Today

So far:

data representations:

how are individual data elements represented in memory?

Now:

memory management:

how is the memory as a whole organised and managed?

memory management

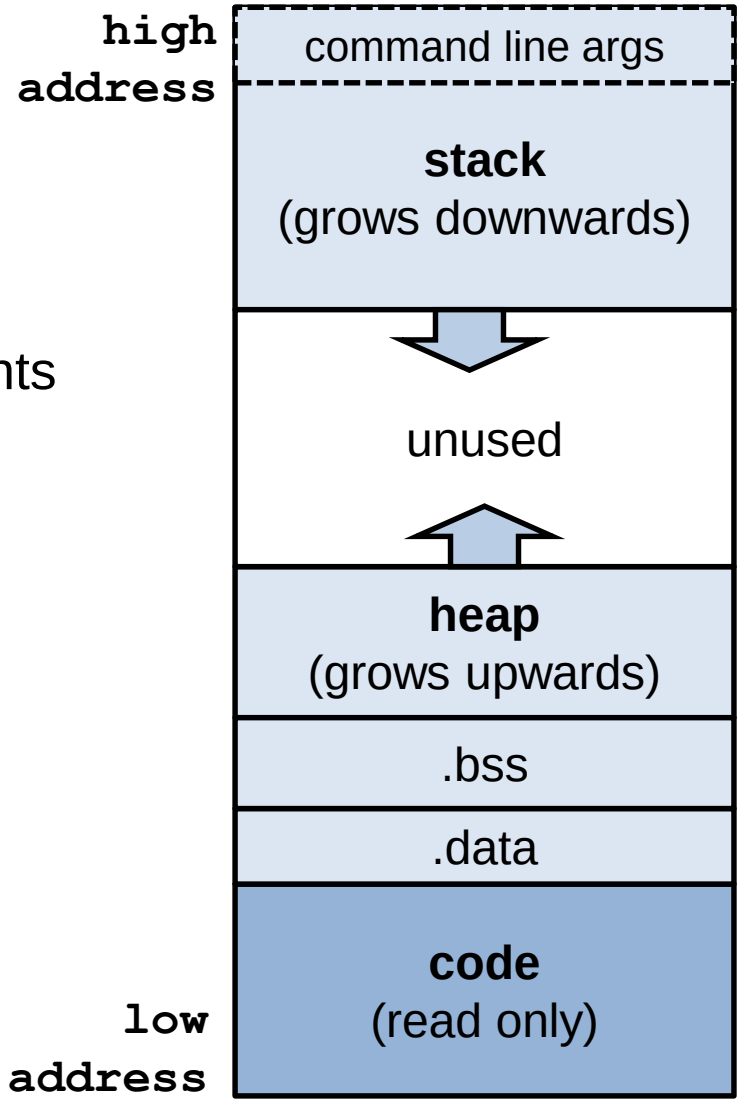
memory segments

The OS allocates memory for each *process* - ie. a running program – for data and code

This memory consists of different segments

- stack - for local variables
  - incl. command line arguments and environment variables
- heap - for dynamic memory
- data segment for
  - global uninitialised variables (.bss)
  - global initialised variables (.data)
- code segment

  typically read-only



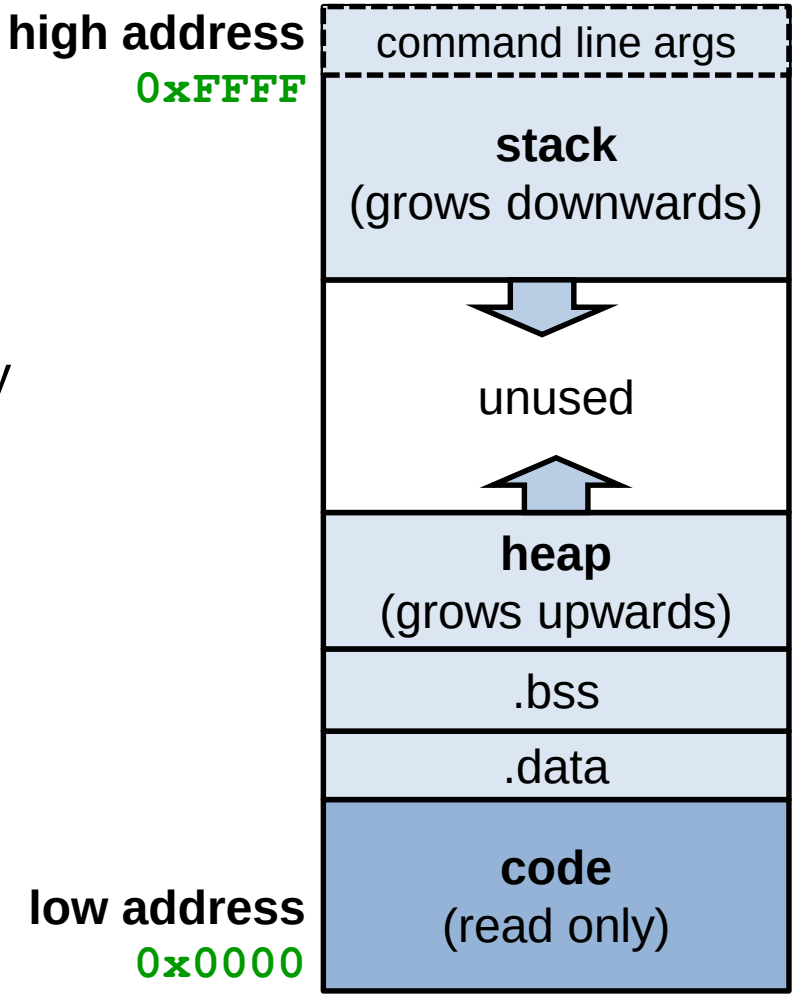
(Aside: real vs virtual memory)

Memory management depends on capabilities of

1. the hardware and
2. the operating system (OS)

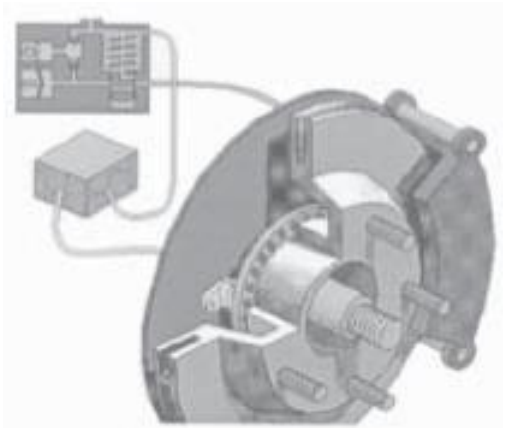
On primitive computers, which can only run a single process and have no real OS, the memory of the process may simply be all the *physical* memory

Eg, for an old 64K computer



(Aside: primitive computers)

These may only run a single process which then gets to use all of the memory



global variables

These are the easy ones for the compiler to deal with.

```
#include <stdio.h>
long n = 12345;
char *string = "hello world\n";
int a[256];
...
```

Here

- the global variables **`n`**, **`string`** and the string literal **`"hello world\n"`**, will be allocated in data
- The uninitialised global array **`a`** will be allocated in .bss

  The segment .bss is initialised to all zeroes. NB this is a rare case where C will do a default initialisation for the programmer!

memory segments

On Linux

```
> cat /proc/<pid>/maps
```

shows memory regions of process **<pid>**

With

```
> ps
```

you get a listing of all processes,
like the Taskbar in windows

(This is not exam material)

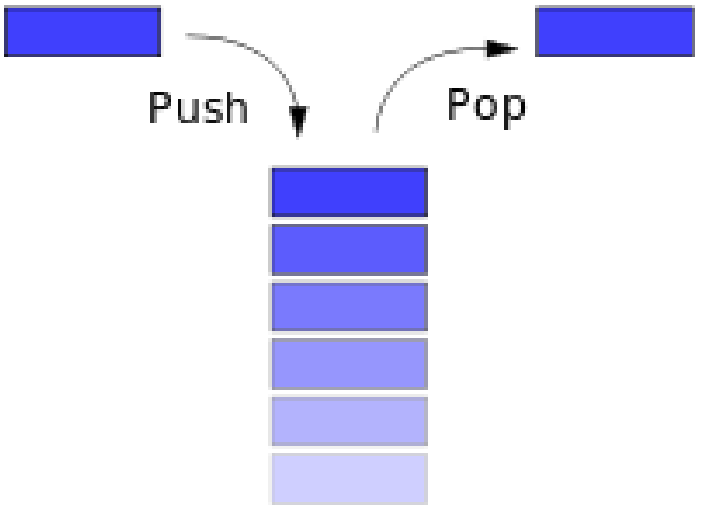
the stack

stack, pop, push

A stack (in Dutch: stapel) organises a set of elements
in a Last In, First Out (LIFO) manner

The three basic operations on a stack are

- pushing a new element on the stack
- popping an element from the stack
- checking if the stack is empty



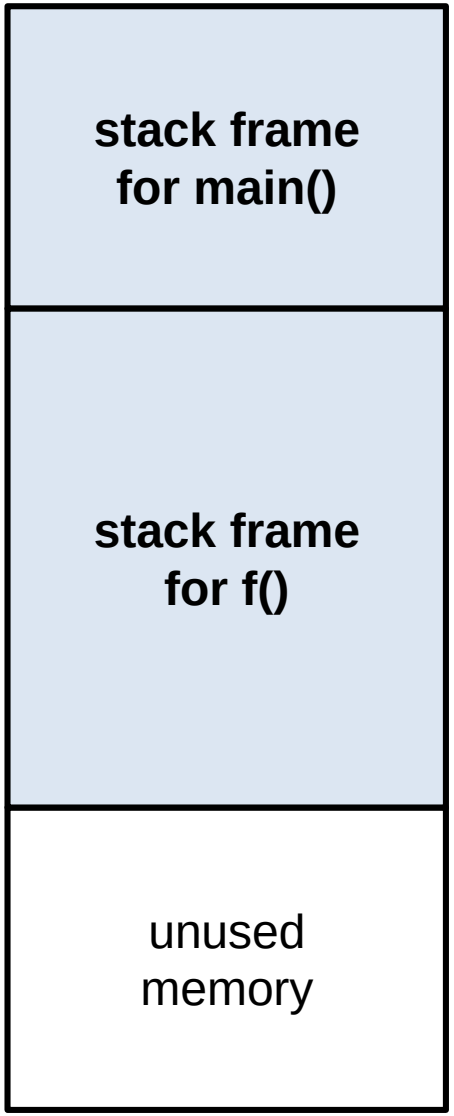
the stack

The stack consists of stack frames aka activation records, one for each function call,

- allocated when a function is called,
- de-allocated when it returns.

```
main(int i){
 char *msg =”hello”;
 f(msg);
}

int f(char *p){
  int j;
  ..;
  return 5;
}
```

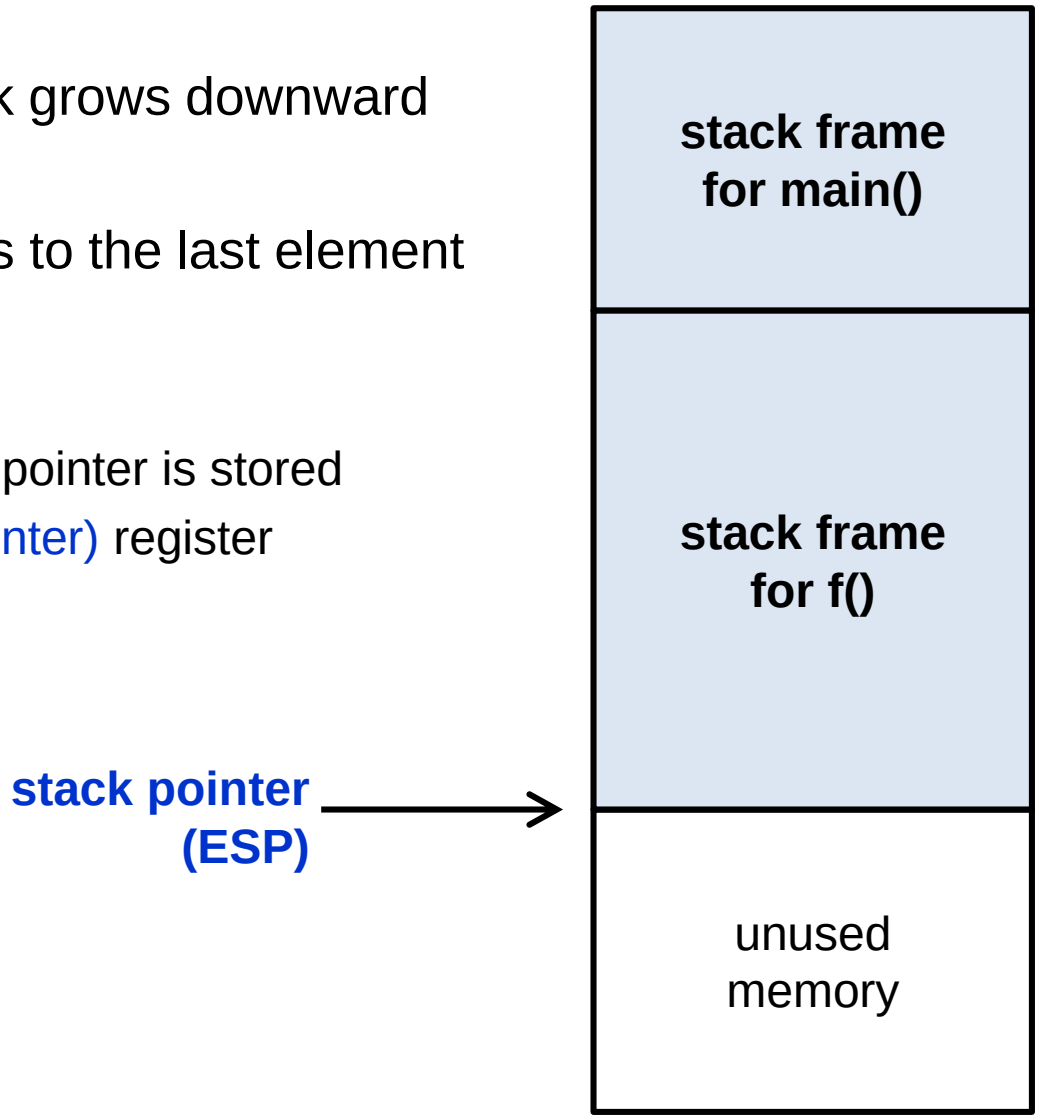


the stack

On most machines, the stack grows downward

The stack pointer (SP) points to the last element on the stack

On x86 architectures, the stack pointer is stored in the ESP (Extended Stack Pointer) register



the stack

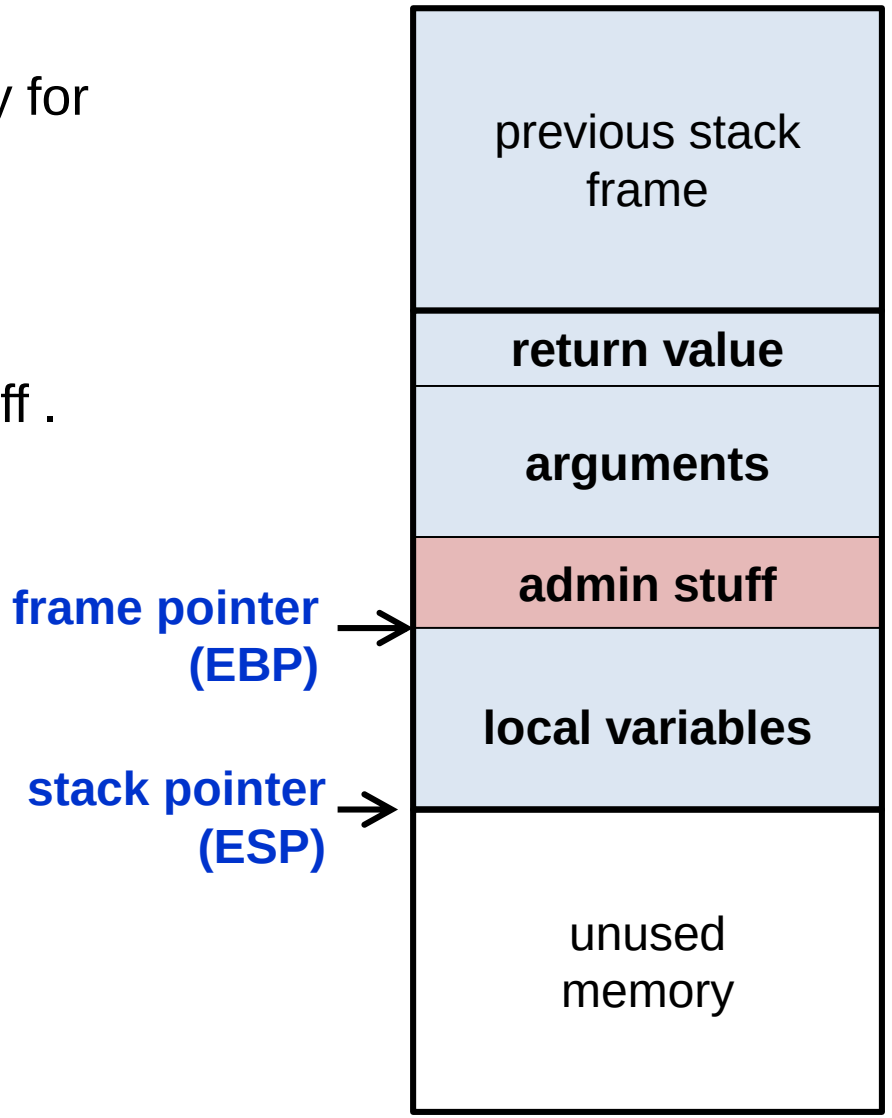
Each stack frame provides memory for

- arguments
- the return value
- local variables

of a function, plus some admin stuff .

The frame pointer provides a starting point to locate the local variables, using offsets.

On x86 architectures, it is stored in the EBP (Extended Base Pointer) register



the stack

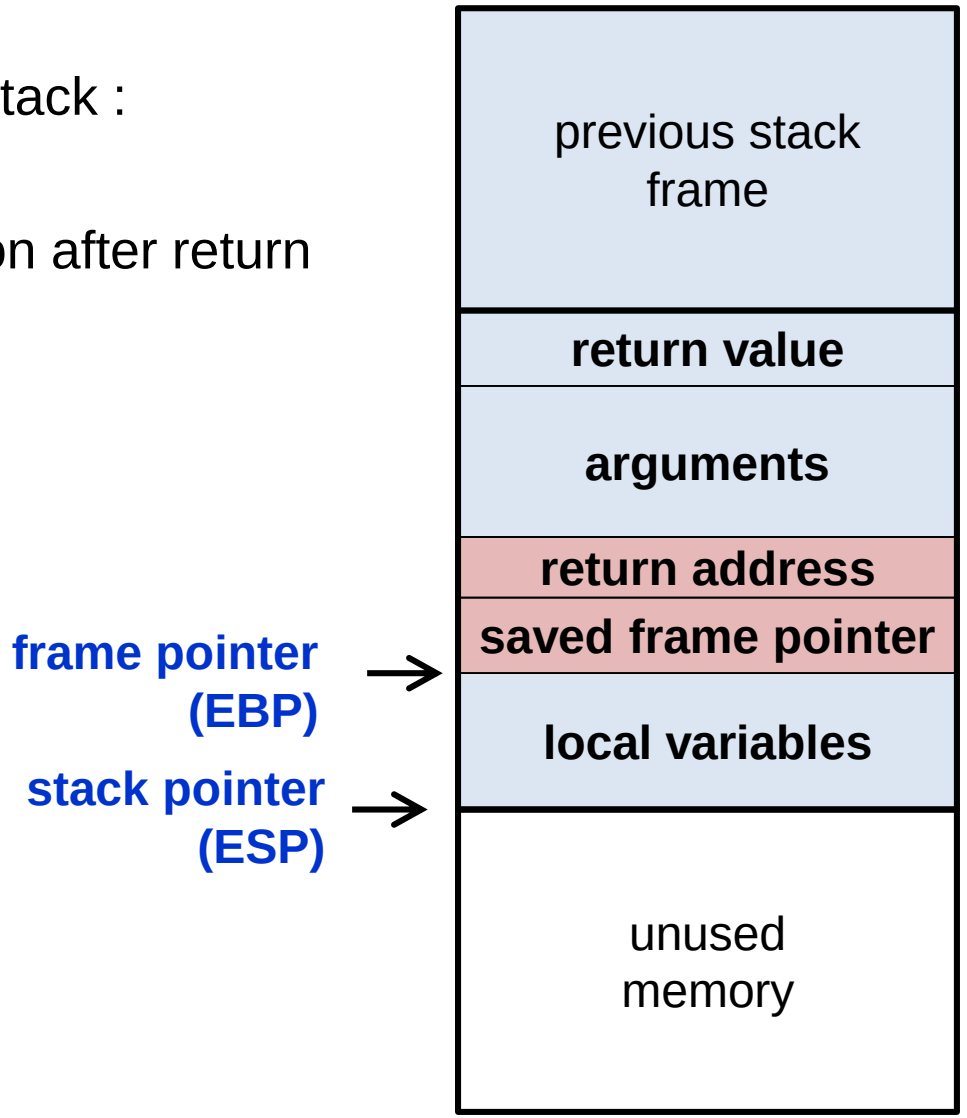
The admin stuff stored on the stack :

- return address

  ie where to resume execution after return

- previous frame pointer

  to locate previous frame

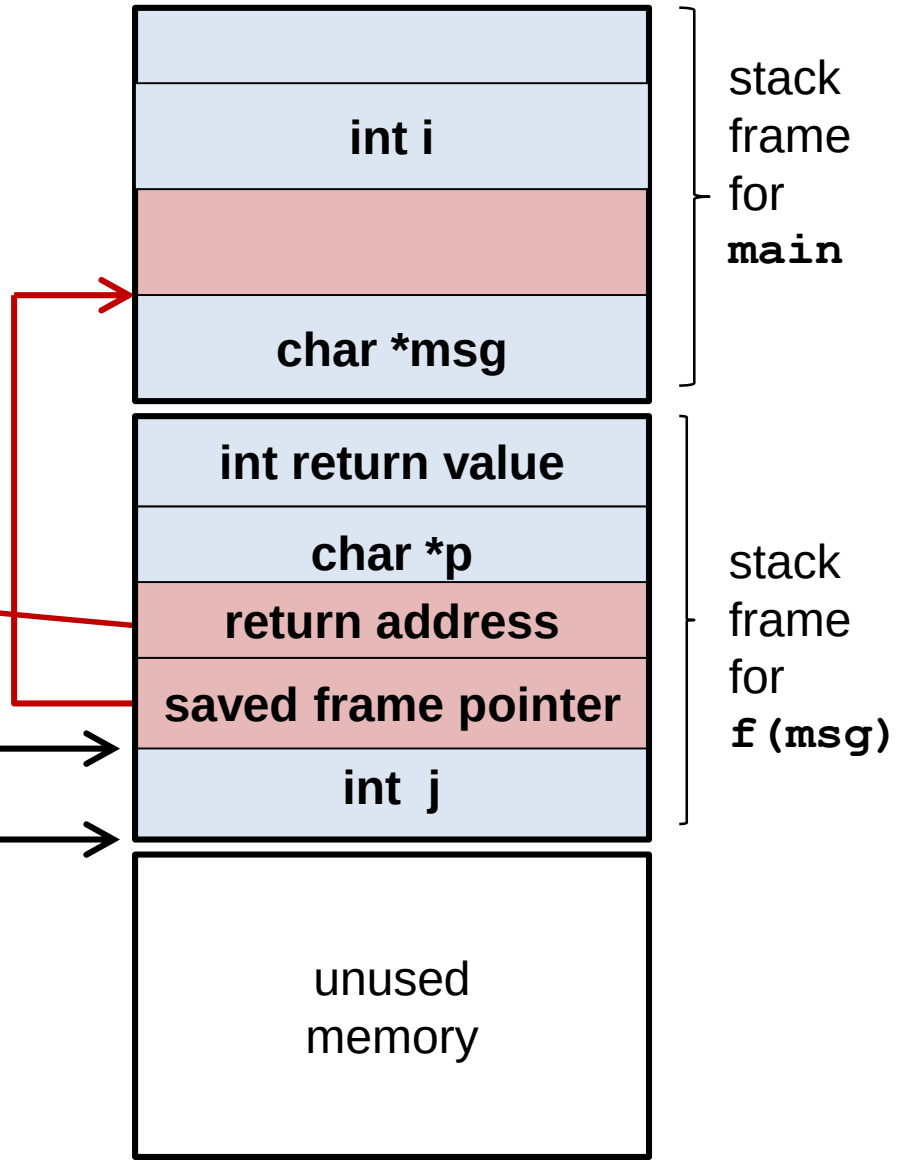


the stack

Stack during call to f

```
main(int i){
 char *msg =”hello”;
 f(msg);
}
```

```
int f(char *p){
  int j;
  ..;
  return 5;
}
```



function calls

- When a function is called, a new stack frame is created
  - arguments are stored on the stack
  - current frame pointer and return address are recorded
  - memory for local variables is allocated
  - stack pointer is adjusted

- When a function returns, the top stack frame is removed
  - old frame pointer and return address are restored
  - stack pointer is adjusted
  - the caller can find the return value, if there is one, on top of the stack

- Because of recursion, there may be multiple frames for the same function on the stack
- Note that the variables that are stored in the current stack frame are precisely the variables that are in scope

security worries

- There is no default initialisation for stack variables
  - by reading unitialised local variables,
    you can read memory content used in earlier function calls

- There is only finite stack space
  - a function call may fail because there is no more memory

  In highly safety- or security-critical code, you may want to ensure that this cannot happen, or handle it in a safe way when it does.

- The stack mixes program data and control data
  - by overrunning buffers on the stack we can corrupt the return addresses!

  More on that the next weeks!

(Aside: hardware-specific details)

- The precise organisation of the stack depends on the machine architecture of the CPU

- Instead of storing data on the stack (in RAM)
  some data may be stored in a register (*in* the CPU)

Eg, for efficiency, the top values of the stack may be stored in CPU registers, or in the CPU cache, or the return value could be stored in a register instead of on the stack.

the heap

Limitations of the stack

```
int *table_of(int num, int len) {
  int table[len+1];
  for (int i=0; i <= len; i++) {
     table[i] = i*num;
  }
  return table; /* an int[] can be used as an int* */
}
```

What happens if we call the function above, with

```
int *table3 = table_of(3,10);
printf("5 times 3 is %i\n", *(table3+5));
                              // we use pointer arithmetic
                              // to mimic array indexing
```

Limitations of the stack

```
int *table_of(int num, int len) {
  int table[len+1];
  for (int i=0; i <= len; i++) {
     table[i] = i*num;
  }
  return table; /* an int[] can be treated as an int* */
}
```

What happens if we call the function above, with

```
int *table3 = table_of(3,10);
printf("5 times 3 is %i\n", *(table3+5));
int *table4 = table_of(4,10);
printf("5 times 4 is %i\n", *(table4+5));
printf("5 times 3 is %i\n", *(table3+5));
```

The function **`table_of`** is weird, because it returns a reference to a local variable **`table,`** but the memory for this variable is de-allocated when the function returns...

NB you should never write such code, and any decent compiler will warn about this. Still, it is legal C...

A cleaner solution would be to let the caller allocate the memory, and pass in a pointer to that

```
void table_of(int num, int len, int[] table)
```

but note this only works because we know the size of the table needed beforehand.

This is a general limitation of stack-allocated memory:

how can a function allocate some memory that can later still be used by the caller?

the heap – for dynamic memory

(De)allocation of stack memory is very fast, but has its limitations:

- any data we allocate in a function is gone when it returns

Solution: the heap

- The heap is a large piece of scrap paper where functions can create data that will outlive the current function call.
- Functions can use it to share values, using pointers to data stored on the heap
- It is up to the program(er) to organise this; the OS will only keep track of which part of the scrap paper is still unused

malloc

The operation to allocate a chunk of memory on the heap is **malloc**

```
#include <stdlib>
 void* malloc (size_t size)
                // size_t is an unsigned integral type
```

returns an pointer to a contiguous block in memory of **size** bytes,
or NULL if an error occurs

Example use

```
int *table = malloc(len*sizeof(int));
 // allocates enough memory for len int's
```

void* ?

Recall that pointers are typed,

eg **`int*`** is the type of pointers to an **`int`**.

`void*` is the type of untyped pointers.

`malloc` just returns a pointer to a blob of memory, and the result does not have any specific type (yet), so its return type is **`void*`**

A **`void*`** pointer can be converted into any other pointer type, without an explicit cast.

NULL ?

- **`NULL`** is a special value, which is guaranteed to be different from any legal address

  So **`&p`** will never return **`NULL`** for any properly allocated variable **`p`**

- Dereferencing a null pointer, eg

```
int* ptr = NULL;
int i = *ptr;
```

  leads to undefined behaviour:

  the program probably crashes, but basically anything can happen.

  So you should never dereference a **`NULL`** pointer

Check for malloc failure!

Malloc may fail, namely if there is not enough heap space available, in which case it returns **`NULL.`**

Programs should always check the result of malloc!!!

Eg directly after

```
int *table = malloc(len*sizeof(int));
```

there should be a line like

```
if (table == NULL) { exit();}
```

or

```
if (!table) { return -1;}
      // NULL is interpreted as false,
      // so !table will be true when table is NULL.
      // Writing table==NULL is probably clearer?
```

free

You, the programmer, are in charge of freeing heap memory that is no longer needed, by calling

```
void free (void* p)
```

Normal usage pattern

```
long *p = (long*) malloc (10*sizeof(long));
if (p == NULL) { exit();}
... // use p
free(p); // when p is no longer needed
```

Here **`free`** and **`malloc`** can be in different functions