C functions

CMSC 313
Raphael Elspas

UMBC

Calling a C function from assembly

- We can call a C function using the C Calling convention shown in the table below:
- These functions can be written by us or be built in (like printf, scanf, etc.)
- Our assembly code would be the caller, so we would need to follow caller rules

x86-64 C declaration
The 64 bit method places the parameters 1-6 in the registers RDI, RSI, RDX, RCX, R8, R9, and all parameters after that get pushed to the stack.

Main function

- For the main subroutine to call another function, main *itself* needs to be setup as subroutine
- This means main needs to:
    - Setup base pointer in the beginning
    - Return at the end

```
main:
    push rbp
    mov rbp, rsp
    ...
    pop rbp
    ret
```

Extern

- In the assembly caller, we need to use the keyword “extern” to alert the linker to look for the function outside of the current assembly file
- We also need to use this keyword if we want to call an assembly subroutine in a different file
- Example: the left asm code calls the C function below with the following parameters:

```
printf(“Hello, world! I like %d”, 42)
```

```
section .data
    format db "Hello, world! I like %d", 0
    num dq 42

section .text
    extern printf
    global main

main:
    push rbp      ; note that if you use c calls
    mov rbp, rsp ; you have to setup main like
                  ; subroutines with rbp and rsp

    ; Pass the format string to printf
    mov rdi, format
    mov rsi, [num]
    call printf

    ; Exit the program
    pop rbp      ; note exiting program can also
    mov rax, 0   ; be done with setting rax and
    ret          ; then ret
```

Call a subroutine from another asm file

- We have to use **extern** in the caller, and **global** in the callee

File A

```
section .data
    msg db 'Hello, world! I like', 0
    len equ $ - msg
section .text
    global main
main:
    ; Call the subroutine in Program B
    mov rsi, msg   ; Argument
    mov rdx, len
    call subroutine

    ; Exit the program
    mov rax, 60   ; Syscall number for exit
    xor rdi, rdi ; Return 0
    syscall

extern subroutine
```

File B

```
section .text
    global subroutine

subroutine:
    ; takes one argument in rdi reg
    ; print argument
    mov rax, 1      ; Syscall number for write
    mov rdi, 1      ; File descriptor 1 (stdout)
    ; rsi already set, Pointer to the argument
    ; rdx already set, Length
    syscall

    ret  ; Return from subroutine
```

Call c file from asm

- You have an assembly file **hello.asm**, and a c file, **world.c**, which you want to link.
- You can do the following in the terminal:

This step compiles a C file into an **object** file.

```
$ nasm -f elf64 hello.asm
$ gcc -c world.c
$ gcc -m64 -o helloworld hello.o world.o
```

C and C++ differences

Feature	C	C++
Programming Paradigm	Procedural	Multi-paradigm (procedural, object-oriented, generic)
Header Files	.h extension	.h or .hpp extension
Function Overloading	Not supported	Supported
Classes and Objects	Not supported	Supported
Inheritance	Not supported	Supported
Encapsulation	Achieved through structs	Achieved through classes
Namespace	Not supported	Supported
Templates	Not supported	Supported
Exception Handling	Not supported	Supported
Standard Libraries	Limited standard library	Standard Template Library (STL) and additional features
Memory Management	Manual memory management using malloc() and free()	Supports manual and automatic memory management; features like new and delete
Usage	System programming, embedded systems, low-level programming	Application development, game development, systems programming

Datatypes

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 or 8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	4 or 8 bytes	0 to 18446744073709551615

Note that some datatypes such as an **int** or **long** may have variable dimensions since they depend on the architecture that is using that code.

Fixed width types

- You can make sure your datatype has a certain dimension by using a fixed width data type. To use these you need to use `include <stdint.h>`.

Type	Storage size
int8_t, uint8_t	1 byte
int16_t, uint16_t	2 byte
int32_t, uint32_t	4 byte
int64_t, uint64_t	8 bytes

Boolean

- No boolean type available by default.
- You have to include a header: **include <stdbool.h>**
- You use the type: **bool**

```
#include <stdbool.h>
bool israining = true;
if (isRaining){
   useUmbrella();
}
```

Ternary Statement

- Aa ternary is a shorthand for an else-if statement
- The format is (condition) ? (result if true) : (result if false)

```
int larger = x > y ? x : y
```

Binary and logical operators

Operator symbol	Meaning
>>	Right shift
<<	Left shift
|	Bitwise or
&	Bitwise and
^	Bitwise xor
&&	Logical and
||	Logical or

Printf

- To print text to stdout, use **printf**.
- Syntax: printf(format, arg1, arg2, ….)
- Example:

```
int num = 3
printf("value = %d\n", num)
```

- Format is a string containing content to be printed, but also contains conversion specifiers.

UMBC

Printf format specifiers

Operator symbol	Meaning
%d	Print integer as decimal
%u	Print integer as unsigned number
%s	Print string
%f	Print float
%x	Print integer as hexadecimal
%c	Print integer as ascii character
%p	Print pointer as hex

Scanf

- Use scanf to store input from the user
- Syntax: scanf(format, arg1, arg2, ….)
- Example:

```
int num;
printf("input a number:");
scanf("%d", &num);
```

- Specifiers are similar to those used in printf
- Format is a string containing content to be printed, but also contains conversion specifiers specifying how the data taken in will be interpreted. The values are stored in the args input.

Files in c

- Open a file with fopen(), close a file with fclose().
- Use FILE * datatype to capture handle to open file

```
FILE *myFile
myFile = fopen(“bob.txt”,”r”);
If (myFile == NULL){
  /*handle error*/
}
```

- Fopen() requires the way the file is being opened.
  - “r” for read only
  - “w” for creating the file and writing, deleting existing file
  - “a” for appending to file

Arrays

- Array syntax follows the pattern:

```
type array_name[size]
```

- What is the value at arr[1] in the example below?

```
int arr[10];
arr[0] = 14;
arr[3] = -3;
```

- It is undefined, and will have a garbage value

Strings

- Strings in C are arrays of type **char**.

```
char str[30] = "hello";
```

- In the background what is happening is:

```
char str[0] = 'h';
char str[1] = 'e';
char str[2] = 'l';
char str[3] = 'l';
char str[4] = 'o';
char str[5] = 0;   //null character
```

Struct

- C++ has objects to store structured data and to group different datatypes together.
- C uses **structs**

```
struct tag {
   member1_declaration;
   member2_declaration;
   member3_declaration;
…
   memberN_declaration;
};
```

- Where struct is the keyword, tag names the struct, member#_declaration are variable declarations which define the members of the struct

Struct example

- We can access members of a struct with “.”
- We can initialize values in a struct in two ways:

```
struct point {
   int x; //x coordinate
   int y; //y coordinate
};

struct point p1;
p1.x = 5;
p1.y = 6;
```

```
struct point {
   int x; //x coordinate
   int y; //y coordinate
};

struct point p1 = {5,6};
```

Struct with array members

- Defined the same way an array is defined
- We might define a particle like this:

```
struct particle
{
    double p[3]; // Position
    double v[3]; // Velocity
    double a[3]; // Acceleration
    double radius;
    double mass;
};
```

Array of struct

- Creating an array of structs first requires a struct definition, and then a declaration of the array

```
#define n 3

struct particle
{
    double p[3]; // Position
    double v[3]; // Velocity
    double a[3]; // Acceleration
    double radius;
    double mass;
};

struct particle particles[n];
```

Dynamic memory functions

- void *malloc(size_t nrBytes);
  - Returns pointer to uninitialized memory of size nrBytes or NULL if request cannot be made
- void *calloc(int nrElements, size_t nrBytes);
  - Same as malloc, but memory is initialized to 0.
  - Parameters divided into number of elements, and element size.
- void free(void *p);
  - Deallocates memory pointed to by p

- **void *** is a generic pointer that can point to any kind of data
- **size_t** is an unsigned integer type that should be used instead of int when identifying the size of something

Pointer notation

- A pointer is described with a *
- The dereference operator is also a *
- The opposite of dereferencing, or “find the address of” uses the operator “&”

Variable length arrays

- In C99, you can create variable length arrays.
- This will be allocated on the stack:

```
int n = 5
int arr[n]
```

- In all versions of C, you can create variable length arrays with malloc()
- arr becomes a pointer to a spot in memory that you can dereference.
- You need to free() the memory at the end of the program.
- Use sizeof() to allocate correct number of bytes per datatype

```
int *arr = malloc(sizeof(int)*5);
...
free(arr)
```

Structs with variable length array members

- Use pointers as members and malloc them when you instantiate your struct.

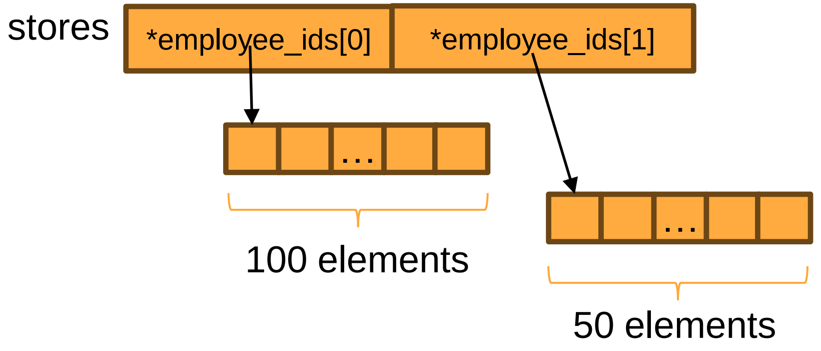
```c
struct store{
    int *employee_ids; //array of ints
};
struct store stores[2];
stores[0].employee_ids = malloc(sizeof(int)*100);
stores[1].employee_ids = malloc(sizeof(int)*50);
```

- What does the memory diagram look like for this?

Memory diagram

- Use pointers as members and malloc them when you instantiate your struct.

```
struct store{
    int *employee_ids; //array of ints
};
struct store stores[2];
stores[0].employee_ids = malloc(sizeof(int)*100);
stores[1].employee_ids = malloc(sizeof(int)*50);
```



Arrays of Strings

- Under the hood is an array of arrays
- We can use several notations to describe an array of strings, since a string is an array itself.

```
char **names = malloc(sizeof(char *) * 3)
Names[0] = "asdf";
```

←Also called a double pointer

```
char *names[] = {"asdf", "asdf2"};
```

```
char names[][] = {"asdf", "asdf2"};
```

←Will fail, multidimensional arrays must have bounds for all dimensions but the first

typedef

- You can use the following syntax for struct shorthand so you don't have to use "struct" keyword before every usage.

```c
typedef struct {
  int x, y;
} Point;

Point a;
a.x = 5;
a.y = 7;
```

Instead of

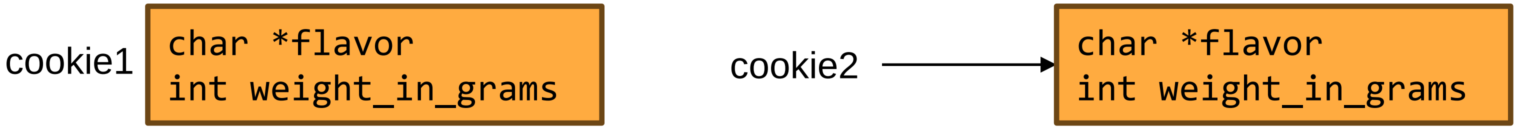
```c
struct Point{
  int x, y;
};

struct Point a;
a.x = 5;
a.y = 7;
```

Consider *struct* vs *pointer to struct*

- I have a cookie struct which contains a flavor string and a weight in grams

```
typedef struct{
    char *flavor;
    int weight_in_grams;
} cookie;
cookie cookie1;
cookie *cookie2 = malloc( sizeof(struct cookie) );
```



-> operator

- When accessing members of a struct, we use two operators:

```
typedef struct{
    char *flavor;
    int weight_in_grams;
} cookie;
cookie cookie1;
cookie *cookie2 = malloc( sizeof(struct cookie) );
```

operator	behavior	example

. (period)	Access member of struct	`cookie1.flavor = “chocolate”`
		`cookie1.weight_in_grams = 55`

-> (arrow)	Access member of pointer to struct	`cookie2->flavor = “chocolate”`
		`cookie2->weight_in_grams = 55`

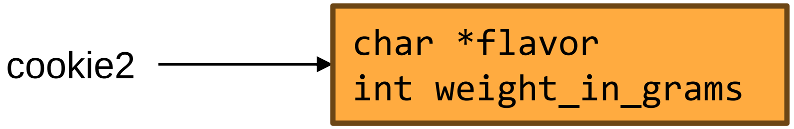
-> operator continued

- The -> operator technically means: dereference this struct, then access this member.
- Therefore the following two are equivalent:

```
cookie2->flavor
```

=

```
(*cookie2).flavor
```



Pass by reference or value

- When a variable is passed by value, a copy of itself is pushed to the stack.
- When a variable is passed by reference, a copy of it's address is pushed to the stack

```
Typedef struct{
    int count;
    char *type;
} cookiejar;
```

Passed by value

```
cookiejar jar = {5, "oats"}

addOne(cookiejar jar){
   jar.count += 1;
}

addOne(jar)
//jar = {5, "oats"}
```

The copy passed in is changed

Passed by reference

```
cookiejar jar = {5, "oats"}

addOne(cookiejar *jar){
   jar->count += 1;
}

addOne(&jar)
//jar = {6, "oats"}
```

The original is changed

What’s wrong with this code?

```
#include <stdio.h>
int main(){
    int *p;
    *p = 200;
    printf("The value is %d\n", *p);
    return 0;
}
```

What’s wrong with this code? Solution

- This is wrong, and it might generate a segmentation fault error.
- Why? We need p to be associated with an area of memory that is valid.

```
#include <stdio.h>
int main(){
    int *p;
    *p = 200; /*This is wrong!*/
    printf("The value is %d\n", *p);
    return 0;
}
```

What’s wrong with this code? Fix

- A quick fix is to initialize a variable, and assign p to the memory address of that variable.

```c
#include <stdio.h>
int main() {
    int *p;
    int x;
    p = &x;
    *p = 200;
    printf("The value is %d\n", *p);
    return 0;
}
```

What’s wrong with this code?

```
#include <stdio.h>
int* process() {
    int x = 10;
    int* p = &x;
    return p;
}
```

What’s wrong with this code? Solution

- During a function call, some memory might be allocated. But when the function returns, that memory is either deallocated, or otherwise lost. In this case the stack frame is no longer in scope.

```
#include <stdio.h>
int* process() {
    int x = 10;
    int* p = &x;
    return p; /*we’re returning a pointer to
                an area that no longer exists*/
}
```

What’s wrong with this code? Fix

- If we dynamically allocate the memory, the memory will be added to the heap and therefore will be available outside of the current stack frame.

```
#include <stdio.h>
int* process() {
    int* p = malloc(sizeof(int));
    *p = 10;
    return p;
}
```

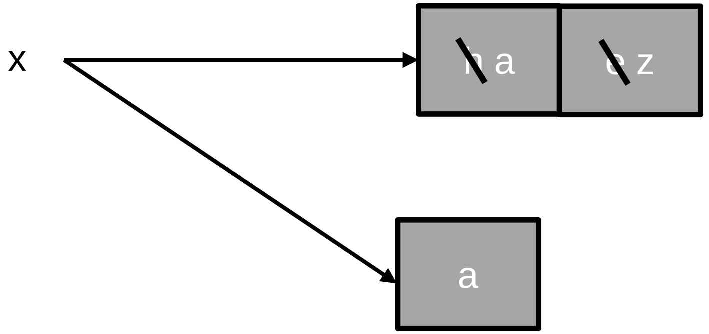
Const modifier

- The const keyword identifies something that will not change and will throw an error if changed. For example:

```
int main() {
    const int x = 10;
    x = 12; //this will throw an error
    return 0;
}
```

Const pointers

- There are several ways a pointer can be constant.
- The data pointed to can remain constant or be changed, but the pointer itself can be constant, or point to a different location



Maybe I can change the values

Or Maybe I can change where I’m pointing to

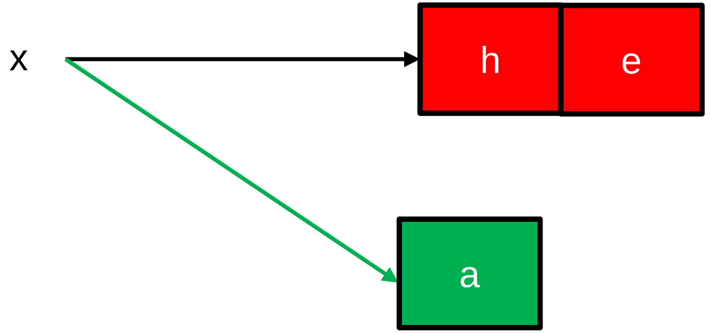
I need to specify what will be constant

Const pointers, part 1

- There are several ways a pointer can be constant.
- First, if I want to make a char pointer const, I use the format:

```
const char * x
```

- This makes the value pointed to const, but the pointer itself modifiable.



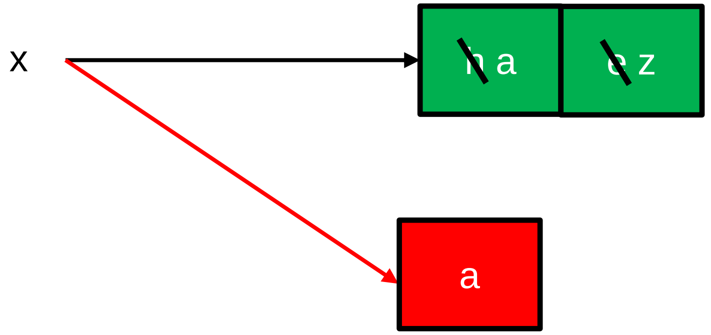
I cannot change the values pointed to …

But I can change where I point to

Const pointers, part 2

- If I want to make the pointer const, but the value it points to modifiable, then I use the following:

```
char * const x
```



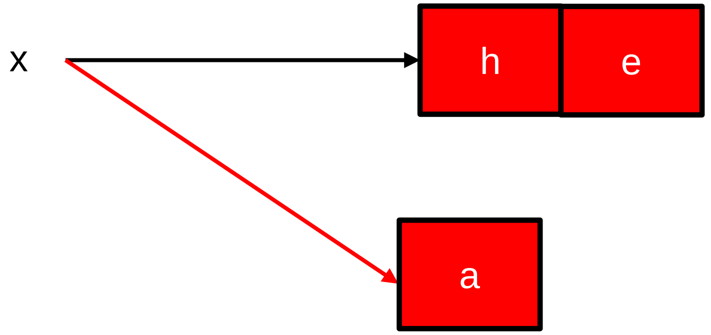
I can change the values pointed to …

But I cannot change where I point to.

Const pointers, part 3

- If I want to make the pointer const, **and** the value it points to const, then I use the following:

```
const char * const x
```



I cannot change the values pointed to …

and I cannot change where I point to.

Review

```
const char * const x
```

Description of what’s pointed to. In this case, we’re pointing to a “const char”

Description of the pointer itself

References

- Stack overflow is your friend