Memory Management
a **C** view

Dr Alun Moon

Computer Science

KF5010

The Von Neumann model

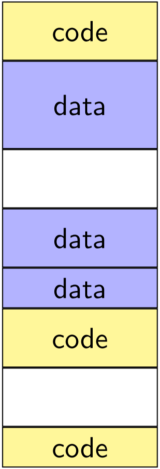
Memory Architecture

- One continuous address space
- Program code and data can occupy any space
- Code and Data are indistinguishable

In the CPU

Program Counter
: holds the address of the next instruction to fetch

Address Register
: holds the address of memory to read/write data

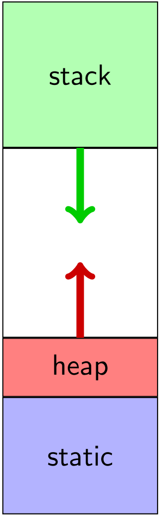


The Program Model

View from a single process

There are three areas of memory of interest to the program

- **Static** memory is fixed, allocated at compile time.
- **Stack** memory is fluid used at runtime, critical for functions, parameters and local variables
- **Heap** memory is dynamic, requested and freed by the program as needed.



Static Memory

Memory is put aside (allocated) at Compile time

- Code
- Global variables
- static variables

Example

```
int life;

int foo()
{
   static int bar ;
}
```

Static Memory

Memory is put aside (allocated) at Compile time

- Code
- Global variables
- static variables

Example

```
int life;

int foo()
{
    static int bar ;
}
```

Static Memory

Memory is put aside (allocated) at Compile time

- Code
- Global variables
- static variables

Example

```
int life;

int foo()
{
   static int bar ;
}
```

Static Memory

Memory is put aside (allocated) at Compile time

- Code
- Global variables
- static variables

Example

```
int life;

int foo()
{
    static int bar ;
}
```

Heap

The heap is a specialist area making use of `malloc` and `free`.

Beware, Here be Dragons!

This can be very error prone.

It allows for *Dynamic Allocation* of memory as needed and requested by the program.

Stack

Run time memory, grows and shrinks as needed

- The most heavily used section of memory
- Invisible to the programmer

Stack

Run time memory, grows and shrinks as needed

- The most heavily used section of memory
- Invisible to the programmer

Used for:

- Function calls

Stack

Run time memory, grows and shrinks as needed

- The most heavily used section of memory
- Invisible to the programmer

Used for:

- Function calls
- Function parameters

Stack

Run time memory, grows and shrinks as needed

- The most heavily used section of memory
- Invisible to the programmer

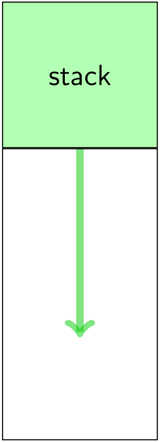
Used for:

- Function calls
- Function parameters
- local variables

```
int foo(int bar)
{
    int baz ;

    return 4 ;
}

main()
{
    foo(6);
}
```

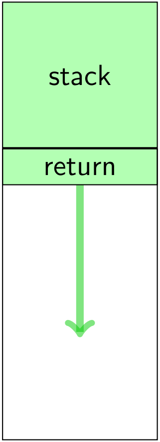


```
int foo(int bar)
{
    int baz ;

    return 4 ;
}

main()
{
    foo(6);
}
```

1. return address pushed onto stack

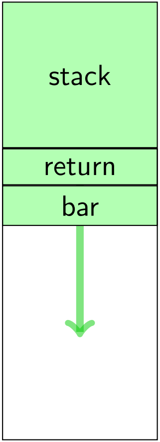


```
int foo(int bar)
{
    int baz ;

    return 4 ;
}

main()
{
    foo(6);
}
```

1. return address pushed onto stack
2. parameters pushed onto stack

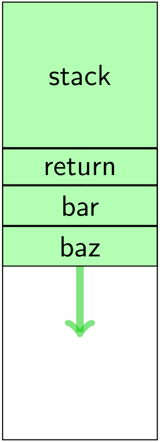


```
int foo(int bar)
{
    int baz ;

    return 4 ;
}

main()
{
    foo(6);
}
```

1. return address pushed onto stack
2. parameters pushed onto stack
3. local variables created on stack

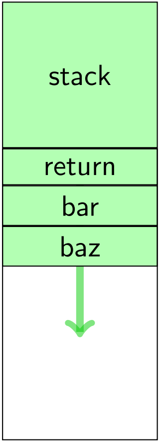


```
int foo(int bar)
{
    int baz ;

    return 4 ;
}

main()
{
    foo(6);
}
```

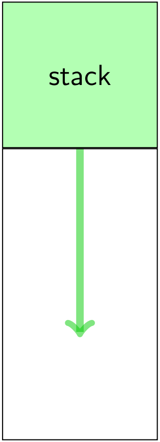
1. return address pushed onto stack
2. parameters pushed onto stack
3. local variables created on stack



```
int foo(int bar)
{
    int baz ;

    return 4 ;
}

main()
{
    foo(6);
}
```



in C

Pointers hold the address of things in a C program.

Example

```
int *p;
int n;

p = &n;
n = *p;
```

integer

in C

Pointers hold the address of things in a C program.

Example

```
int *p;
int n;

p = &n;
n = *p;
```

pointer to integer

Parameters are copied by value

You can't alter the value of parameters and change the value outside the function.

Wrong

```
void swap (int x, int y)
{
  int t = x;
  x = y;
  y = t;
}
```

```
swap(a,b);
```

Right

```
void swap (int *x, int *y)
{
  int t = *x;
  *x = *y;
  *y = t;
}
```

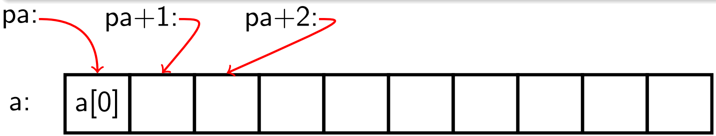
```
swap( &a, &b);
```

C is *Pass by Value*

Arrays

```
int a[10];
```


```
int *pa = &p[0];
```



array	**pointer**	
a[0]	*(pa+0) or *pa	pa[0]
a[1]	*(pa+1)	pa[1]
a[9]	*(pa+9)	pa[9]

Arrays and Pointers

- The relationship between arrays and pointers is defined.
- If you have a pointer `pa` and an array `a`
- Iteration
  - `a[i++]`
  - `*pa++`
- As parameters
  - `int f(int n[])`
  - `int f(int *n)`

  calling
  - `f(a)`
  - `f(pa)`

Note:

- A pointer is a variable and can be changed `pa+=2`.
- An array name is a constant and cannot be altered `a+=3` .

```c
#include <stdio.h>
void g(void *p)
{
  printf("     p is %p at %p\n",p,&p);
}

int f(int n[])
{
  int a[2];
  printf("  n is %p at %p\n",n,&n);
  printf("  a is at %p \n", a);
  g(n);
}

int b[10];

int main(int argc, char *argv[] )
{
  int a[10];
  printf("I am at   %p\n", (void *)main);
  printf("f() is at %p\n", (void *)f);
  printf("g() is at %p\n", (void *)g);
  printf("I have %d argument(s)\n", argc);
  printf("paramters at %p and %p\n", &argc, &argv);
  printf("My name is \"%s\"\n", argv[0]);
  printf("array a is at %p\n",a);
  printf("array b is at %p\n",b);
  f(a);
  f(b);
}
```

```
I am at   :00401792
f() is at :00401752
g() is at :00401730
I have 1 argument(s)
paramters at 0022FF00 and 0022FF04
My name is "iamat"
array a is at 0022FEC8
array b is at 004053E0
  n is 0022FEC8 at 0022FEB0
  a is at 0022FE98
     p is 0022FEC8 at 0022FE80
  n is 004053E0 at 0022FEB0
  a is at 0022FE98
     p is 004053E0 at 0022FE80
```

Functions and Pointers

A similar relationship between pointers and functions exists.

The function name is a constant holding the address of the function

A pointer to a function can be defined

- it can be assigned to
- it can be passed as a parameter
- the function can be called

In the standard library see these examples for some uses (`stdlib`)

- atexit
- qsort
- bsearch

Given

```
typedef void (*isr)() isr_t;

isr_t handler;

void dothis()
{
}

handler = dothis ;
```

The function stored in the variable can be called using function notation (*c.f.* array usage)

```
handler();
```

Function pointers

atexit

```
#include <stdlib.h>

int atexit(void (*function)(void));
```

Here `function` is a pointer to a function that has no parameters (`void`) and returns no value (`void`)

example

```
void cleanup(void) {...}

int main() {
    atexit( cleanup );
    .
    .
    exit(EXIT_SUCCESS);
}
```

- `atexit` registers the function `cleanup`
- when the program finishes via exit or in the result of a signal
- the `cleanup` function is called before the program finally finishes.

Function pointers

pthreads

```
void cleanup(void) {...}

int main() {
    atexit( cleanup );
    .
    .
    exit(EXIT_SUCCESS);
}
```

`start_routine` is a pointer to a function, that takes a generic pointer (`void*`) as an argument and returns a generic pointer (`void*`)
This function does the work of the thread.

Part II

`malloc`

Dynamic Memory Allocation
The Heap

The Heap is an area of Operating-System managed memory, that programs can request chunks of.

- `malloc` requests a number of bytes
- `calloc` requests memory for an array of *n* things of a given size

```
(void*)
```

Size of things in C – Memory used

In performing operations using `malloc` and friends, it is necessary to know how many bytes things are, to request memory
C provides a useful operator and type for these activities

The `sizeof` operator and `size_t` type

The `size_t` is an unsigned integer type, used where a size, or count is needed.
The `sizeof` operator gives the size in bytes (a `char` is guaranteed to be 1) of an object, a variable, and array, or a type.

sizeof and size_t

examples

`char`

sizeof

Compile time operator

The sizeof operator measures storage in units of char

sizeof variable returns the number of bytes that the variable needs for its storage.

sizeof array returns the total number of bytes allocated for the arrays storage.
The number of elements in an array can be found by
(void*)

```
void cleanup(void) {...}

int main() {
    atexit( cleanup );
    .
    .
    exit(EXIT_SUCCESS);
}
```

returns the ammount of storage that the (type) needs. The type has to be in parenthesis

malloc

malloc is used to dynamically allocate blocks of memory.

- it returns a generic pointer to a block of memory of the requested size.
- it returns NULL if the request cannot be satisfied.
- the pointer needs casting into an appropriate type.

```
void *malloc(size_t size);
```

```
struct element *data;
data = (struct element *)malloc( sizeof(struct element) );
if( data ) { /* populate data */
    data->number = 12;
}
```

calloc

malloc is used to dynamically allocate blocks of memory, for use as arrays.

- it returns a generic pointer to a block of memory to hold the required number of elements of the given size.
- it returns NULL if the request cannot be satisfied.
- the pointer needs casting into an appropriate type.

```
typedef struct element atom;
atom *array;
array = (atom*)calloc( 12, sizeof(atom));
if( array ) { /* populate data */
    for( n=0 ; n<12 ; n++ ) {
        array[0]->munber = 2+n;
    }
}
```

Finished – tidy up after yourself

```
void free(void *ptr);
```

Once you have finished using memory obtained using `malloc` and `calloc`. It is good practice to release the memory back to the Operating-System using `free`

Beware...

Programmers who are sloppy about keeping track of dynamic memory can create memory leaks

```
void function(void) {
    int *d = (int*)calloc(100,sizeof(int));

    free(d);
}
```