

# Systems Programming

## Lecture 9: Scope

Stuart James

[stuart.a.james@durham.ac.uk](mailto:stuart.a.james@durham.ac.uk)



# Recap

<https://PollEv.com/stuartjames>

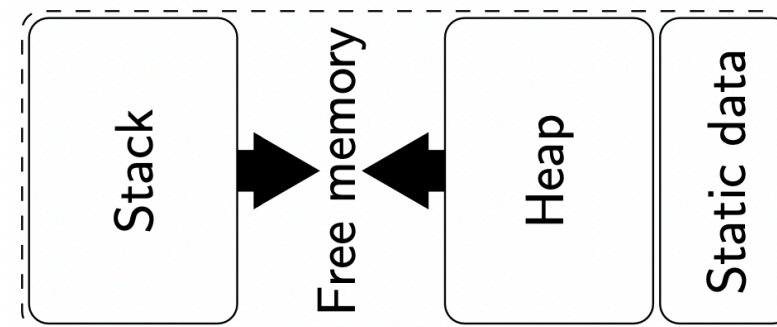


# Recap

Last lecture, we learned about **Memory Layout**

## Stack:

- Stores “temporary” data
- Variables in a function
- Function header
- Small data

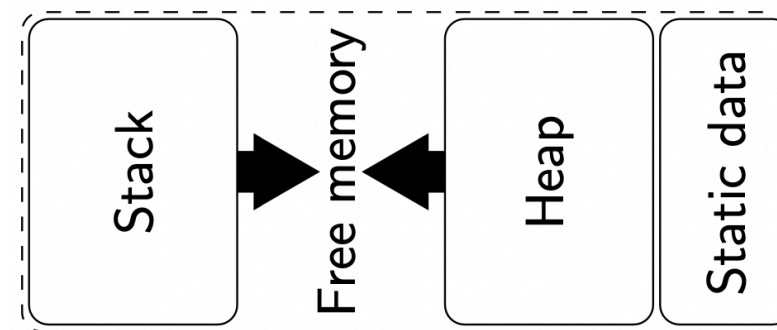


# Recap

## Memory Layout

### Heap:

- Used for storing more long-term data.
- YOU, the programmer, control what is in the Heap and when it is released.
- Much more space than the Stack.



## Recap: `malloc()` `<stdlib.h>`

Function prototype: `void *malloc(size_t size);`

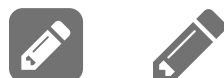
- Allocates a contiguous block of memory `size` bytes long.
- The return type is `void*`, which is a generic pointer type that can be used with all types.
- Returns a `NULL` pointer if it fails to allocate the requested memory.
- Always test for `NULL` return in case `malloc()` has failed to allocate the requested memory!



## Recap: `free()` `<stdlib.h>`

Function prototype: `void free(void *ptr);`

- Takes a generic pointer to a block of memory and returns the memory for reuse by `malloc()`.
- If you “forget” about memory you have `malloc()` ed and don’t `free()` it then you have a “memory leak”.
- “Memory leaks” can be very dangerous and difficult to trace [garbage collection in Java/Python].
  - Can eventually use up all memory
  - `free()` has no return value, so even if you pass it a pointer not allocated by `malloc()`, it will process it!



## Recap: `calloc()` `<stdlib.h>`

Function prototype for `void *calloc( size_t n, size_t size );`

- Allocates a contiguous block of memory of `n` elements each of `size` bytes long, initialised to 0 in all bits.
- Useful to ensure old data is not reused inappropriately.
- The return type is `void*` - generic pointer type used for all types.
- `calloc()` returns a `NULL` pointer if it fails to allocate memory.



## Recap: `realloc()` `<stdlib.h>`

Function prototype for `void *realloc( void *ptr, size_t size );`

- Allows a dynamic **change** in size of a previously allocated block of memory pointed to by `ptr`.
  - `ptr` must point to memory previously allocated by `malloc()`, `calloc()` or `realloc()`.
- Moves and copies contents if it needs to, freeing original block, which means `ptr` might change.
- Returns a `NULL` pointer if it fails.



# Scope

An identifier in a C program is visible (that is it may be used) only in some possibly discontinuous portion of the source code called its **scope**.

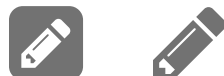
- Block Scope
- Program Scope
- Function Scope
- File Scope
- Prototype Scope



# Scope - where the name can be seen

```
int i;           // i has program scope
                 // and is accessible anywhere
int foo(int j) { // foo() also has program scope
    int i;       // this i has block scope
                 // and is only accessible between {}
    if (...) {
        int i;   // this i also has block scope
    }
}

static bar() {...} // bar() has file scope and is only
                   // accessible by code in this file
float pab(int k);  // k has prototype scope and is only
                   // accessible as part of the prototype
```



In [4]:

```
1 #include<stdio.h>
2 int i=2;
3 void foo(int j);
4
5 int main(){
6     printf("i at the top of main = %d\n",i);
7     foo(3);
8     if (i==2) {
9         int i=4;
10        printf("i in the if block = %d\n",i);
11    }
12    printf("i after the code block = %d\n",i);
13
14    return 0;
15 }
16
17 void foo(int j){
18     int i=3;
19     printf("i in the function = %d\n", i);
20 }
```

```
i at the top of main = 2
i in the function = 3
i in the if block = 4
i after the code block = 2
```



# Lifetime – variable birth and death

- Static – life of the program
- Automatic – till the end of the current block
- Dynamic – we control (`malloc()` / `free()`)

```
int* d;
int foo(int j) {
    static int t;           // static
    int p;                  // automatic
    d = malloc(400*sizeof(int)); // dynamic
}
int bar(int k) {
    free(d);
}
```



# Storage Classes

A storage class defines the scope (visibility) and life-time of variables and/or functions.

- Each variable in C has one of the following four storage types (these are also keywords):
  - extern
  - static
  - auto
  - register



# Extern

- When a variable is defined, it is allocated storage.
  - possibly initialised (`int i = 5;`)
- When a variable is declared, compiler knows a variable of a given type exists.
- Top-level variables default to extern storage class.
  - including definition and declaration
  - different from the `extern` keyword
- Use `extern` keyword to declare but not define a variable.
  - i.e. it will be defined elsewhere but accessible here
- Lifetime and scope of whole program



# extern Keyword

func.c

```
int cost;

int compute_cost(int q) {
    return q * cost;
}
```

main.c

```
#include <stdio.h>
extern int cost;
int compute_cost(int q);
int main() {
    cost = 5;
    printf("cost = %d\n",
        compute_cost(3));
    return 0;
}
```

- To run:

```
gcc -c func.c
gcc main.c func.o
./a.out
```



# static

- `static` and `extern` are mutually exclusive as keywords.
- Static variables have the same lifetime as the program.
- Static *global* variables (i.e. those outside function declarations) have **file scope**.
- Static *local* variables (i.e. those inside function declarations) have **function scope**.
- A static variable inside a function keeps its value between invocations.
- Calling a variable `static` can sometimes be confusing because it means different things in different languages!



# auto

- `auto` variables have the same lifetime as the function where they are defined.
  - They have **function scope** and **function lifetime**.
- Local variables are automatic by default, so the `auto` keyword is never explicitly used in practice.
- `auto` was part of C from the early days to make it easier to convert code from B, where it was necessary when defining local variables.
- *N.B.* `auto` has a very different meaning in C++!
  - where `auto` means the compiler has to infer the variable's type from the call-chain, using the type of the value used as initialiser.



# register

- Suggests that a variable should (if possible) be stored in a **register** rather than in main memory.
- Cannot use the **address of (&)** operator on register variables.
- Storing in a register is much faster to access.
- Not all **register** variables are necessarily stored in registers (may be too many).
- Not all variables stored in registers are declared as such (code optimisation)
- Modern compilers are very good at working out which variables are best made into register variables and will do this in the background automatically, so using **register** is quite rare!



# Local Variables

- Automatic storage duration:
  - Storage is automatically allocated when the function is called and de-allocated when it terminates.
- Block scope:
  - A local variable is visible from its point of declaration to the end of the enclosing function body.
  - These are stored in the function context on the call stack.
- In performance terms they do add a small overhead to each function call.



# Example Stack

The stack is an area of memory used for temporary storage, often used for:

- Return addresses
- Local variables
- Parameters
- Return values

```
int function(int p1,
            int p2, int p3) {
    int A, B, C;
    ...
}
```

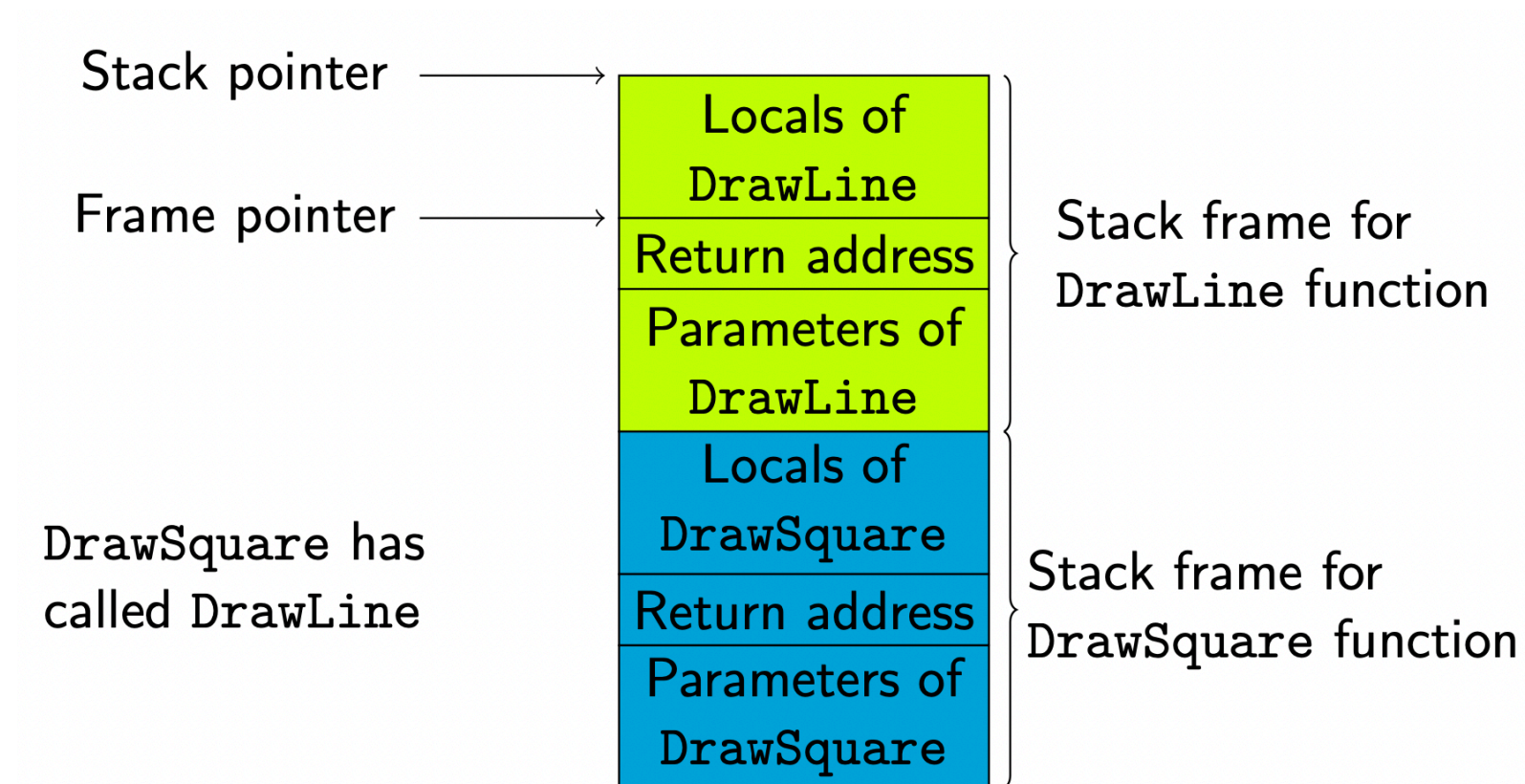
Variable A
Variable B
Variable C
Return address
Parameter p1
Parameter p2
Parameter p3
Some other value



# Call Stack

The call stack is divided up into contiguous pieces called stack frames

- each frame is the data associated with one call to one function and contains the function's *arguments*, *local variables* and the *execution address*.



# Code Block Scope

Block scope refers to any code block not just functions...

```
if (a > b) {  
    int tmp = a;  
    // tmp is local to this code block  
  
    a = b;  
    b = tmp;  
}
```

- tmp is automatic and local



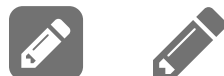
# Static and Global Variables

```
// scope inside a single source file
int a = 10; // global & static
static int c = 1; // file & static

foo() {
    int tmp = 3; // local automatic
    static int count = 0; // local static
    a = a + tmp;
    count++;
}
```

**static** variables exist for the duration of the program.

- Variables declared outside functions: visible in program and static by default.



# Static and Global Variables

```
// scope inside a single source file
int a = 10; // global & static
static int c = 1; // file & static

foo() {
    int tmp = 3; // local automatic
    static int count = 0; // local static
    a = a + tmp;
    count++;
}
```

- Same count variable each time you call `foo()`.
- A `static` variable inside a function keeps its value between invocations.



In [5]:

```
1 #include<stdio.h>
2
3 // scope inside a single source file
4 int a = 10;           // global & static
5 static int c = 1;     // file & static
6
7 void foo();
8
9 int main(){
10     printf("in main: a = %d c = %d\n", a, c);
11     foo();
12     printf("in main: a = %d c = %d\n", a, c);
13     foo();
14     printf("in main: a = %d c = %d\n", a, c);
15     foo();
16     return 0;
17 }
18
19 void foo(){
20     int tmp = 3;       // local automatic
21     static int count = 0 ; // local static
22     a = a + tmp;
23     count++;
24     printf("in foo : tmp = %d count = %d\n", tmp, count);
25 }
```

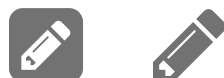
```
in main: a = 10 c = 1
in foo : tmp = 3 count = 1
in main: a = 13 c = 1
in foo : tmp = 3 count = 2
in main: a = 16 c = 1
in foo : tmp = 3 count = 3
```



# Function Parameters

Parameters have the same properties as local variables:

- i.e. automatic storage duration and block scope.
- Each parameter is initialised automatically when a function is called (by being assigned the actual value of the corresponding argument)



# Summary of scope in a single file

file1.c:

```
int gv;           // gv - global scope (static)

static int fv;    // fv - file scope (static)

void f( int pv ){ // pv - block scope of f() (automatic)

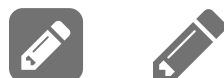
    int lv = 0;    // lv - block scope (automatic)

    static int sv = 0; // sv - block scope (static)
}
```



# Pros and Cons of Global Variables

- Global variables are convenient when many functions must share a variable or when a few functions share a large number of variables.
- In most cases, it is better for functions to communicate through parameters rather than shared variables:
  - If we change a global variable during program maintenance (by altering its type, say), we'll need to check every function in the same file to see how the change affects it.
  - If a global variable is assigned an incorrect value, it may be difficult to identify the guilty function.
  - Functions that rely on global variables are hard to reuse elsewhere.



# Summary

- Scope
- Lifetime
- Storage classes
  - `extern`
  - `auto`
  - `static`
  - `register`
- Local and Global variables
- Call stack

