COMP1005
Programming and Algorithms

Dynamic Memory Allocation

Jamie Twycross

Overview

- Program memory organisation
- Dynamic memory allocation
- Garbage collection

String Copy Example 1

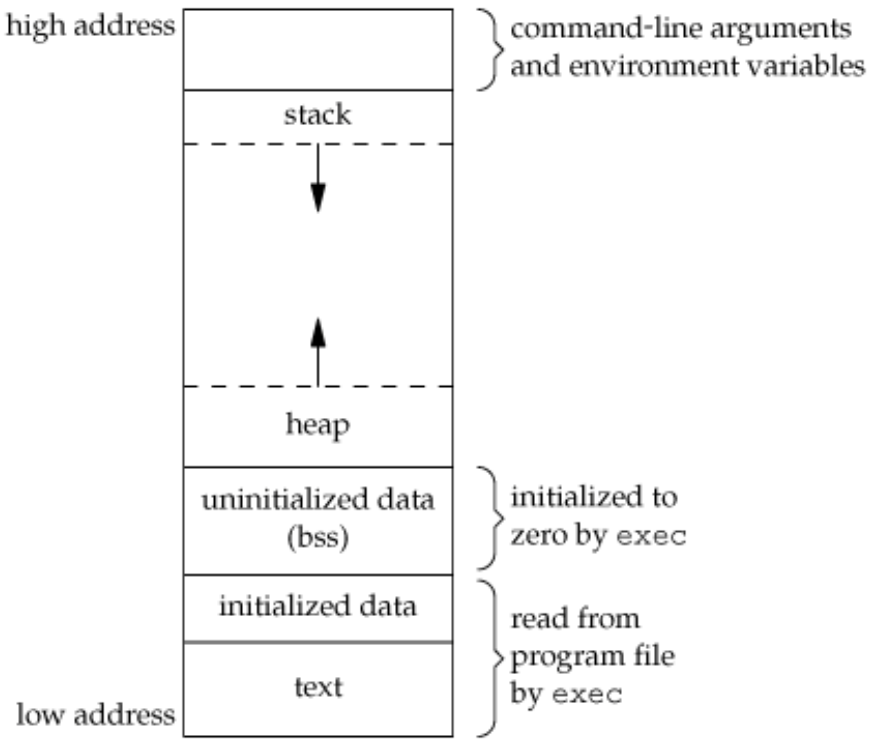
```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      char buffer[16], *p;
6      int i;
7
8      if(argc != 2)
9          return 1;
10
11     p = argv[1];
12     i = 0;
13     while(*p)
14         buffer[i++] = *p++;
15     buffer[i] = '\0';
16     printf("%s\n", buffer);
17
18     return 0;
19 }
```

Compile-time and Run-time Memory

- Array size is set at **compile-time**
- Often do not know what the size array should be till run-time
- Could guess, and hope we have enough space - **insecure**
- Better if we could set the size of the array at **run-time**
- C lets us do this using **dynamic memory allocation**
- Need to understand the way program **manages memory**
- Although all memory is identical, the program sees it as several different **sections** (**segments**)
- These sections are used for various purposes

Program Memory Layout

- **Text**: compiled executable code
- **Initialised Data**: initialised global and static variables
- **Uninitialised Data** (BSS): uninitialised statics - set to zero
- **Heap** - dynamically allocated memory
- **Stack** - automatic variables and function return addresses



Dynamic Memory Allocation

- **Automatic** variables created on the stack, and **local** to function
- Refer to them by **name** in source code
- Get a pointer to them using `&` operator
- Can also store values (data) on the **heap**
- The heap is the rest of available memory
- Only have **pointer** to heap data, no direct (named) access
- Used `malloc`, `calloc`, `realloc` and `free` standard library functions in `stdlib.h` to manage heap

`malloc()`

- **Allocate** (reserve) memory on heap using `malloc()` function

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

- In `stdlib.h`:

  ```
  #include <stdlib.h>
  ```

- Allocates `size` **bytes** on heap
- Use `sizeof` operator to calculate `size`
- Example: Allocate memory for an array of 4 integers:
  - number of bytes = `4 * sizeof(int)`
- Example: Allocate memory for a string of 256 characters:
  - number of bytes = `256 * sizeof(char)`
- Returns `void` **pointer** to allocated block of memory on heap
- Allocated memory is **uninitialised**

Type Casts

- Convert one variable type to another type using a **type cast**
- Put the type name to convert to in **parenthesis** before the variable to convert
- **Integer promotion** and **demotion**: values preserved if possible, other implementation specific (often truncated)
- Can also **cast pointers**

```
1       float x = 256.8;
2       int i, *pi;
3       char c;
4       void *pv;
5
6       i = (int) x;
7       c = (char) x;
8       pi = (int *) pv;
```

malloc() Examples 1

```
1  #include <stdlib.h>
2
3  ...
4
5  int *pi = NULL;
6  char *pc = NULL;
7
8  ...
9
10 pi = (int *) malloc(4 * sizeof(int));
11 pc = (char *) malloc(256 * sizeof(char));
```

Using `malloc()` Safely

- The heap (program memory) is not infinite
- `malloc()` returns `NULL` if no more space on heap
- **Always** check value returned by `malloc()` to verify that memory was allocated successfully

```
1  pi = (int *) malloc(4 * sizeof(int));
2
3  if(pi == NULL) {
4      /* error - do something */
5  }
6
7  if(!pi) {
8      /* error - do something */
9  }
10
11 if(!(pi = (int *) malloc(4 * sizeof(int)))) {
12     /* error - do something */
13 }
```

Using Allocated Memory

- Can use allocated memory to store data like any other variable
- Only access it via **pointer** - cannot obtain a variable name for it

```
1  #define NUM_VALS 10
2
3  int i, *p = NULL, *p2;
4
5  if(!(p = (int *) malloc(NUM_VALS * sizeof(int))))
6      exit(EXIT_FAILURE);
7
8  p2 = p;
9  for(i = 0; i < NUM_VALS; i++) {
10     p[i] = 1.0;
11     *p2++ = 1.0;
12 }
```

`free()`

- When finished using dynamically allocated memory, need to release it back to operating system
- Otherwise, memory cannot be used for anything else until program exits
- Use `free()` standard library function to do release memory:

```
void free(void *ptr)
```

- `ptr` is a pointer previously returned by `malloc`
- Accessing memory after `free` will cause a segmentation fault

calloc()

- `calloc()` is similar to `malloc()` - allocate a block of memory:

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

- Except `calloc` will **initialise memory** to **0** - higher **computational overhead**
- Safer to use for arrays (avoids **integer overflow**)

```
1  pi = (int *) malloc(4 * sizeof(int));
2
3  pi = (int *) calloc(4, sizeof(int));
4
5  if(!(pi = (int *) calloc(4, sizeof(int)))) {
6      /* error - do something */
7  }
```

realloc()

- `realloc()` **enlarges** or **shrinks** allocated memory:

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

- New memory **not** initialised
- Original pointer may move
- `free()` called if pointer moves or memory shrunk

```
1  if(!(pi = (int *) malloc(4 * sizeof(int)))) {
2      /* error - do something */
3  }
4
5  if(!(pi = (int *) realloc(pi, 8 * sizeof(int)))) {
6      /* error - do something */
7  }
```

Garbage Collection

- Memory is not infinite
- **Garbage**: memory occupied by objects no longer in use
- **Memory leak**: memory no longer needed is not released
- In C, **you** are the garbage collector
- Must keep track of the dynamic memory you have allocated
- **Release it** back to the program after you have finished with it - but not before - `free()`

Checking for Garbage

- Can use `valgrind` tool to check for correct garbage collection:

```
$ valgrind --tool=memcheck --leak-check=yes
  --show-reachable=yes ./test1
...
==796== ERROR SUMMARY: 1 errors from 1 contexts
        (suppressed: 0 from 0)
```

- Compile your code with the `gcc -g` flag to get line numbers
- Garbage collection will be **assessed** in labs and coursework

String Copy Example 2

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char **argv)
6  {
7       char *buffer, *p;
8       int i;
9
10      if(argc != 2)
11           return 1;
12      if(!(buffer = (char *) calloc(strlen(argv[1]) + 1, \
13                sizeof(char))))
14           return 1;
15
16      p = argv[1];
17      i = 0;
18      while(*p)
19           buffer[i++] = *p++;
20      buffer[i] = '\0';
21      printf("%s\n", buffer);
22
23      free(buffer);
24      return 0;
   }
```

Summary

- **Compile-time** and **run-time** memory
- Program memory **sections/segments**
- Dynamic memory allocation - **heap**
- C standard library functions `malloc()`, `calloc()`, `realloc()`, `free()`
- Type casts
- Garbage collection - `valgrind`

Activities

- Read **K&R Chapters 5.4, 7.8.5, 8.7**