Pointers

Three chapters

This chapter is about what pointers are

The next (objects) is about how to use pointers to pass data around in programs

The one after (memory) is about how to use pointers to manage memory in programs

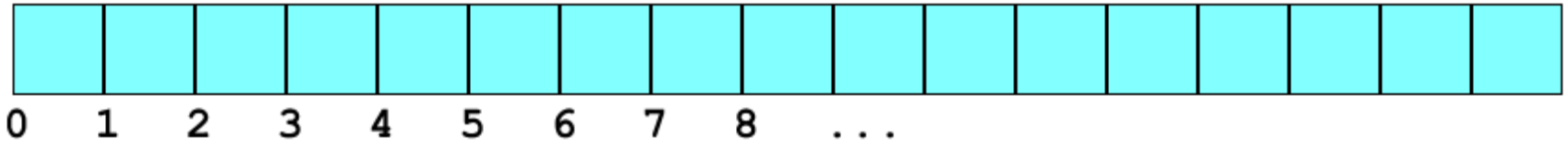
What are pointers for?

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

And it is used with dynamic allocation, to allow 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

In C, you can assume that addresses are always byte-based (not bit-based or word-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 gives you direct and total access to pointers, but without worrying about exactly what the addresses actually are

A pointer has 4 bytes (on a 32-bit system) or 8 bytes (on a 64-bit system, to beat the 4Gb limit)

Pointer variables

Pointers can be stored in memory, in variables

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

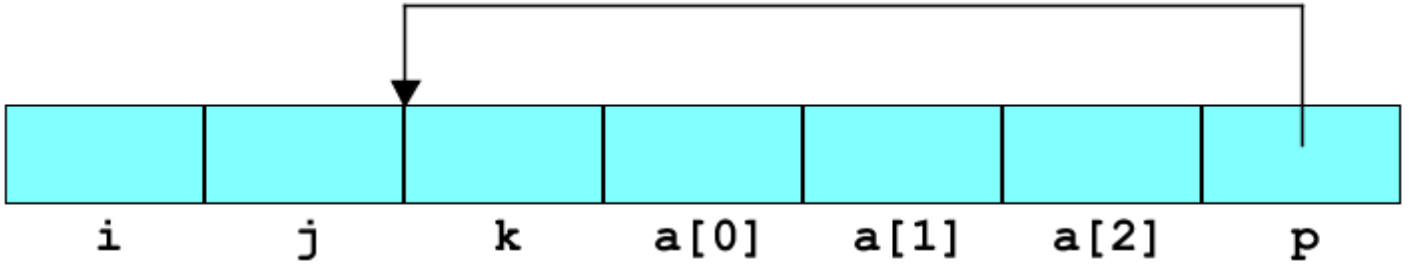
`p` is declared to be of type `int *` and must point to the beginning or end of an actual `int` in memory

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 `a`, or to the end of the array

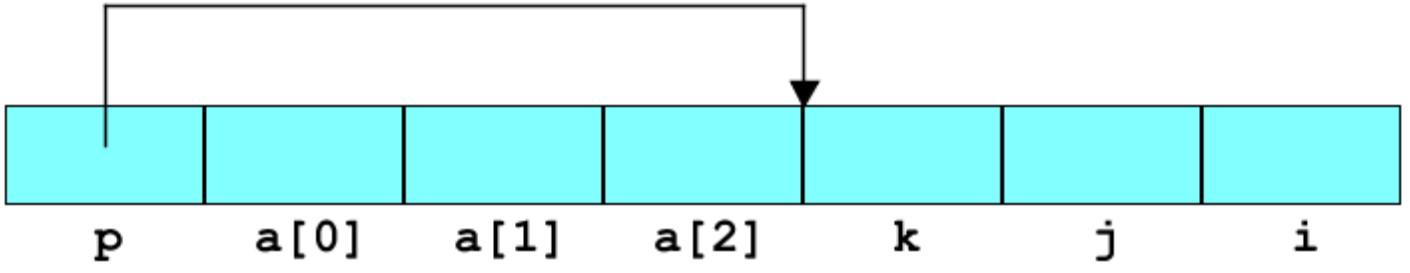
Suppose `p` is made to point to `k`

Picturing pointers

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

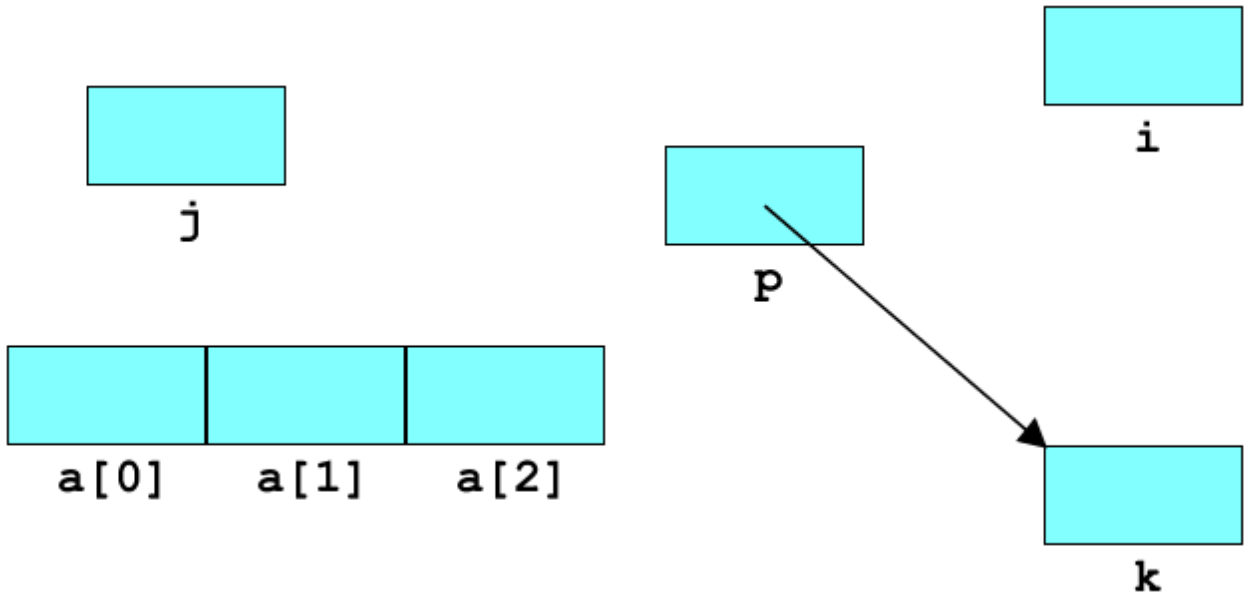


We don't know how memory is allocated, so the picture of p pointing to k could equally well be



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 `a[1]` then `p+1` points to `a[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

pointer.c

```
/* Print a pointer. */
#include <stdio.h>

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

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

```c
/* Print a value. */
#include <stdio.h>

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

NULL

One special pointer is provided in C, called `NULL`, available from `stdio.h` for example

Don't confuse it with the null character, written `'\0'`, which is only one byte long

The `NULL` pointer is guaranteed to be unusable (it points to location `0` which belongs to the OS)

It is used for uninitialised pointers, as an error indicator for functions that return pointers, and so on

Deliberate segfault

Here's a deliberate segfault:

```c
/* Demo: cause a segfault */
#include <stdio.h>

int main() {
    // Point to the beginning of the memory
    char *s = NULL;
    // Demonstrate that it doesn't belong to us
    s[0] = 'x';
}
```