COMPSCI2030 Systems Programming

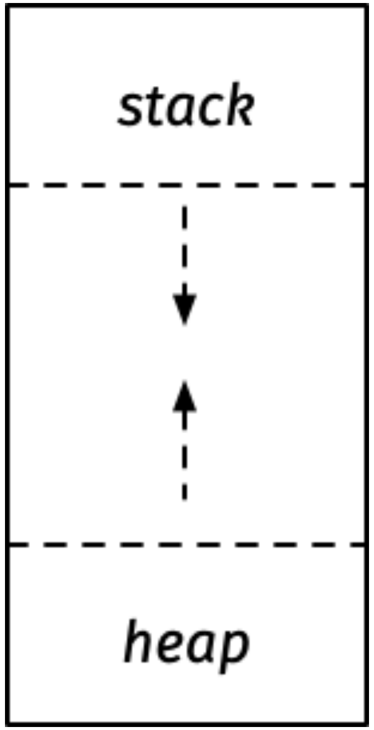
Dynamic Memory Allocation

Yehia Elkhatib

University of Glasgow

Stack vs. Heap memory regions

- So far we have only seen variables with automatic allocation
  - lifetime managed by the compiler
- These variables are stored in the *stack* part of the memory
  - the size of every variable on the stack has to be known *statically*, i.e. without executing the program
- In many cases we do not know the size of a variable statically
  - e.g. dynamically sized arrays
- For such cases we manage the memory manually by dynamically *requesting* and *freeing* memory
- The part of memory used for dynamic allocation is the *heap*
- The stack and the heap share the same address space and grow with use towards each other



stack and heap are in a single address space

Dynamic memory management

All defined in `<stdlib.h>`

- We request a new chunk of memory from the heap

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

- We specify the number of bytes we like to allocate
  - success: it returns a void-pointer to the first byte of the un-initialised block
  - failure: it returns `NULL`; so it is good practice to check directly afterwards
- Other related functions
  - Reallocate a given area of memory

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

  - Allocate an array of num objects of size, initialize all to zero

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

Dynamic memory management

- Memory allocated with `malloc`, etc. <u>must</u> be manually deallocated

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

- If `free` is not called, we *leak* the allocated memory
  - i.e. memory blocks are allocated but not used
- Good practice: set a pointer to `NULL` after freeing it
  - makes it easy to check if you have free’d it or not

Example: Dynamically sized array

- The size of the array is the first number entered by the user

```
#include <stdlib.h>
#include <stdio.h>
int main() {
  printf("How many numbers do you want to average?\n");
  int count;
  if (scanf("%d", &count) == EOF) { exit(-1); }
  // allocate memory based on dynamic input (here its size)
  int* array = malloc(count * sizeof(int));
  for (int i = 0; i < count; i++) {
    int number;
    if (scanf("%d", &number) == EOF) { exit(-1); }
    array[i] = number;
  }
  float sum = 0.0f;
  for (int i = 0; i < count; i++) { sum += array[i]; }
  printf("The average is %.2f\n", sum / count);
  free(array); array=NULL; // free the memory manually after use
}
```

Returning a pointer to a local variable

- It is an easy mistake to return a pointer to a local variable. Never do it!
- Because the pointer has a longer lifetime than the variable it is/was pointing to

```
struct node* create_node(char value) {
        struct node node;
        node.value = value;
        node.next = NULL;
        return &node;
} // lifetime of node ends here...
  // but its address lives on in a_ptr

int main() {
        struct node* a_ptr = create_node('a');
        // ...
} // lifetime of a_ptr (pointing to node) ends here
```

- Solutions
  - Pass the root/leading node, then manipulate it inside the function before node expires
  - Allocate the memory and pass the pointer back

Function Pointers

- Memory does not store data but also program code
- It is possible to have a pointer pointing to code
- These pointers are called *function pointers* and have the type:
  `return_type (*)(argument_types)`
- This is typically used when passing to another function
  - i.e. telling the second function which function to use *at runtime*
  - very useful with design patterns such as command, iterator, visitor
- Example:
  - a sorting function that should compare elements according to their type
  - numbers >; strings `strcmp()`; …

Function Pointers

- Now we can implement a generic print function for a binary tree:

```
void print_string(const void * value_ptr) {
  char * string = value_ptr; // by changing the type we give the bits meaning
  printf("%s\n", string);
}
void print(node * root, void (* print_function)(const void *) ) {
  if (root) {
    print_function(root->value_ptr);
    print(root->left_child, print_function);
    print(root->right_child, print_function); }
}
int main() {
  node * root = ... ;
  print(root, print_string);
}
```

- Function names are automatically converted to function pointers
  - i.e. we do not have to write `&print_string`

Memory Management Challenges

- When we allocate memory with `malloc`, we are responsible for calling `free`
- We must call `free` *exactly* once for each malloc'd address
- Good practice: assign `NULL` value to pointers that have been free'd
- But this does *not* prevent all double `free` errors, e.g.:

```
int main() {
  void * ptr1 = malloc(sizeof(int));
  void * ptr2 = ptr1;
  free(ptr1); ptr1 = NULL;
  free(ptr1); // OK. Calling free with NULL is fine
  free(ptr2); // *** error for object 0x7f9355c00690: pointer being freed
              // was not allocated
}
```

Memory Management Challenges

- Another problem are *dangling pointers*: these point to locations that have been free'd

```
int main() {
  node * left_child = create_tree("a", NULL, NULL);
  node * root = create_tree("b", left_child, NULL );
  destroy_tree(left_child); // now: root->left_child points to freed memory!
}
```

Memory Management Challenges

- Inconspicuous memory leaks

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

int main() {
  char *mem = (char*) malloc( sizeof(char) * 20); // allocate some memory
  if (!mem) { exit(EXIT_FAILURE); } // check if allocation went fine
  // use allocated memory

  mem = (char*) malloc( sizeof(char) * 30); // allocate more memory
  // …BUT we just lost the pointer to the old memory => LEAK!

  // use newly allocated memory

  free(mem); // free newly allocated memory
}
```