Scope

1 Memory allocation: calloc() <stdlib.h>

- Function prototype for calloc()

```
void *calloc( size_t n, size_t  size );
```

- Allocates a contiguous block of memory of n elements each of size bytes long, initialised to all bits 0
- Useful to ensure old data is not reused inappropriately
- The return type is void*, which is a generic pointer type that can be used for all types
- calloc() returns a NULL pointer if it fails to allocate the requested memory
- Always test for NULL return!

2 Memory allocation: realloc() <stdlib.h>

- Function prototype for realloc()

```
void *realloc( void *ptr, size_t  size );
```

- Allows a dynamic change in size of an allocated block of memory pointed to by ptr
  - ptr must point to memory previously allocated by malloc(), calloc() or realloc()
- Will move and copy contents if it needs to, freeing original block
- realloc() returns a NULL pointer if it fails. Check for this!
- Cf. ArrayList in Java

3 realloc() example

- Simple program that takes integers typed in by the user and stores them in an array
- Each time the array becomes full, it is dynamically increased in size to hold more numbers
- Contains a key function getline2(), which reads the integers from the command line

```
int getline2(char line[], int max) {
    int nch = 0;
    int c;
    max = max - 1;/* leave room for '\0' */
    while((c = getchar()) != 'q') {
        if(c == '\n')
            break;
        if(nch < max) {
            line[nch] = c;
            nch = nch + 1;
        }
    }
    if(c == 'q' && nch == 0)
        return 'q';

    line[nch] = '\0';
    return nch;
}
```

4 realloc() Example

- `getline2()`
- Uses `getchar()` to read in characters as they are typed
- Runs in a loop until a `'q'` or a newline is encountered
- Reads in the characters typed by the user one by one and stores them in the array line
- When the character `'\n'` is pressed, the function returns, via use of the `break` statement to exit a loop
- No checking performed to see if the input is an integer

```
ip = malloc(array_size * sizeof(int));
while( getline2(line, MAXLINE) != 'q' ) {
    if(nitems >= array_size ) {/* increase allocation */
        int *newp;
        array_size += INCREASE ;
        newp = realloc(ip, array_size * sizeof(int));
        printf("<< Expanding by %d to size %d >>\n",
                            INCREASE, array_size );
        if(newp == NULL) {
            printf("out of memory\n");
            exit(1);
        }
        ip = newp;
    }
    ip[nitems++] = atoi(line);
}
```

5 realloc() example

- `main()`
- Uses `getline2()` to read in a line of text
- Creates an array to store current line of text, `line`
- Creates a second array to store the integers entered: `ip`
- As soon as `ip` is full, `realloc()` is called to resize the array

6 atoi() <stdlib.h>

```
int atoi(const char *s);
```

- Converts a string pointed to by s to an integer
- Also see `atof()`, `atol()` and `atoll()` (since C99) equivalents
- To convert from an integer to a string use :

  ```
  int sprintf( char *s, char *format, <value list> );
  ```

- Where the value list is the variables used in the format string

7 -> Operator

- The -> operator gives us a shorthand accessing members of structures using a pointer.

```
struct point {
  int x;
  int y;
} pt, *ptr;
ptr=&pt;
```

- We can now modify `pt.x` in three ways:

```
pt.x=3;     // Access directly

(*ptr).x=3; // Access by dereferencing a pointer

ptr->x=3;   // Access using the -> operator
```

8 Course details

- Intro, HelloWorld, Compiling, Pre-processor
- Control flow and functions
- Data types, structs and unions
- Memory access using pointers
- Dynamic memory management
- Scope of variables and recursive functions
- Large programs and external libraries
- Debugging
- UNIX/Linux and C
- C++

9 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
```

10 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?

11 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);
}
```

12 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

13 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

14 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
```

15 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

16 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++!)

17 `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

18 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

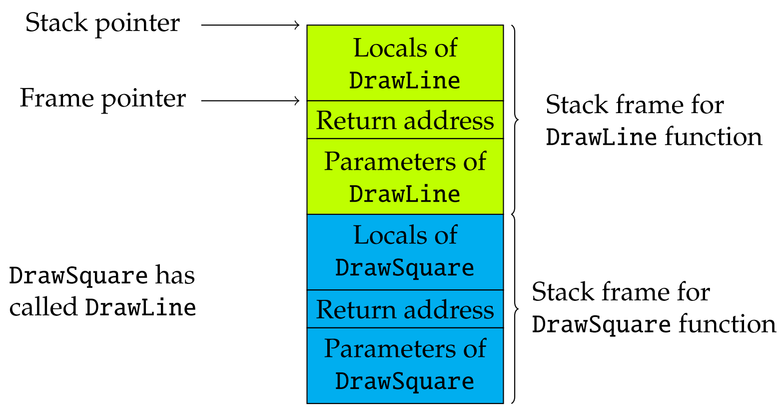
19 Example Stack

- 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

20 Call stack example



21 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

22 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()`

23 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)

24 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)
}
```

25 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