

CMPT 125

**Introduction to Computing Science
and Programming II**

September 27, 2021

A bit more on syntax:

Return values and conditions

Return values and conditions

- All command in C return a value. (Exception: void functions)

- Examples:

```
printf("%d\n" , 3 < 5); // prints 1  
printf("%d\n" , 3 == 5); // prints 0
```

if statements:

```
if (cond)  
    do_something();  
else  
    do_something_else();
```

Equivalent to:
if (cond != 0)

while statements:

```
while (cond)  
    do_something();
```

Equivalent to:
while (cond != 0)

Multiple conditions

AND of conditions:

```
if (cond1 && cond2)
    do_something();
else
    do_something_else();
```

&& - and

- checks first if cond1 is satisfied (i.e., cond1 !=0)
- runs cond2 **only** if cond1 is satisfied

OR of conditions:

```
while (cond1 || cond2)
    do_something();
```

|| - or

- checks first if cond1 is satisfied (i.e., cond1 !=0)
- runs cond2 **only** if cond1 is **not** satisfied

Recall the example: while (i < len && !found)

Global variables vs Static variables

Global variables

So far we have seen only variables defined inside functions. The scope of the variables is limited only to the function.

It is possible to define a **global variable** that is visible everywhere.

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

```
int counter = 0; // init the global variable to zero
```

```
int main() {  
    printf("global counter %d\n", counter); // prints 0  
    counter++;  
    printf("global counter %d\n", counter); // prints 1  
    int counter = 0; // local variable  
    printf("local counter %d\n", counter); // prints 0  
    return 0;  
}
```

Static variables

It is also possible to define a **static variable** that will "remember" its value in different calls of the function.

```
#include <stdio.h>

void test_static_count() {
    static int count = 0; // initialized only once!
    // do something...
    count++;
    printf("count = %d\n", count);
}

int main(void) {
    test_static_count(); // prints "count = 1"
    test_static_count(); // prints "count = 2"
    test_static_count(); // prints "count = 3"
    ...
}
```

Macros

Using macros: #define

`#define` creates a constant that can be use globally.

Macros are not variables. Cannot be changed by the program.

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

```
#define COURSE_NAME "CMPT125"
```

```
#define PI 3.1415925
```

```
int main() {  
    printf("%f\n", PI); // prints 3.1415925  
    printf("%s\n", COURSE_NAME); // prints CMPT125  
    printf("PI\n"); // prints PI  
    ...  
}
```

Using macros: #define

`#define` macros are simply textual substitutions.

Preprocessor replaces the occurrences of the macros before compiling the code.

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

```
#define MY_FRAC 70/14
```

```
#define SQR(a) a*a
```

```
int main() {
```

```
    float x = MY_FRAC; // replaced by float x = 70/14;
```

```
    int y = SQR(5); // replaced by int y = 5*5;
```

```
    int z = SQR((5+2)); // replaced by int z = (5+2)*(5+2);
```

```
    int w = SQR(5+2); // replaced by int z = 5+2*5+2;
```

```
    ...
```

Type casting in C

Type casting

- C allows conversions between different types of variable.
- Done when one data type can be changed to a different data type.
- Example:
 - int to float
 - short to int
 - int to long
- We can also type cast the result to make it of a particular data type.
- Need to be careful. Information may be lost!
- Examples:
 - float to int
 - int to char
 - char* to int*

Memory allocation

Memory allocation

- Motivating example:
Write a program that gets from the input
 - a positive number N
 - N integer numbersPrints the numbers in the reverse order
- We don't know the size of the array in advance.
- Need **dynamic** memory allocation

- *(void*) malloc(size_t)*

Casting

// allocates size_t consecutive bytes
// size_t alias for unsigned integer type

Allocating
n ints

- Usage: *int* array = (int*) malloc(n * sizeof(int));*

Memory allocation

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

`(void*) malloc(size_t)`

// allocates `size_t` consecutive bytes

// `size_t` alias for unsigned integer type

Pointer without type.
Requires casting

The return type of `malloc()` is `(void*)`

`malloc()` returns `NULL` if allocation fails.

NULL pointer does
not point to an address

Usage: `int* arr = (int*) malloc(n * sizeof(int));`

After we are done, **we must release the memory** using `free()`.

Usage: `free(arr);`

Memory allocation

Another example:

Write a function that gets a parameter *int n* and returns the array of *ints* of length *n* with $array[i] = i^2$

Note the difference:

```
int* foo() {  
    int arr1[15];  
    int* arr2 = (int*) malloc(15 * sizeof(int));  
    ...  
}
```

arr1 is a local variable!

arr2 can be returned, and used outside of *foo()*!

More on scope of variables soon...

Memory allocation

unsigned int

void realloc(void* ptr, **size_t** size)* - expanding or contracting the existing area pointed to by *ptr*, by copying to new location.

If fails, returns *NULL*.

If the area is expanded, copies the previous memory to the new area. The contents of the new part of the array are undefined.

If the area is contracted, it copies the first *size* bytes.

Memory allocation

void memcpy(void* dest, const void* source, size_t num)*
// copies the values of num bytes from source to dest.

void memset(void* ptr, int val, size_t num)*
//sets the first num bytes in the memory to be val.

void calloc(int num_of_items, int size)*
// allocates num_of_items objects of the specified size
// in the memory and sets the memory to zero.
// Essentially equivalent to
void ptr = malloc(num_of_items*size);*
*memset(ptr, 0, num_of_items*size);*

Memory allocation

When we are done, we release the allocated memory using `free()`.

If not, the OS will think that the memory is still in use.

This is called memory leak

More on this next.

Questions so far?

Execution stack and scope of variables

Execution stack and scope of variables

An execution stack (call stack, run-time stack...) is a data structure that stores the information about the functions called during the execution of a program.

For each function the stack stores the following information:

- parameters of the function


- local variables

- return value


- return address: When a function completes,
it returns control to the function that called it.

Execution stack and scope of variables


```
□ int bar(int size) {  
    int i = size+1;  
    return i;  
}
```



```
int foo(int n) {  
    int ret = 3;  
    bar(4);  
    bar(8);  
    return ret;  
}
```



```
int main() {  
    foo(5);  
    return 0;  
}
```



bar()
params: int size = 8;
variables: int size = 8; int i = 9;
return value: ?
return address: ...

foo()
params: int n = 5;
variables: int n = 5; ret = 3;
return value:
return address: 0x9378ad

main()
params: --
variables: ...
return value: ?
return address: 0x128fbad

Execution stack and scope of variables

All local variables of a function are stored in the corresponding stack-frame. When the function completes, the variables become unavailable.

Variables obtained from `malloc()` are stored in "global memory", and do not die when the function completes.

We must free the memory when we don't need them anymore.

Terminology:

Local variables are stored on the *stack*.

Global variables are stored on the *heap*.

Stack (data structure)

Stack is a data structure with two principal operations:

Push(element) – adds an element to the collection

Pop() - removes the most recently added element from the collection

Example: Stack of blocks

we add and remove only to/from the top.

Same with function calls:

Each function call adds a block to the stack.

When finished, we remove it from the stack and continue to the next function on top.



Functions and structs

Functions and Structs

Note that struct is treated similarly to any other type (int, char)

Example:

Consider the function

```
person get_person() {  
    person p;  
    ...  
    return p;  
}
```

It returns the struct by copying p.data1, p.data2... to the caller.

More examples in struct_ret.c and struct_ret_ptr.c

Pointers to Functions

Function Pointers

In C functions are stored in the memory just as any other data.

And we can use pointers to refer to them. [A crazy idea, but why not]

The syntax is:

```
int foo(char c, int n, char* s) {...}
```

```
int (*my_function_ptr)(char, int, char*);
```

```
my_function_ptr = foo;
```

```
my_function_ptr('A', 10, str);
```

See `function_pointers.c`

We did not cover...

What we did not cover

Union – similar to struct except that all data fields share the same memory. For example:

```
union op {  
    char op_c;  
    int op_a;  
    int op_b[3];  
};
```

will have the size of 3 ints. If you modify op_a it may change op_c

File I/O – Your assignment 2 will have a question about file I/O.

You will have to read about it yourself.

**We are done with the
syntax of C!!!**

Questions?
Comments?