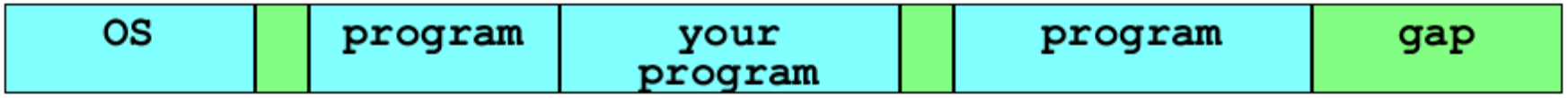
Memory

Programs in memory

The layout of memory is roughly:



Virtual memory means that memory is allocated in pages or *segments*, accessed ***as if*** adjacent - the platform looks after this, so your program doesn't have to

If you try to access memory not belonging to ***your*** program, you get a segmentation fault (segfault)

Zooming in

The layout of your program is roughly:

The code and constants are loaded when the program is run, then the heap expands upwards, the stack expands downwards

The three parts are usually separate segments, with the code and constants being read-only

The stack

The stack is where local variables are allocated during function calls

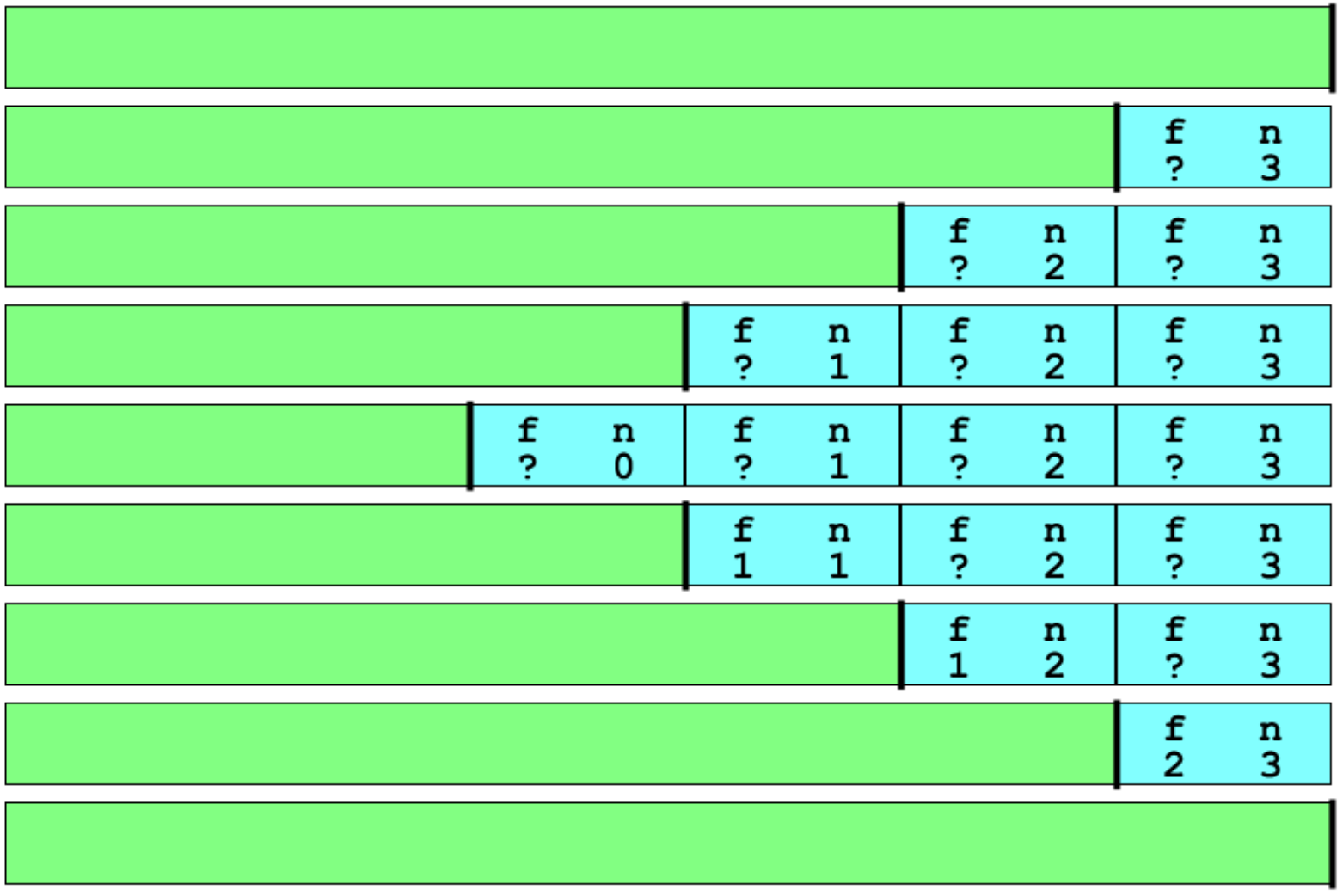
New space is allocated on entry to a function, then discarded on exit

This allows functions to be recursive, e.g.

```
int factorial(int n) {
    if (n == 0) return 1;
    int f = factorial(n - 1);
    return n * f;
}
```

Calls

Here's the stack during `factorial(3)` (simplified)



Normal calls

The example showed a recursive function

But the same sort of thing happens with normal functions

Memory is allocated for `main`, then for `sort` (say), then for `compare`, then `compare` returns and `sort` calls `swap`, then `sort` repeatedly makes similar calls, then returns, then maybe `main` calls something else, then eventually returns

The stack grows and shrinks 'at random' as functions are called and return, until eventually `main` returns

The heap

The heap is used for dynamically allocated memory, i.e. for items which can't be handled by function-call-based nested lifetimes

The most common case is an array or any other data structure which needs to grow when new data arrives

The heap is managed by the `malloc` and `free` library functions

malloc and free

The library functions `malloc` ("memory allocate") and `free` allocate and deallocate memory blocks

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

int main() {
    char *s = malloc(4);  // was char s[4];
    strcpy(s, "cat");
    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 standard libraries like `stdlib` and `stdio` is linked automatically by the compiler, but other libraries may need to be mentioned explicitly

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

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

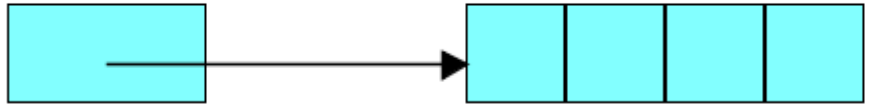
The return type of `malloc` is `void *` which means "pointer to something", compatible with all pointer types

Visualising malloc

You need to visualise the effect of malloc

```
char *s;
```


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



Before the call, `s` is random rubbish

After the call, `s` is a pointer to some new space

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 you should `free` all `malloc`ed memory, to avoid memory leaks, and the advanced debugging option `-fsanitize=address` will make sure you do

Indexing

The new memory is indexed like an array

```
s[0] = 'c';
strcpy(s, "cat");
```

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, but they are treated the same by the compiler when it comes to indexing

The heap

Here's the heap after some `malloc` and `free` calls

The heap never shrinks, but gaps appear after `free`

`malloc` searches for the best gap, `free` merges gaps, and both use a header, not shown, at the start of allocations and gaps to keep track of everything

So, they can be a bit expensive, but there are further details which reduce the cost

What's wrong?

Why is this not a good thing to do?

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

What's wrong?

Why is this not a good thing to do?

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

The pointer `s` is updated to point to the constant string, so it no longer points to the allocated memory

The allocated memory will remain allocated but unused, i.e. wasted, for the rest of the program

Allocating an array

Suppose you want an array of `10` integers:

```
int *numbers = malloc(10 * sizeof(int));
```

Don't forget to multiply by the size of the things you are allocating

calloc

There is an alternative function calloc

```
int *numbers = calloc(10, sizeof(int));
```

One difference is trivial (comma instead of `*`)

The other is that the memory is ***cleared*** (set to zero)

Some textbooks, tutorials, lecturers use `calloc` all the time, but (a) clearing the memory is inefficient if you are about to initialise it yourself, and (b) it might give beginners the mistaken idea that variables in C are always zeroed

Reallocation

How do you change the capacity of an array?

```
char *array = malloc(8);
int capacity = 8;
...
capacity = capacity * 3 / 2;
array = realloc(array, capacity);
```

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

The pointer changes, so `array` needs to be updated

Reallocation efficiency

The `realloc` function sounds costly (searching for a new gap and copying the old array into it)

But there are two circumstances where it is cheap

If the old array is at the end of the heap, `realloc` can just make it bigger without moving it

If the array is large, `realloc` uses a separate virtual memory segment for it, to avoid any further copying costs

Strategy

Suppose arrays increase in size (using `realloc`) when they run out of space

What size should they start at, and how much should their sizes be increased by?

Generally, ***start small*** (24 bytes) so lots of empty arrays aren't space-inefficient

And ***multiply the size*** (by 1.5) so that copying large arrays isn't time-inefficient

(Multiplying by 2 may prevent merging old arrays to store a new one)

Structures

Before, we did this:

```
struct bird ...;
typedef struct bird bird;

int main() {
    bird jaydata = { 41, 37 };
    bird *jay = &jaydata;
    ...
}
```

But there are problems if we don't know in advance how many birds we are going to want

Allocating structures

Instead we can now do this

```
bird *newBird(int x0, int y0) {
    bird *b = malloc(sizeof(bird));
    b->x = x0;
    b->y = y0;
    return b;
}

int main() {
    bird *jay = newBird(41, 37);
    ...
}
```

Initialising structures

To initialise more compactly, we can do this:

```
bird *newBird(int x0, int y0) {
    bird *b = malloc(sizeof(struct bird));
    *b = (bird) {x0, y0};
    return b;
}
```

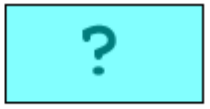
Or this:

```
...
    *b = (bird) {.x = x0, .y = y0};
...
```

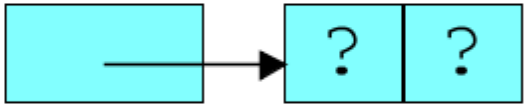
Visualisation

Visualising the memory during `newBird`:

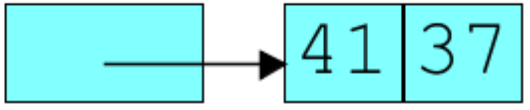
```
struct bird *b;
```




```
b = malloc(...);
```




```
*b = ...{x0,y0};
```



Word counting

Before, we did this:

```
struct word {
    char s[10];
    int count;
};
typedef struct word word;
```

The problem is that words have different lengths

Flexible array fields

Now, we can do this:

```
struct word {
    int count;
    char s[];
};
typedef struct word word;
```

The array field must go last in the structure, with no length specified, then it can have a variable length (stretching past the notional end of the structure)

Allocation

Here's how to allocate a flexible array:

```
struct word { int count; char s[]; };
typedef struct word word;

word *newWord(char *s) {
    int n = strlen(s) + 1;
    word *w = malloc(sizeof(word) + n);
    strcpy(w->s, s);
    w->count = 0;
    return w;
}
```

You allocate memory for the structure plus the array

Note this is a recent C feature

Lines

Suppose a program reads in a line

We might guess that this would be enough:

```
char line[1000];
```

But if a user feeds a line into our program which has been generated from some other program, this is probably not enough!

We've already seen that we can use `realloc` to increase the size of an array

Flexible array field?

So maybe we could write this:

```
struct line { int size; char s[]; };
typedef struct line line;

// Resize to make room for at least n characters
line *resize(line *l, int n) { ... }
```

But the pointer to the structure changes on resize, so this would have to be called with:

```
l = resize(l, n);
```

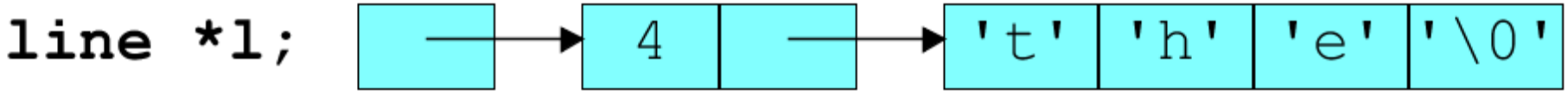
It is incredibly easy to forget the `l =` bit

Pointer field

The normal solution is to write this:

```
struct line { int size; char *s; };
...
void resize(line *l, int n) { ... }
```

Now there are two lumps of memory and two pointers



The structure pointer allows functions to update the fields in place, the array pointer makes sure the structure never moves, only the pointer field inside it

Binary compatibility

Many other languages are compiled to have 'binary compatibility' with C

That means they use the same conventions about code, heap, stack, and function calls, either for the whole language, or at least for the operating system service calls and cross-language calls

Stack details

The compiler uses the stack memory for each function call to store

- local variables, including arguments
- the return address in the code
- saved register contents from outer calls
- intermediate calculations that don't fit in registers

The result is that the exact layout of the stack is very much dependent on architecture and compiler choices, and can't easily be analysed by hand (hence `-g` option and `gdb` for debugging)

Arguments

Conventionally, arguments belong to the caller (calling function) rather than the callee (called function)

This allows variable-argument functions like `printf`

In retrospect, this was a bad design choice: it is illogical, and it prevents simple tail-call optimisations

It would have been better to generate special-case code for (fairly rare) variable-argument calls

But the issue isn't as simple as described here!

Two improvements

There are two common ways to improve programs where dynamic allocation efficiency is an issue

One is to use the `glib` library, which contains improved versions of `malloc` and `free`

Another is to allocate memory in large lumps, and implement a custom system for efficient high-turnover, small-object allocation within the lumps

Memory leaks

Since calling `free` is up to the programmer, even a correct program may gradually use up more and more memory unnecessarily

That's called a memory leak, and is an important potential flaw in long-running programs such as servers

Counter-measures are to use a library which deallocates automatically via garbage collection, or to use a library which detects leaks so they can be fixed, or use the new `-fsanitize=address` compiler option

Relocation

A program can be compiled into code which expects to be loaded at a particular location in memory

Alternatively, a program can be compiled into code plus extra information about the location-sensitive parts

The extra info allows the program to be relocated, i.e. loaded into different locations on different runs

Position independent code

A scheme which is much more elegant and flexible is for compiled machine code to be independent of where it is loaded in memory

Then relocation issues are avoided

It involves having an instruction set where jumps and calls are relative to the current location rather than absolute (e.g. "call the function 100 bytes further on from here")

Despite its clear superiority, this hasn't become normal

Linking

Even with position independent code, linking is necessary

This involves sorting out function calls (and other references) from one program component to another, e.g. calls to library functions

Static linking

With static linking, the parts of the library which are actually used by the program are copied into the program by the compiler

That way, the compiler can relocate the library code in advance, sort out all the function calls and other references between parts, and create a complete program which is ready to run

Dynamic linking

With dynamic linking, the library code is potentially shared between programs to save memory space

The compiler needs to know, somehow, where to expect the library to be in memory when the program runs

Then the program and the library are linked by the system when the program is loaded and prepared for execution

Shared libraries are called DLLs in Windows, and SOs on Linux/MacOs

The DLL approach

The approach taken by Windows is:

Compile a DLL library into code which is always at a fixed place in virtual memory

Then compile each program into fixed code which refers to the library code at its known location

When loading each program, arrange its virtual memory so that the virtual library location refers to the actual physical library location

A DLL problem

The DLL approach has a fundamental problem:

What happens if two independent DLL libraries have been compiled into the same place in virtual memory, and a program wants to use both?

The solution in Windows is (a) have a central authority for 'official' libraries which allocates locations and (b) if that fails, abandon sharing and copy one of the libraries into the program

For further problems, look up "DLL hell" in Google!

The SO approach

The "Shared Object" approach in Unix-based systems is:

Compile a program which uses an SO to retain relocation information about the library references

When loading the program, find the library location and complete the linking of the program by resolving the library references

This is slightly less efficient, but always works

SO problems

The SO approach still has administrative problems:

The compiler needs to know where the SO library file is, to find out what functions it makes available

The loader needs to know which SO the program needs, and where the SO library file is in case it needs to be loaded into memory for the first time

There are considerable potential problems with installation locations on disk, library versions, where to put the location information, and discrepencies between compile-time and load-time information

Shared library design

Shared libraries often have a monolithic design, making them unsuitable for static linking (because the whole library gets copied into the program)

The libraries are typically very big - programs only load quickly because the platform-specific libraries they use are already loaded into memory

If you port a program to another platform, it typically takes 20 seconds to load, because the shared libraries have to be loaded as well

So ***true*** cross-platform programming is very difficult

The future

Nobody knows the future, but these would be good:

- scrap virtual memory
- make all machine code position independent
- make data position independent (relative pointers)
- make pointers variable-sized
- ensure all machine code is validated in advance
- make all memory 'object oriented', even the stack
- provide hardware support for garbage collection
- make all platforms compatible