Memory Management

a **C** view

Dr Alun Moon

Computing, Engineering and Information Sciences

26th October 2011

The Von Neumann model

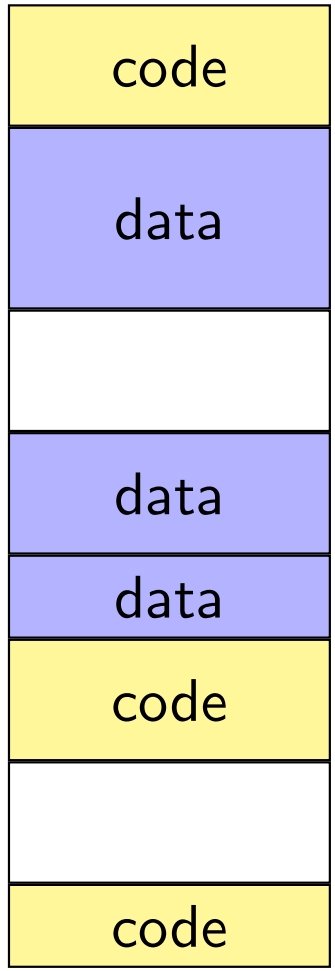
Memory Architecture

- 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

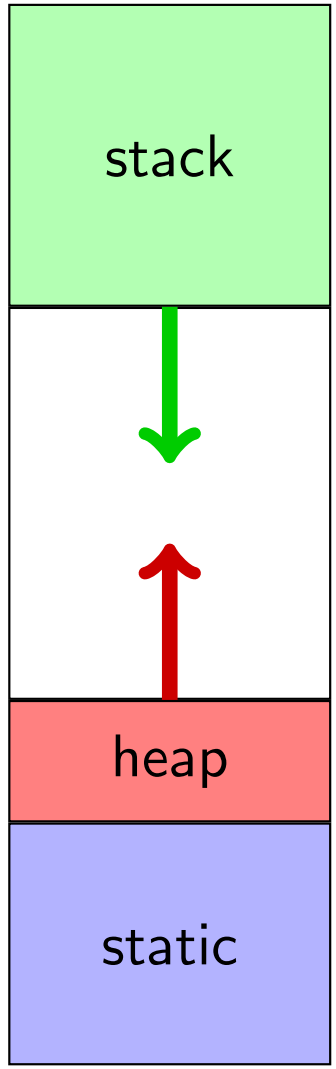
View from a single process

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`.

Best stayed away from for now!

This can be very error prone.

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

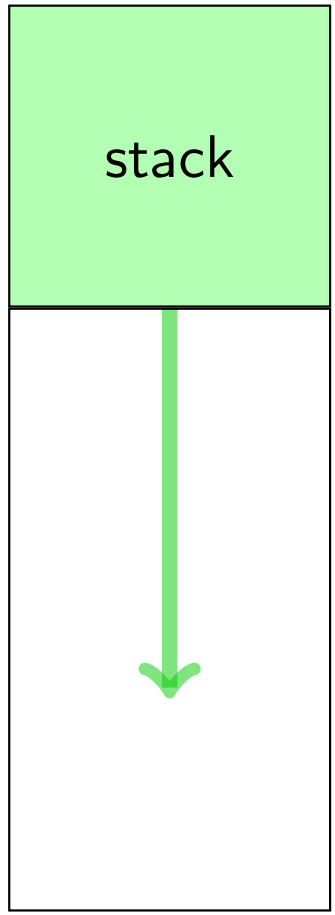
Used for:

- Function calls
- Function parameters
- local variables

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

    return 4 ;
}

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

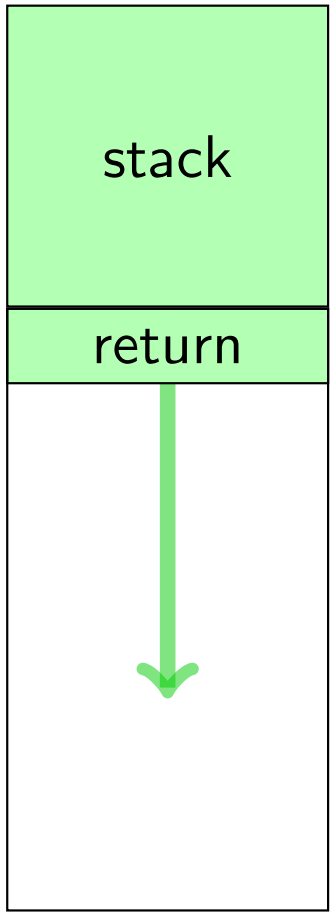


```
int foo(int bar)
{
    int baz ;

    return 4 ;
}

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

1. return address pushed onto stack

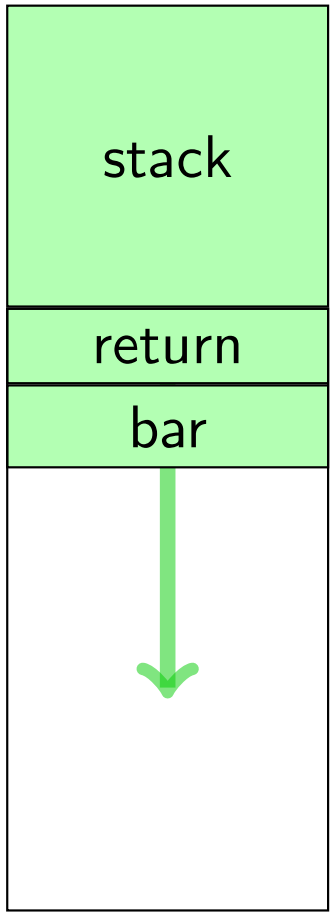


```
int foo(int bar)
{
    int baz ;

    return 4 ;
}

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

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

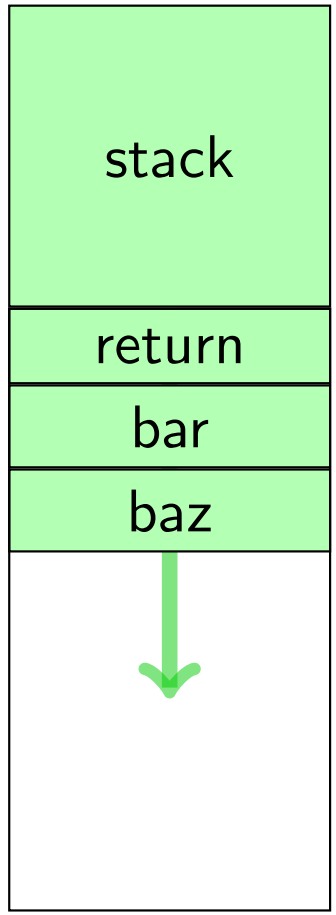


```
int foo(int bar)
{
    int baz ;

    return 4 ;
}

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

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

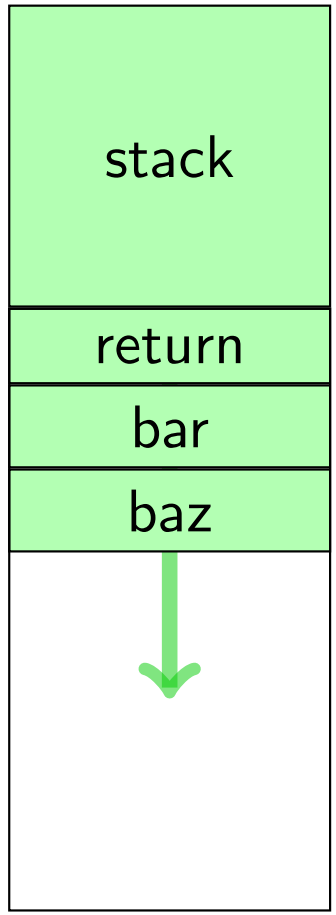


```
int foo(int bar)
{
    int baz ;

    return 4 ;
}

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

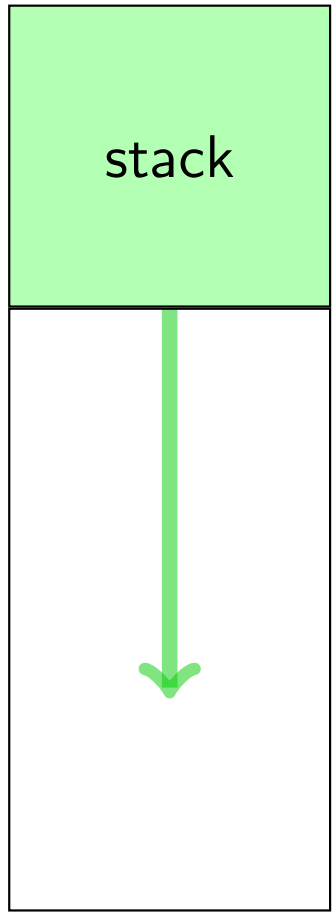
1. return address pushed onto stack
2. parameters pushed onto stack
3. local variables created on stack



```
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);
```

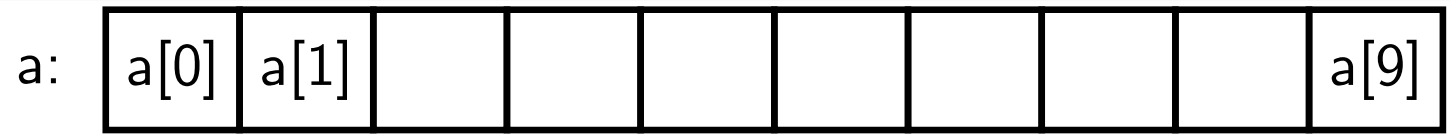
Right

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

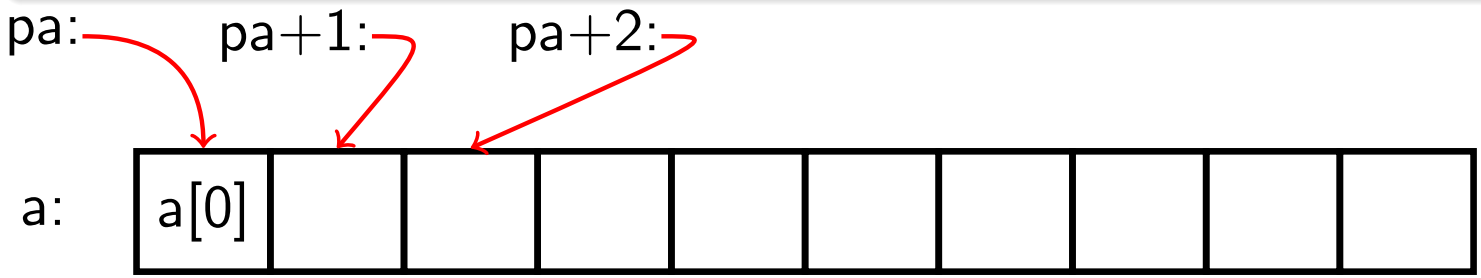
```
swap( &a, &b);
```

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();
```