the structure type is