Memory management, File IO, ...

Jamal Hussein

jah1g12@ecs.soton.ac.uk

Electronics and Computer Science

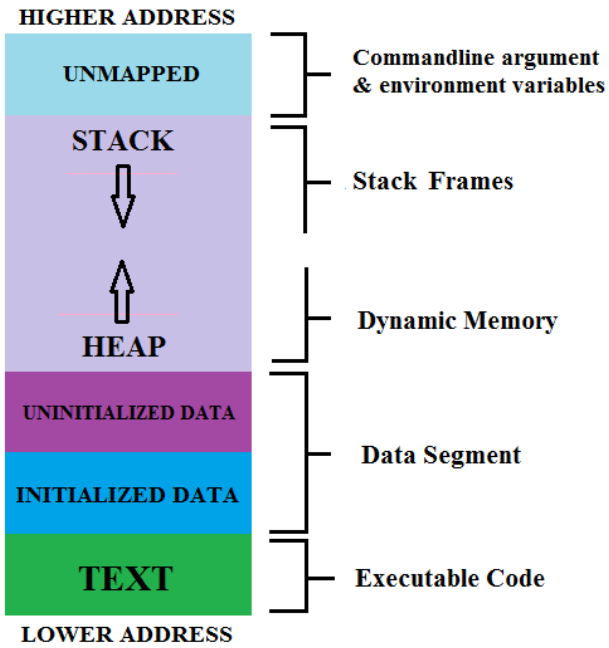
December 4, 2015

UNIVERSITY OF Southampton

Outline

Memory Layout - Heap

- Global variables, static variables and program instructions get their memory in permanent storage area
- local variables are stored in Stack
- The memory space between these two region is known as Heap area.
- This region is used for dynamic memory allocation during execution of the program.
- The size of heap keep changing.



Memory Allocation (stdlib.h)

- The header `<stdlib.h>` declares functions for storage allocation
- `malloc` returns a pointer to space for an object of a specified size, or `NULL` if the request cannot be satisfied. The allocated space is uninitialized

```
void *malloc(size_t size)
```

- `free` deallocates the space that previously allocated by `calloc`, `malloc`, or `realloc`

```
void free(void *P)
```

- `size_t` is an unsigned integer defined in `stddef.h` as (on my computer):

```
typedef long unsigned int size_t;
```

Memory Allocation (malloc)

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

int main() {
        char *language;

        language = malloc( 200 * sizeof(char) );

        if( language == NULL ) {
                fprintf(stderr, "Error - ...\n");
        }
        else {
                strcpy( language, "C programming language");
        }
        printf("Language: %s\n", language );
        free(language);
        return 0;
}
```

Memory Allocation (stdlib.h)

- `calloc` returns a pointer to space for an array of objects, each of size `size`, or `NULL` if the request cannot be satisfied. The space is initialized to zero bytes.
  `void *calloc(size_t nobj, size_t size)`
- `realloc` changes the size of the previously allocated memory by a new size.
- `realloc` returns a pointer to the new space, or `NULL` if the request cannot be satisfied, in which case the old allocated memory is unchanged.
  `void *realloc(void *p, size_t size)`

Memory Allocation (calloc)

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

int main() {
        char *language;

        language = calloc( 200, sizeof(char) );

        if( language == NULL ) {
        fprintf(stderr,
                "Error - ...\n");
        }
        else {
                strcpy( language, "C programming language");
        }
        printf("Language: %s\n", language );
        free(language);
        return 0;
}
```

Memory Allocation (realloc)

```
int main() {
        char *language;
        language = malloc( 25 * sizeof(char) );
        if( language == NULL ) {
                fprintf(stderr, "...\n");
        }
        else {
                strcpy( language, "C programming language");
        }
        language = realloc( language, 10 * sizeof(char) );
        if( language == NULL ) {
                fprintf(stderr, "...\n");
        }
        else {
                strcat( language, " tutorial");
        }
        printf("Language: %s\n", language );
        free(language);
        return 0;
}
```

Memory Allocation (pointer)

```
int main() {
        int *iptr = malloc (sizeof(int));
        float *fptr = malloc (sizeof(float));
        double *dptr = malloc (sizeof(double));

        *iptr = 100;
        *fptr = 4.13f;
        *dptr = 459000.0;

        printf("*iptr: %d\n", *iptr);
        printf("*fptr: %.2f\n", *fptr);
        printf("*dptr: %.2f\n", *dptr);

        free(iptr);
        free(fptr);
        free(dptr);
        return 0;
}
```

UNIVERSITY OF Southampton

File IO (stdlib.h)

- `fopen` opens the named file, and returns a stream, or `NULL` if the attempt fails

  ```
  FILE *fopen(const char *filename, const char *mode)
  ```

  Legal values for mode include:

"r"	open text file for reading
"w"	create text file for writing; discard previous contents if any
"a"	append; open or create text file for writing at end of file
"r+"	open text file for update (i.e., reading and writing)
"w+"	create text file for update; discard previous contents if any
"a+"	append; open or create text file for update, writing at end

File IO (stdio.h)

- `fflush` causes any buffered but unwritten data to be written on an output stream
  `int fflush(FILE *Stream)`
- on an input stream, the effect is undefined
- It returns `EOF` for a write error, and zero otherwise
- `fflush(NULL)` flushes all output streams
- called between a write and a read
- `fclose` flushes any unwritten data for stream, discards any unread buffered input
  `int fclose(FILE *Stream)`
- frees any automatically allocated buffer, then closes the stream
- It returns `EOF` if any errors occurred, and zero otherwise

File IO

- Writing a file:
  - `fputc()` writes the character value to an output stream

    ```
    int fputc( int c, FILE *fp );
    ```

  - `fputs()` writes a string to an output stream

    ```
    int fputs( const char *s, FILE *fp );
    ```

  - `fprintf` write a string to an output stream

    ```
    int fprintf(FILE *fp, const char *format,
    ...)
    ```

File IO

```c
#include <stdio.h>

int main() {
FILE *fp;

fp = fopen("test.txt", "w+");

fprintf(fp, "This is printed by fprintf...\n");
fputs("This is printed by fputs...\n", fp);

fclose(fp);
}
```

File IO

- Reading a file:
  - `fgetc()` reads a character from the input file. he return value is the character read, or in case of any error, it returns `EOF`
    ```
    int fgetc( FILE * fp );
    ```
  - `fgets()` reads up to `n-1` characters from an input stream. It copies the read string into a buffer, appending a null character to terminate the string.
    ```
    char *fgets( char *buf, int n, FILE *fp );
    ```
  - `fscanf` reads strings from a file, but it stops reading after encountering the first space character
    ```
    int fscanf(FILE *fp, const char *format,
    ...)
    ```

File IO

```c
#include <stdio.h>
main() {
        FILE *fp;
        char buff[255];
        fp = fopen("test.txt", "r");

        fscanf(fp, "%s", buff);
        printf("1: %s\n", buff );

        fgets(buff, 255, (FILE*)fp);
        printf("2: %s\n", buff );

        fgets(buff, 255, (FILE*)fp);
        printf("3: %s\n", buff );

        fclose(fp);
}
```

File IO

This is the output of the program:

```
1: This
2:  is printed by fprintf...

3: This is printed by fputs...
```

Command-line Arguments

- In C it is possible to accept command line arguments
- Command-line arguments are given after the name of a program
- passed to the program by the operating system
- The full declaration of main looks like this:
  `int main ( int argc, char *argv[] )`
  - `argc`: number of command line arguments including the name of the program
  - `argv`: list of all command line arguments
    - `argv[0]`: the program name (or an empty string)
    - `argv[1] to argv[argc-1]`: the actual command line arguments

UNIVERSITY OF Southampton

Command-line Arguments

```
#include <stdio.h>
int main ( int argc, char *argv[] ) {
        if ( argc != 2 ) {
                printf( "usage: %s filename", argv[0] );
        }
        else {
                FILE *file = fopen( argv[1], "r" );
                if ( file == NULL ) {
                        printf( "Could not open file\n" );
                }
                else {
                        int x;
                        while((x = fgetc( file )) != EOF) {
                                printf( "%c", x );
                        }
                }
                fclose( file );
        }
}
```

Variable Arguments

- Sometimes, you would like to have a function that accept an arbitrary number of arguments
- for example a function that accepts any number of values and returns the average

```
double average(int num, ...)
```

- the last argument is written as ellipses, i.e. three dotes (`...`)
- the one just before the ellipses is always an `int` which will represent the total number variable arguments passed

Variable Arguments

- The header file `stdarg.h` provides the functions and macros to implement the functionality of variable arguments as follows
  - Define a function with its last parameter as ellipses and the one just before the ellipses is always an int which will represent the number of arguments.
  - Create a `va_list` type variable in the function definition
  - Use `int` parameter and `va_start` macro to initialize the `va_list` variable to an argument list
  - Use `va_arg` macro and `va_list` variable to access each item in argument list
  - Use a macro `va_end` to clean up the memory assigned to `va_list` variable

Variable Arguments

```
#include <stdio.h>
#include <stdarg.h>
double average(int num,...) {
        va_list valist;
        double sum = 0.0;
        int i;
        va_start(valist, num);
        for (i = 0; i < num; i++) {
                sum += va_arg(valist, int);
        }
        va_end(valist);
        return sum/num;
}
int main() {
        printf("Average of 2, 3, 4, 5 = %f\n",
                average(4, 2,3,4,5));
        printf("Average of 5, 10, 15 = %f\n",
                average(3, 5,10,15));
}
```

typedef

- The `typedef` keyword allows the programmer to create new names for types such as int
- Typedefs can be used both to provide more clarity to your code and to make it easier to make changes to the underlying data types

```
typedef long unsigned int size_t;
```

- In C, `struct` variables must be declared by a combination of the keyword `struct` and the name of the struct

```
typedef struct Books
...
} Book;
```

typedef

```
#include<stdio.h>
#include<string.h>
typedef struct employee {
        char  name[50];
        int   salary;
} emp ;
void main( ) {
        emp e1;
        printf("Enter Employee record\n");
        printf("Employee name:\t");
        scanf("%s", e1.name);
        printf("Enter Employee salary:\t");
        scanf("%d", &e1.salary);
        printf("\nstudent name is %s\n",e1.name);
        printf("roll is %d\n",e1.salary);
}
```

typedef vs #define

- `#define` is a C-directive which is also used to define the aliases for various data types similar to `typedef` but with the following differences:
  - `typedef` is limited to giving symbolic names to types only where as `#define` can be used to define alias for values as well

    ```
    #define TRUE 1
    #define FALSE 0
    ```
  - `typedef` interpretation is performed by the compiler whereas `#define` statements are processed by the pre-processor

Preprocessors

- Preprocessors are a way of making text processing with your C program before they are actually compiled
- Before the actual compilation of every C program it is passed through a Preprocessor.
- The Preprocessor looks through the program trying to find out specific instructions called Preprocessor directives that it can understand.
- All Preprocessor directives begin with the # (hash) symbol

Preprocessors

`#define`	Substitutes a preprocessor macro.
`#include`	Inserts a particular header from another file.
`#undef`	Undefines a preprocessor macro.
`#ifdef`	Returns true if this macro is defined.
`#ifndef`	Returns true if this macro is not defined.
`#if`	Tests if a compile time condition is true.
`#else`	The alternative for `#if`.
`#elif`	`#else` and `#if` in one statement.
`#endif`	Ends preprocessor conditional.
`#error`	Prints error message on stderr.
`#pragma`	Issues special commands to the compiler, using a standardized method.

Preprocessors

- `#define` is used to define values or macros that are used by the preprocessor to manipulate the program source code
- By convention, values defined using `#define` are named in uppercase
- can produce quite unexpected results if not done right

```
#define MAX(a,b) ((a)>(b)?(a):(b))
⋮
i = 2;
j = 3;
k = MAX(i++, j++);
```

- the definition must parenthesize every use of a,b in the macro definition

Preprocessors

```
#define SLICES 8
#define ADD(x) ( (x) / SLICES )
int main(void) {
        int a = 0, b = 10, c = 6;

        a = ADD(b + c);
        printf("%d\n", a);
        return 0;
}
```

- The `#undef` directive undefines a macro. The identifier need not have been previously defined.
- `#ifdef` and `#ifndef` are used to check whether the code block following them are defined or undefined respectively

Preprocessors

- ANSI C defines some useful preprocessor macros and variables

`__FILE__`	The name of the current file, as a string literal
`__LINE__`	Current line of the source file, as a numeric literal
`__DATE__`	Current system date, as a string
`__TIME__`	Current system time, as a string
`__STDC__`	defined as 1 when the compiler complies with the ANSI standard.

Preprocessors

```
#include <stdio.h>
main() {
        printf("File :%s\n", __FILE__ );
        printf("Date :%s\n", __DATE__ );
        printf("Time :%s\n", __TIME__ );
        printf("Line :%d\n", __LINE__ );
        printf("ANSI :%d\n", __STDC__ );
}
```

The output:

```
File :macros2.c
Date :Dec  3 2015
Time :16:41:02
Line :8
ANSI :1
```

UNIVERSITY OF Southampton

Header files

- A header file is a file with extension `.h` which contains C function declarations and macro definitions to be shared between several source files.
- There are two types of header files:
  - the files that the programmer writes
    `#include <stdio.h>`
  - the files that comes with your compiler
    `#include "my_header.h"`

Header files

```c
/* my_header.h file */
int max (int a, int b);
```

```c
/* my_header.c file */
int max( int x, int y){
        return x > y ? x : y;
}
```

```c
/* main program */
#include <stdio.h>
#include "my_header.h"
int main () {
        printf("max(2,3) = %d\n", max (2, 3));
        return 0;
}
```

Header files (Once-Only Headers)

- If a header file happens to be included twice, the compiler will process its contents twice and it will result in an error
- The standard way to prevent this is to enclose the entire real contents of the file in a conditional pre-processing statement:

```
#ifndef MY_HEADER
#define MY_HEADER

int max (int a, int b);

#endif
```