Scope and Recursion

1 Scope – where 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
```

2 Scope – where it can be used

```
int i;                     i
                           i
                           i
int foo(int j) {           i
  int i;                     i
      i;                     i

  if (...) {                 i
    int i;                     i
  }                          i
}                          i
static bar() {...}         i
                           i
float pab(int k);          i
```

- Which i is visible?

3 Lifetime – variable birth and death

- Three types of lifetime:
  - 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);
}
```

4 Storage classes

- Each variable in C has one of the following four storage types (these are also keywords):
- `extern` (not the same as `extern` declaration)
- `static`
- `auto`
- `register`

5 extern

- When a variable is defined it is allocated storage
  - possibly initialised (`int i = 5;`)
- When a variable is declared it informs the compiler that a variable of a given type exists
- Top-level variables default to `extern` storage class
  - including definition and declaration
  - but not 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
- Cf. abstract classes and interfaces in Java

6 extern keyword

- Use a variable from a different file

func.c

```c
int cost;

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

main.c

```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
```

7 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*
- Calling a variable `static` is confusing because it means different things in different languages
  - and also within C

8 `auto`

- Automatic variables have the same lifetime as the function in which they are defined
- They have function scope
- Automatic variables are stored in the *stack frame*
- 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++!)

9 `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

10 Local variables

- Properties of 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

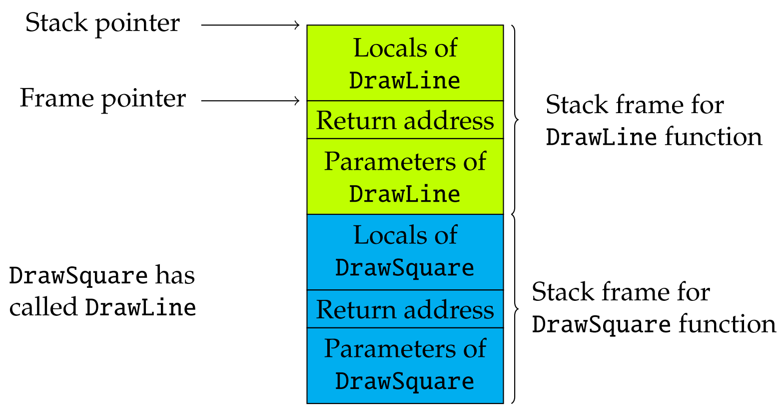
11 Example Stack

- The stack is an area of memory used for temporary storage
- Often (but not always) used for
  - Return addresses
  - Local variables
  - Parameters
  - Return values

```c
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

12 Call stack example



13 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

14 Static and global variables

- `static` variables exist for the duration of the program
- Variables declared outside a function are visible to all code in the same program and are `static` by default

```
// 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()`

15 Function parameters

- Parameters have the same properties as local variables
  - i.e. automatic storage duration and block scope
  - Each formal parameter is initialized automatically when a function is called (by being assigned the actual value of the corresponding argument)

16 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 - blck scope (static)
}
```

17 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's 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 in other programs

18 Iterative functions

```
int loop_power( int a, int n ) {
  int result = 1;
  while (n > 0 ) {
    result = result * a;
    n--;
  }
  return result;
}
```

- Calculate `a` raised to the power `n`
- `1 * a * a *` $\cdots$ `n` times

19 Recursive functions

- Recursion is an alternative to using a loop
- C allows this by allowing functions to call themselves
- Like any loop this needs:
  - initial conditions
  - conditional test (a termination test)
  - a variable change, e.g. a decrement
- Relies on a new function scope being created every time a function calls itself

20 Recursive power function

```
int recursive_power(int a, int n){
  if ( n == 0 )  // termination test
    return 1;    // base case
  else {         // recursive case
    return ( a * recursive_power( a, n - 1 ) );
  }
}
```

- The loop variable here is `n`, which decrements to zero as repeated recursive calls are made
- N.B. Repeated need to initialise the function context is a cost

21 Recursive power function – short form

```
int recursive_power2( int a, int n ) {
  return (n == 0) ? 1 : (a * recursive_power2(a,n-1));
}
```

- The same function written with the conditional operator:

  ```
  value = expr1 ?  expr2 : expr3;
  ```

- is the same as

  ```
  if (expr1)
    value = expr2;
  else
    value = expr3;
  ```

22 Factorials

- Look at `factorial.c` (a little simpler than `power.c`)
- `gcc -g -Wa,-ahl=factorial.s factorial.c`
- This will interleave assembly language with source code statements
- If you know the way function calls are made, you can mix C programs with assembly language programs

23 Recursive Fibonacci function

```
int fib(int n) {
  if(n >= 2)      // recursion test
    return fib(n-1) + fib(n-2); // recursive case
  return n;       // base case for n = 0 and n =1
}
```

- This returns the `n`th element in the Fibonacci series:
- `0,1,1,2,3,5,8,13,21`, etc.
- Base case: `fib(0)` is `0` and `fib(1)` is `1`
- Production rule: `fib(n) = fib(n-1) + fib(n-2)`

24 Recursion summary

- Write loops using a function that calls itself
- This must have both a:
  - base case
  - recursive case
- To terminate, the base case must happen
- Relies on the run time system to:
  - create the function's scope
  - keep track of the local variables for each call
  - This is a performance overhead compared to iterative loops