C Programming
Functions and Pointers

Adam Sampson (based on material by Greg Michaelson)
School of Mathematical and Computer Sciences
Heriot-Watt University

C lectures

- Compiling code, program layout, printing/reading data, expressions, arithmetic, memory addresses, control flow, precedence
- **Functions, pointers, file IO, arrays**
- Memory allocation, casting, masking, shifting
- Strings, structures, dynamic space allocation, field access
- Recursive structures, 2D arrays, union types

Function declaration

```
type name(type1 name1, ..., typeN nameN)
{  declarations
   statements
}
```

- Result **type** is `void` if nothing is returned
- `name`*i* == **formal** parameters
- `{ ... }` == body
- C99: declarations may be interleaved with statements

Function declaration

- Formal parameters are optional
  - If you want your function to take no parameters, you must write `(void)` – empty brackets means “unspecified”
- `return` will exit the function immediately
  - Useful when something's gone wrong
- A function that doesn't return `void` **must** finish by returning a value, with:
  `return expression;`

Function prototype

- In C, you can only use a function (or variable, type...) **after** it has been declared
- So how do you write a pair of functions that call each other? Or call a function declared in a different file?
- A **prototype** is a function header with no body, e.g.
  `float diagonal_length(float a, float b);`
  - Tells the compiler that the function exists, and will be defined later
  - (With `extern` in front: will be defined in another file)
  - Normal to have prototypes at the top of a source file

Function call

`name(exp1, ..., expN);`

- `exp`*`i`* == **actual** parameters

- Evaluate `exp`*`i`* expressions in an arbitrary order
- Store `exp`*`i`* values in stack/registers
- Execute body, allocating stack/registers to any declarations
- Reclaim stack space for parameters and declarations
- Return result if any

`poly2.c`

Polynomial as a function

Polynomial as a function

```
#include <stdio.h>

int poly(int a, int b, int c, int x)
{
    return a * x * x + b * x + c;
}

int main(int argc, char *argv[])
{
    printf("%d\n", poly(2, 4, 3, 5));
    return 0;
}
```

Polynomial as a function

```
#include <stdio.h>

int poly(int a, int b, int c, int x)
{
    return a * x * x + b * x + c;
}

int main(int argc, char *argv[])
{
    printf("%d\n", poly(2, 4, 3, 5));
    return 0;
}
```

Formal parameters

Actual parameters

Changing parameters

- Parameters are always **passed by value** in C

- Value of the actual parameter is copied to space for the formal parameter
- Final value is **not** copied back from formal to actual
- ... so changing the formal does not change the actual

`swap.c`

Swapping parameters

Swap (does not work!)

```
#include <stdio.h>

void swap(int a, int b)
{
    int t;
    t = a;
    a = b;
    b = t;
}
```

Swap (does not work!)

```
int main(int argc, char *argv[])
{
    int x, y;
    x = 1;
    y = 2;
    swap(x, y);
    printf("x: %d, y: %d\n", x, y);
    return 0;
} →
```

x: 1, y: 2

Pointers

To declare a pointer variable:

```
type *name;
```

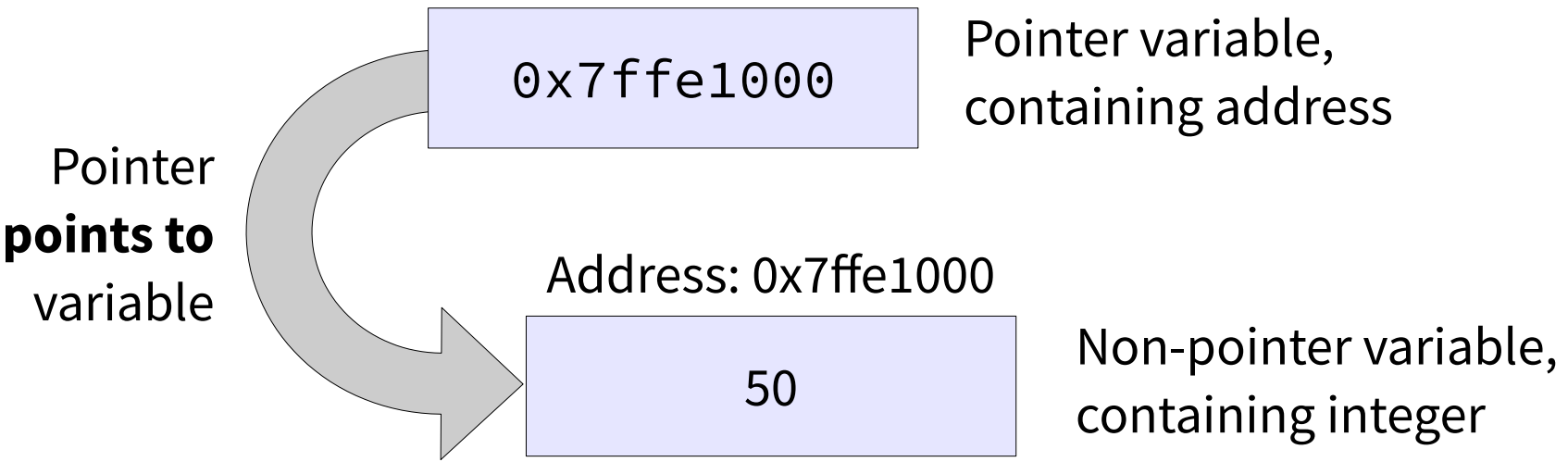
e.g.

```
int *i;
```

- Variable *name* holds **address** of data of type *type*
- Allocates stack space for the address
- ... but does **not** create instance of *type*

Pointers

- Pointer: a variable, whose value is the address of another variable



Code samples with this logo are for you to look at if you want another example – I won't go through them on the slides.

`pointer_to_int.c`

Pointer to an integer

Changing parameters

- To write a function that can change an actual parameter:
  - Pass the address of (&) the actual parameter
  - Make the formal parameter a pointer
  - Use indirect (*) when referring to the formal parameter

```
void doubler(int *x) { *x = *x * 2; }

int main(int argc, char *argv[]) {
    ...
    y = 1;
    doubler(&y);
    ...
```

`call_by_value.c`

Doubler – call by value

`call_by_reference.c`

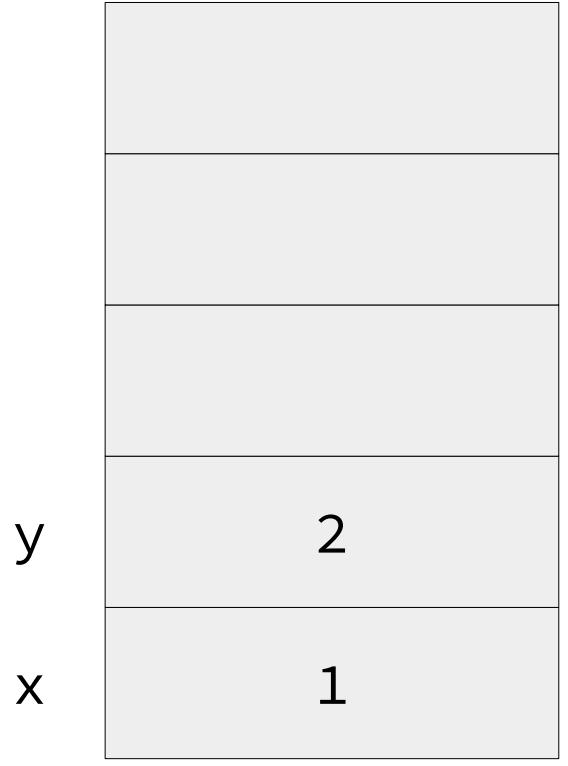
Doubler – call by reference

swap.c

Swap – call by value

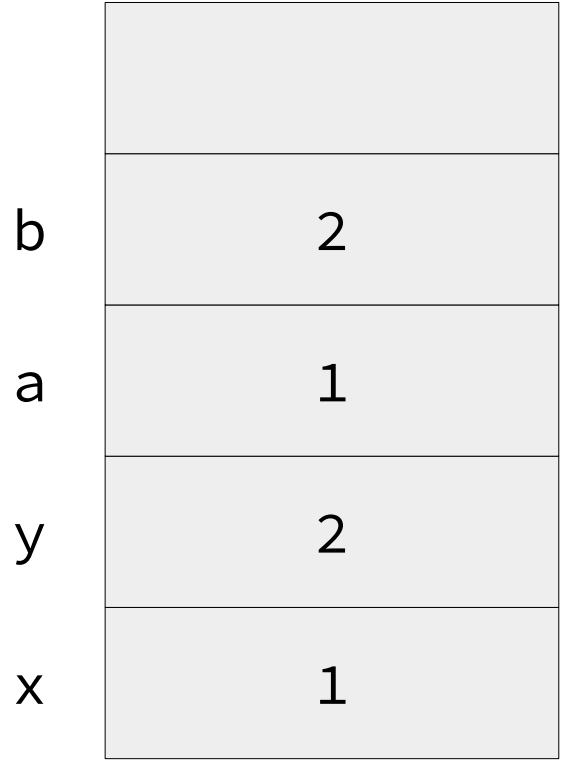
Swap – call by value

```
void swap(int a, int b)
{
    int t;
    t = a;
    a = b;
    b = t;
}
...
int x, y;
x = 1;
y = 2;
swap(x, y);
```



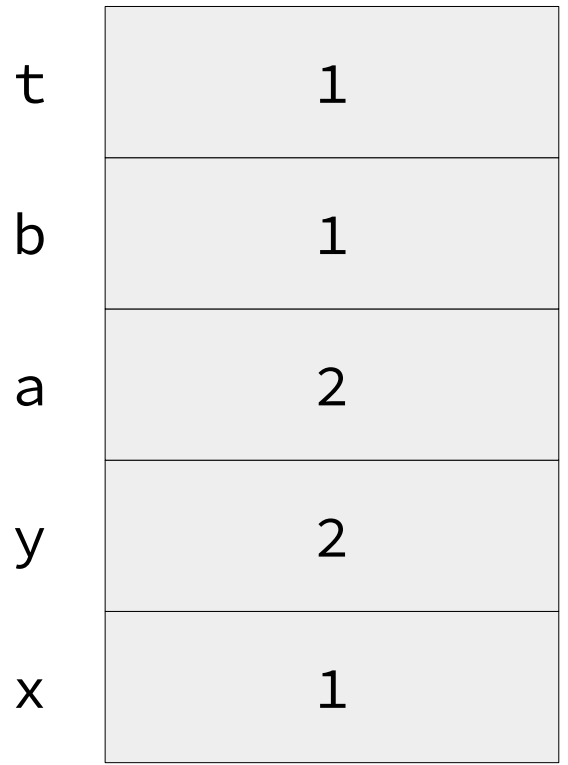
Swap – call by value

```
void swap(int a, int b)
{
    int t;
    t = a;
    a = b;
    b = t;
}
...
int x, y;
x = 1;
y = 2;
swap(x, y);
```



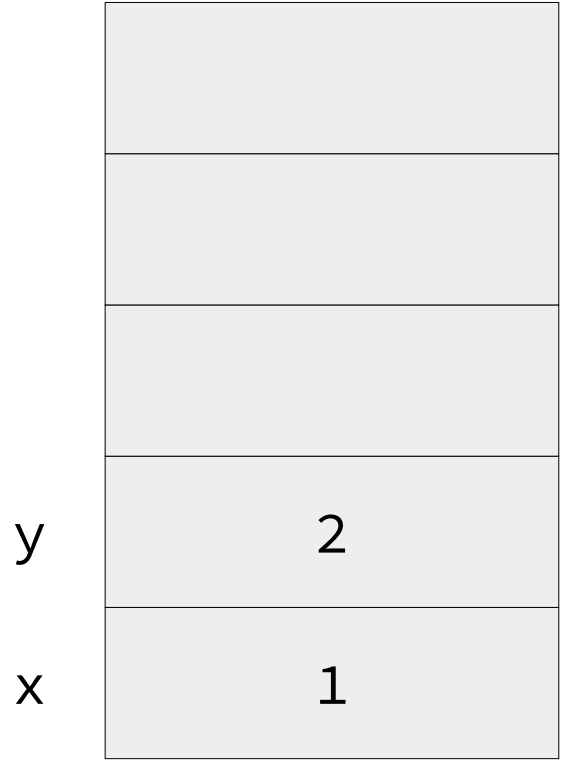
Swap – call by value

```
void swap(int a, int b)
{
    int t;
    t = a;
    a = b;
    b = t;
}
...
int x, y;
x = 1;
y = 2;
swap(x, y);
```



Swap – call by value

```
void swap(int a, int b)
{
    int t;
    t = a;
    a = b;
    b = t;
}
...
int x, y;
x = 1;
y = 2;
swap(x, y);
```



`swap2.c`

Swap – call by reference

Swap – call by reference

```
#include <stdio.h>

void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
```

Formal parameters are now pointers

***a** means the value **a** points to

Swap – call by reference

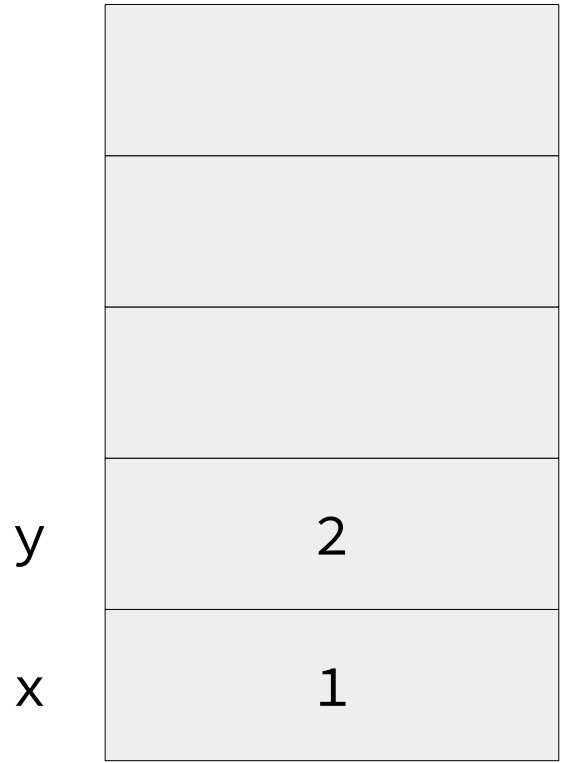
```
int main(int argc, char *argv[])
{
    int x, y;
    x = 1;
    y = 2;
    swap(&x, &y);
    printf("x: %d, y: %d\n", x, y);
    return 0;
} →
```

Actual parameters need to be pointers too – use **&** to get pointer

```
x: 2, y: 1
```

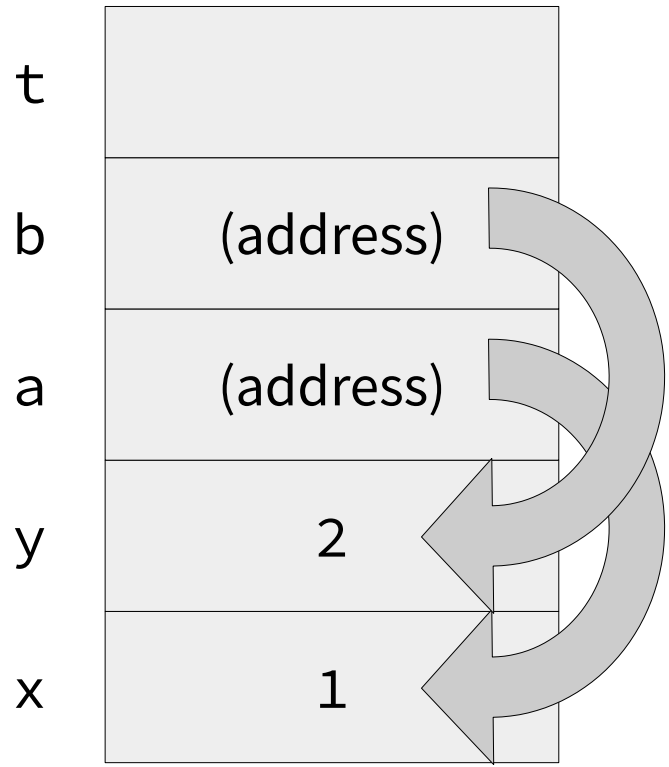
Swap – call by reference

```
void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
...
int x, y;
x = 1;
y = 2;
swap(&x, &y);
```



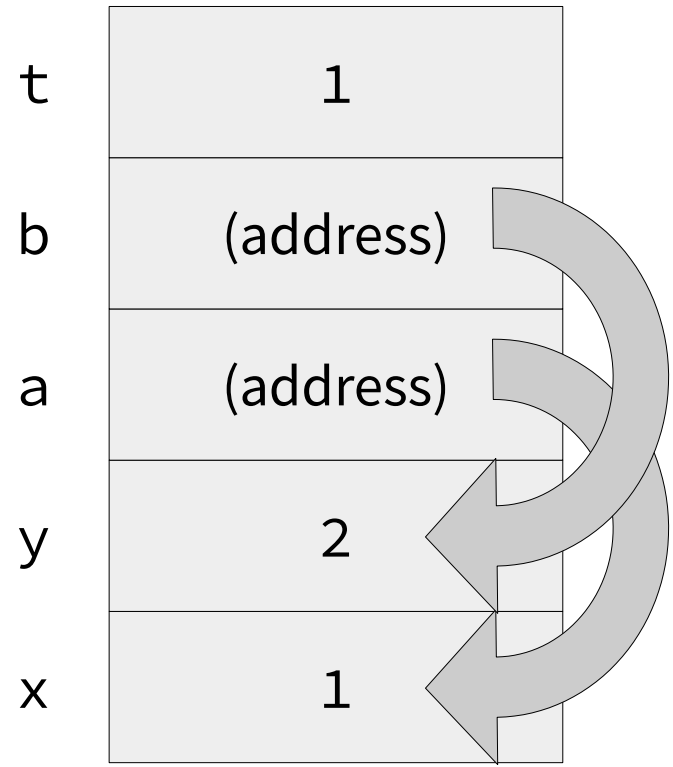
Swap – call by reference

```
void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
...
int x, y;
x = 1;
y = 2;
swap(&x, &y);
```



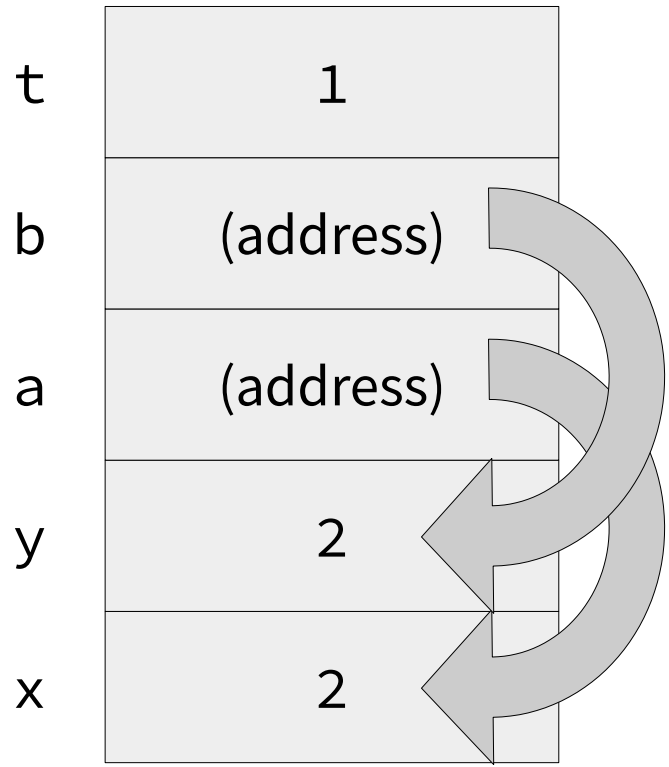
Swap – call by reference

```
void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
...
int x, y;
x = 1;
y = 2;
swap(&x, &y);
```



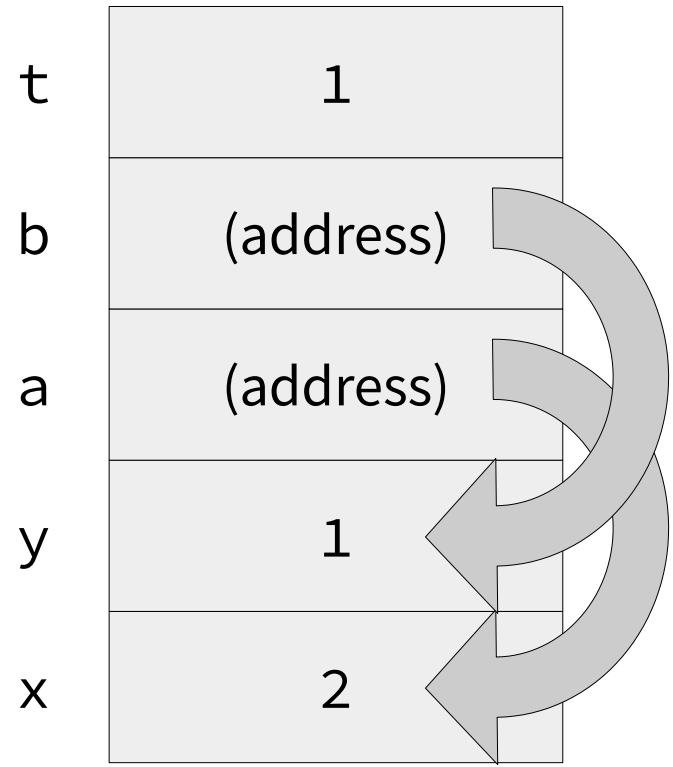
Swap – call by reference

```
void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
...
int x, y;
x = 1;
y = 2;
swap(&x, &y);
```



Swap – call by reference

```
void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
...
int x, y;
x = 1;
y = 2;
swap(&x, &y);
```



`pointers.c`

Pointer information