COMP20200 Unix Programming

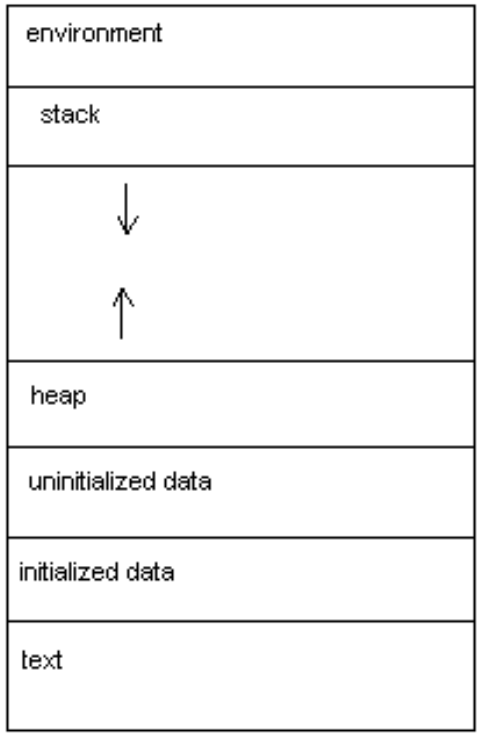
Lecture 5

CS, University College Dublin, Ireland

Virtual Memory

A process's address space typically has 6 sections:

- Environment
  - Environment variables
  - Command line arguments
- Stack
  - Function arguments
  - Return values
  - Automatic variables
- Heap
  - Dynamic allocation
- Data (uninitialized/initialized)
  - Static & global variables
- Text
  - The program code



Virtual memory organization

Variables in C: a recap

- Variable is a named region of storage
  - Specific chunk of memory associated with variable
  - Can hold a single value
  - Must declare name and type before using it
    - Definition = declaration + memory allocation
- Variable scope:
  - Local Variables
  - Global Variables
- 3 types of memory allocation
  - Static
  - Automatic
  - Dynamic
- C storage "classes" or qualifiers
  - auto register static extern

Local & Global variables

- Local
  - Scope of visibility: the block or function where it is declared.

```
int i;
{
    int i = 0;
    i++;
}
i = 2;
```

Local & Global variables

- Global
  - Defined outside of function
  - Scope: Visible in all functions after definition.
  - Functions themselves are global
  - Can produce unexpected behaviour when different functions edit same variable. Difficult to debug and track error.
  - Should be avoided except when absolutely necessary.

```
int b;

int my_funct(int a){
  a += b
  b = 0;
  return a;
}
```

Automatic

- The default memory type for **local** variables
- Only be used within functions.
- When defined, memory is allocated in stack but not initialised
- When the block ends the variable 'dies'.

```
int main(){
  auto int j;
  j = 1;
  int i = 4;
  i++;
  {
    int i = 100;
    printf("i is %d\n",i); // Prints:100
  } //'i' (value 100) dies here.
  printf("i is %d\n",i); // Prints: 5
return 0; }
```

register

- Automatic variables that should be stored in a register
- Used for optimisation
- Compiler can choose to ignore this
- Cannot apply unary '&' operator, no memory location
- Cannot be greater than register size (usually one word)

```
{
  register int count;
}
```

Static

- Default memory type for **global** variables
- Can be local or global.
- Life time equal to the life time of the program.

Wrong

```
char *Func(void);
main() {
   char *Text1;
   Text1 = Func();
   printf("%s\n", Text1);
}

char *Func(void) {
   char Text2[10]="hello";
   return(Text2);
}
```

Right

```
char *Func(void);
main() {
   char *Text1;
   Text1 = Func();
   printf("%s\n", Text1);
}

char *Func(void) {
   static char Text2[10]="hello";
   return(Text2);
}
```

extern

- For multiple source files
- Declares a global variable defined elsewhere and visible to ALL object modules

Source 1

```
extern int count;
write(){
  printf("count: %d\n", count);
}
```

Source 2

```
int count=5;
main() {
  write();
}
```

```
gcc source1.c source2.c -o program
```

Dynamic Memory

- Memory is allocated as needed.
- Memory size may not be known until program runtime
  (size of static and automatic variables required at compile-time)
- Life time: From when manually created (`malloc`, `calloc`) until freed (`free`)

```
int* array = malloc(sizeof(int) * 10);
if (array == NULL) {
/* Handle error   */
}
...
free(array);
array = NULL;
```

Malloc, Calloc

- Returns pointer to `void *`
- Not guaranteed to succeed, returns `NULL`
- Common errors
  - Not checking for allocation failures
  - Memory leaks
  - Logical errors (using after free or before malloc)
- `malloc()` does not initialise memory
- `calloc()` sets to zero

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
```

Size of types: sizeof()

```
printf("char          %zu\n", sizeof(char));
printf("short int     %zu\n", sizeof(short int));
printf("int           %zu\n", sizeof(int));
printf("long long int %zu\n", sizeof(long long int));
printf("float         %zu\n", sizeof(float));
printf("double        %zu\n", sizeof(double));
printf("int*          %zu\n", sizeof(int*));
printf("sizeof        %zu\n", sizeof(sizeof(int)));
```

```
$ cat /proc/cpuinfo | grep addr |head -n1
```

32 bit address sizes :

```
char          1
short int     2
int           4
long long int 8
float         4
double        8
int*          4
sizeof        4
```

64 bit address sizes :

```
char          1
short int     2
int           4
long long int 8
float         4
double        8
int*          8
sizeof        8
```

Program Interaction

```
$ ./my_prog
Hello!
Please enter name:
> Bob
Hello Bob, please enter score:
> 45
Please enter file:
> scores.dat
Scores updated, goodbye!
```

Program Interaction

```
$ ./my_prog
Hello!
Please enter name:
> Bob
Hello Bob, please enter score:
> 45
Please enter file:
> scores.dat
Scores updated, goodbye!
```

- Not good programming style.
- Cannot be used in scripts.
- One example where it is used is `passwd`

Program Arguments

```
$ ./myprog -n Bob -s 45 scores.dat

$ ./myprog -s 45 -n Bob scores.dat

$ ./myprog -l scores.dat

$ ./my_prog -h  # normally prints help message
```

- Arguments used to change the behaviour of programs.

argc, argv

```
#include <stdio.h>
int main (int argc, char *argv[]){
  return 0;
}
```

```
$ ./myprog -s 45 -n Bob scores.dat
```

```
argc 6
argv[0] "./myprog"
argv[1] "-s"
argv[2] "45"
argv[3] "-n"
argv[4] "Bob"
argv[5] "scores.dat"
```

Program Arguments

```
$ cp from to
$ cp -i -v from to
$ cp -iv from to
$ cp -vi from to
```

- All of the above ultimately do the same
- `int argc, char *argv[]` standard in C.
- Is "from" path `argv[1]`, `argv[2]`, or `argv[3]`?
- Write complex logic in every program to decide? - No!

```
man 3 getopt
#include <unistd.h>

int getopt(int argc, char * const argv[],
                const char *optstring);}
extern char *optarg;
extern int optind, opterr, optopt;
```

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

   int main(int argc, char *argv[]) {
   int flags, opt;
   int nsecs, tfnd;
   nsecs = 0;
   tfnd = 0;
   flags = 0;
   while ((opt = getopt(argc, argv, "nt:h")) != -1) {
     switch (opt) {
       case 'n':
         flags = 1;
         break;
       case 't':
         nsecs = atoi(optarg);
         tfnd = 1;
         break;
       case 'h':
         fprintf(stderr, "Help message how to use tool\n");
         exit(EXIT_SUCCESS);
         break;
```

```
        default: // '?'
            fprintf(stderr, "Usage: %s [-t nsecs] [-n] name\n",
                    argv[0]);
            exit(EXIT_FAILURE);
        }
    }

    printf("flags=%d; tfnd=%d; optind=%d\n", flags, tfnd, optind);
    if (optind >= argc) {
        fprintf(stderr, "Expected argument after options\n");
        exit(EXIT_FAILURE);
    }

    printf("name argument = %s\n", argv[optind]);

    /* Other code omitted */

    exit(EXIT_SUCCESS); }
```

Getopt

```
opt = getopt(argc, argv, "nt:h")
```

- this takes arguments `-n` `-t` `-h` or `-nh`
- `-t` has a value because of `:`.

- Command line tip(s) of the day:
  - Hidden files have names staring with "."
    To list them: `ls -la`
  - `rm` can be dangerous
    Never type command like `# rm -rf /`
    Made safer with interactive flag `rm -i`
    Can edit .bashrc file and add `alias rm='rm -i'`
- Vi tip(s) of the day:
  - `gg` Moves to start of file
  - `G` Moves to end of file
  - `=` auto-indent code
  - `==` indents current line
  - `=G` indents from current position to end
  - Indent whole file with `gg =G`