



Systems and Internet Infrastructure Security

Network and Security Research Center
Department of Computer Science and Engineering
Pennsylvania State University, University Park PA

CMPSC 447 ***C Language Review***

Trent Jaeger

*Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University*

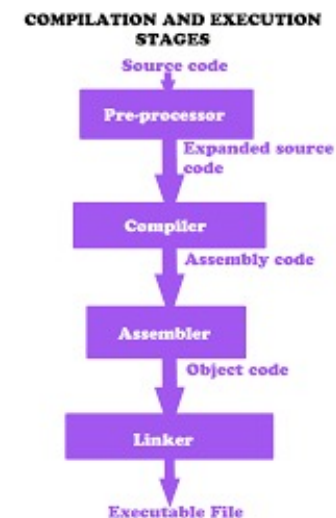
C Program Files

- Program code: [gdb_demo.c](#)
- Headers, if any – why header files?
 - None in this case
 - What is typically in a header file?



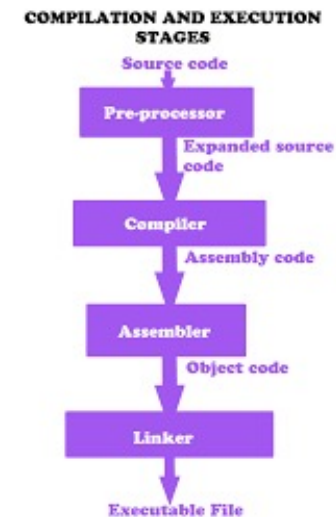
C Program Compilation

- Program code: `gdb_demo.c`
- Compiler options: `gcc` and `clang`
- What is the command line to compile `gdb_demo.c` using `gcc` or `clang`?



C Program Compilation

- Program code: `gdb_demo.c`
- Compiler options: `gcc` and `clang`
- What is the basic command line to compile `gdb_demo.c` using `gcc` or `clang`?
 - `gcc -o gdb_demo gdb_demo.c`
 - `clang -o gdb_demo gdb_demo.c`
- We will use `clang` in this course
 - `-Wall` – enable all compiler warnings



- People refer to C “primitives” meaning the primitive data types in the C language
 - ▶ **Int**: Integers
 - ▶ **Char**: Character
- These are the main ones
 - ▶ Integers may vary in their size (memory used)
 - ▶ Also, floats (various types)
- How do we use these?

- Integers are represented in a small number of bytes
 - Architecture dependent
- Say, four bytes
 - What is the largest number we can represent?
- Integers may be signed or unsigned
 - **Signed (default)**: highest order bit represents the 'sign'
 - **Unsigned**: 0 to max
- Can switch between signed and unsigned at will
 - Compiler may generate a warning

Pass by Value or Reference

- You can pass integers from one function to another as a parameter
 - ▶ `void main () { int x=7; foo(x); printf(“%d\n”, x); }`
- What is the value is printed?
- What if `foo` is supposed to modify the value of `x`?

Pass by Value or Reference

- You can pass integers from one function to another as a parameter
 - ▶ *void main () { int x=7; foo(x); printf(“%d\n”, x); }*
- What is the value is printed?
- What if *foo* is supposed to modify the value of *x*?
 - ▶ Pass a reference to *x* instead
 - ▶ The code above passes *x* “by value”
 - ▶ *void main () { int x=7; foo(&x); printf(“%d\n”, x); }*
 - ▶ Is said to pass *x* “by reference” – What’s the difference?

- The “**char**” data type is a **one-byte entity**
 - Often used to represent text (e.g., ascii characters)
 - But you can use it for any one-byte data
- You can represent a sequence of chars (or ints) as an array – *char name[25];* – an array of 25 one-byte chars
 - Could be a “**string**” (or not)
 - Can you identify what differentiates a string from an array of characters in general?
- Working with strings in C can be tricky

Int and Char Pointers

- A key concept in the C language is the **pointer**
 - A pointer is a memory reference associated with a type
- Can define or create a pointer
 - `int x, *y; y = &x;`
 - A pointer to `y` declared “`int *y`” and a pointer to `x` is created by “`&x`” and the value of the pointer to `x` is assigned to `y` “`y=&x`”
 - Got it?
- Pointers just store the memory location of the referenced object (`x` in “`y=&x`”)

Int and Char Pointers

- A key concept in the C language is the **pointer**
 - A pointer is a memory reference associated with a type
- What does a variable “z” of type “char *” mean?
 - `char *z;`

Int and Char Pointers

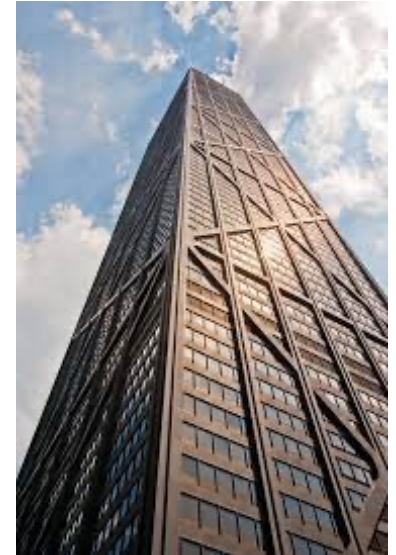
- A key concept in the C language is the **pointer**
 - A pointer is a memory reference associated with a type
- What does a variable “z” of type “char *” mean?
 - Depends on the context
 - Could reference a single char “char *z, a; z=&a;”
 - Or could reference an array “char z[25];”
 - E.g., “char a[25], *z; z = a;”
 - Why no “&” here?
- And “z” may or may not actually reference a string

Pointer Arithmetic

- Can manipulate pointers as values
 - `char *z; z++;`
- What does that do?
- What if `z` is a pointer to an "int" type?

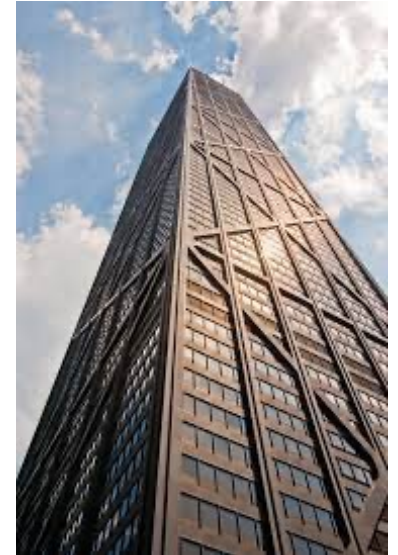
Structures

- The C language permits the definition of user-defined types as structures
 - ▶ Permits the programmer to define memory layouts for related data
 - ▶ *struct x { int a; char *b; };*
 - ▶ What does this structure look like in memory?



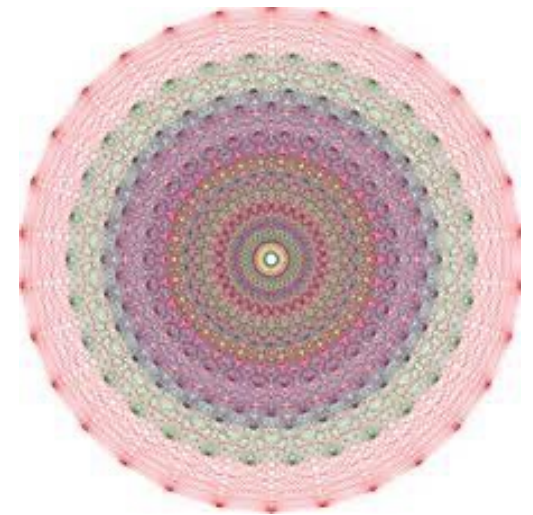
Structures

- What does this structure look like in memory?
 - ▶ *struct x { int a; char *b; };*
- Assuming four-byte integers and pointers: four bytes for “a” followed by four-bytes for the pointer “b”
 - ▶ Where is the memory referenced by “b”?
 - ▶ Is the type “struct x” stored anywhere?



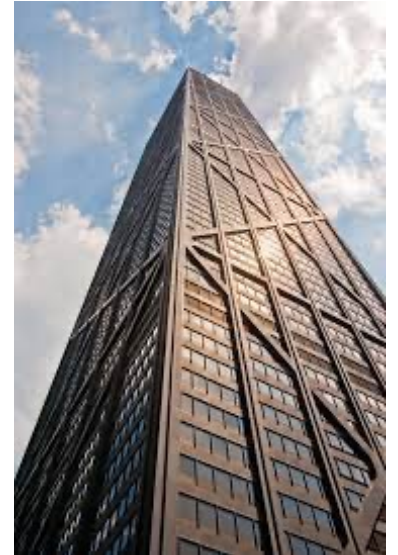
Structures

- A more complex structure: gdb_demo.c
 - typedef struct tree_s {
 - int val;
 - struct tree_s* left;
 - struct tree_s* right;
 - } tree_t;
- Here there are pointers to two fields: *left* and *right*, referencing structures of the same type



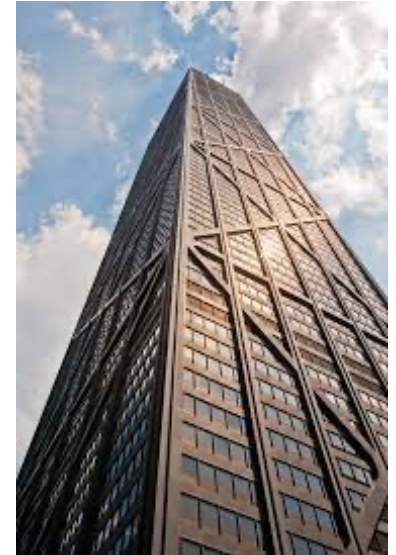
More Structures

- What does this structure look like in memory?
 - ▶ *struct x { int a; char str[4]; tree_t *t; };*



More Structures

- What does this structure look like in memory?
 - ▶ *struct x { int a; char str[4]; tree_t *t; };*
- Here the four-byte integer is followed by a four-byte field and then a four-byte pointer



Type Casting

- C allows for programmers to change the type associated with a pointer
 - Called **type casting**
- What happens when we cast a pointer of type “struct x” to a pointer of type “tree_t”?
 - `struct x { int a; char str[4]; tree_t *t; };`
 - `struct x *ptr, x1; ptr = &x1;`
 - `tree_t *tptr; tptr = (tree_t *)ptr;`
- This is legal in C!



- What happens when we cast a pointer of type “struct x” to a pointer of type “tree_t”?
 - ▶ *struct x { int a; char str[4]; tree_t *t; };*
 - ▶ *struct x *ptr, x l; ptr = &l;*
 - ▶ *tree_t *tptr; tptr = (tree_t *)ptr;*
- In this case, field “a” and “t” are unchanged
 - ▶ But the field “str” is converted to a tree_t pointer “left”
 - You can do this all you want! (you will see in Project I)
 - And the structures need not even be the same size

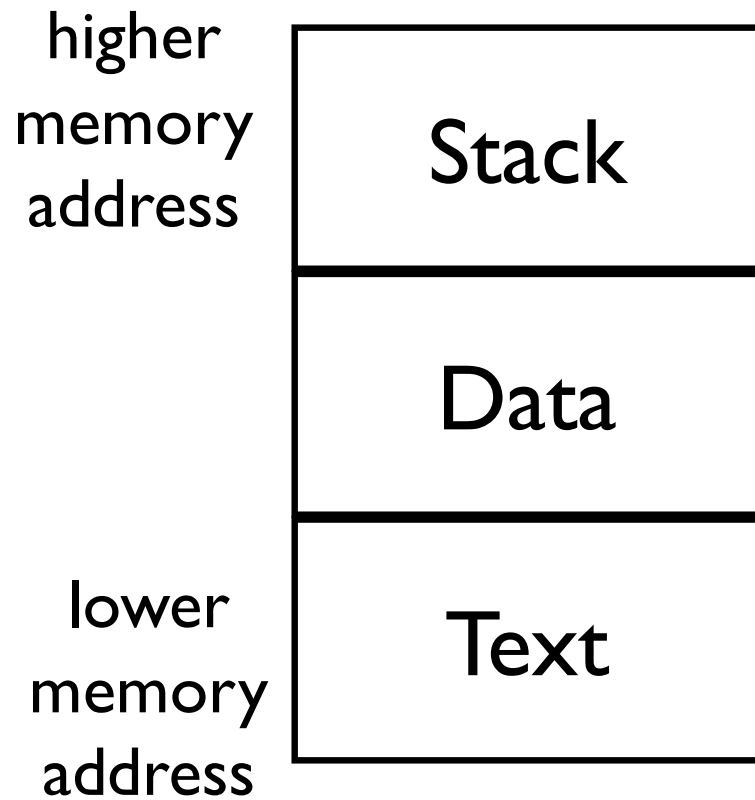
Function Pointers

- C allows you to specify pointers to code as well as data in your programs – typically referring to the start of a function although not required
 - ▶ Called **function pointers**
 - ▶ Who has used function pointers?
- Suppose we have a function – *int foo(int x)*
 - ▶ Can declare a variable to reference such a function
 - ▶ `int(*fnptr)(int) = foo;`
 - ▶ What's the variable declared here?

Function Pointers

- C allows you to specify pointers to code as well as data in your programs – typically referring to the start of a function although not required
 - ▶ Called **function pointers**
 - ▶ Who has used function pointers?
- Suppose we have a function – *int foo(int *x)*
 - ▶ Can declare a variable to reference such a function
 - ▶ `int(*fnptr)(int *) = foo;`
 - ▶ **And invoke: `int a=7, b=0; b=(*fnptr>(&a);`**

Process Address Space



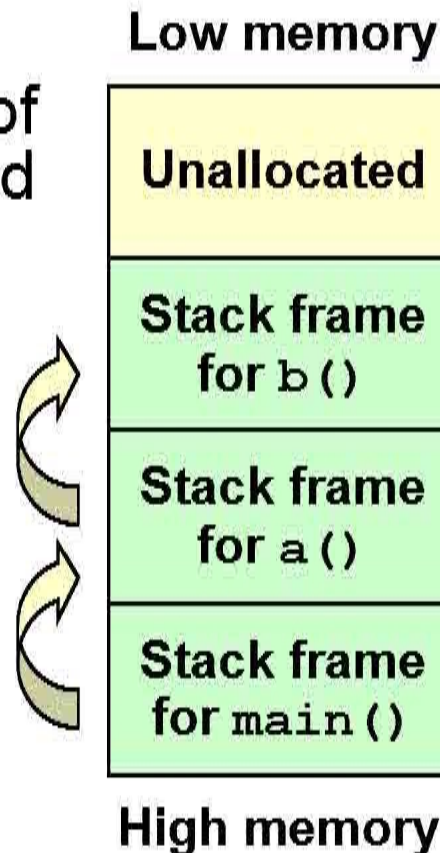
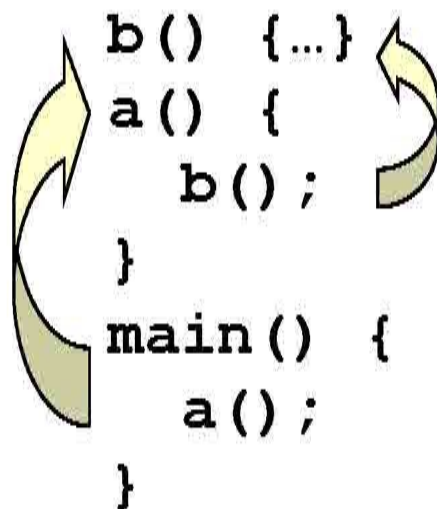
- **Text**: static code
- **Data**: also called heap
 - static variables
 - dynamically allocated data (malloc, new)
- **Stack**: program execution stacks

Stack Segment

The stack supports
nested invocation calls

Information pushed on
the stack as a result of
a function call is called
a frame

```
    b() {...}
  a() {
    b();
  }
main() {
  a();
}
```



A stack frame is
created for each
subroutine and
destroyed upon
return.

*Slide by Robert Seacord

Stack Frames

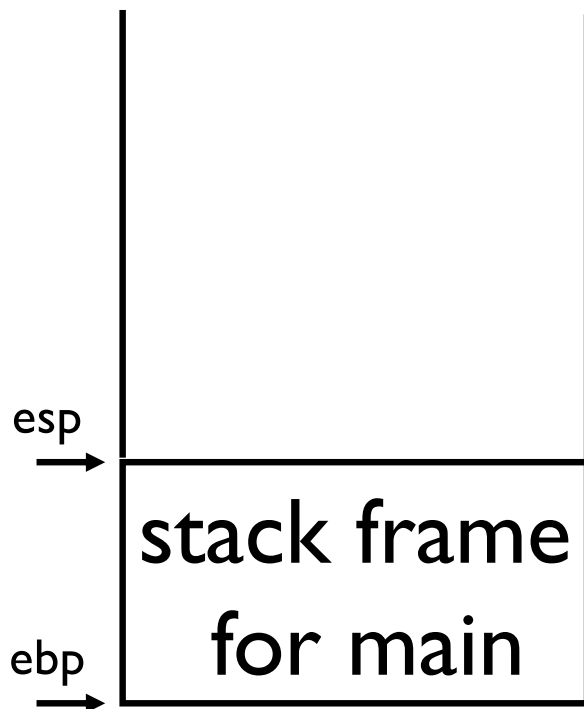
- Call stacks store both program data and program runtime information
 - Return addresses of functions
 - Reference to prior stack frame
- Suppose we have the code
 - `int main() { fn(1, 2); exit(0); }`
- What would the stack look like when "fn" is called, initializes a local variable "var", and returns?
 - **Keep in mind:** stack grows downward

Stack Frames

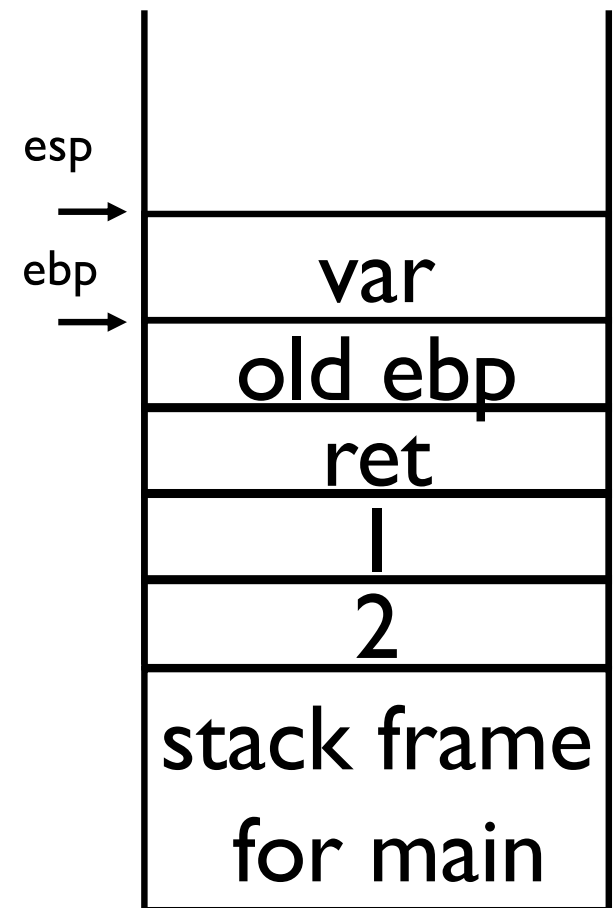
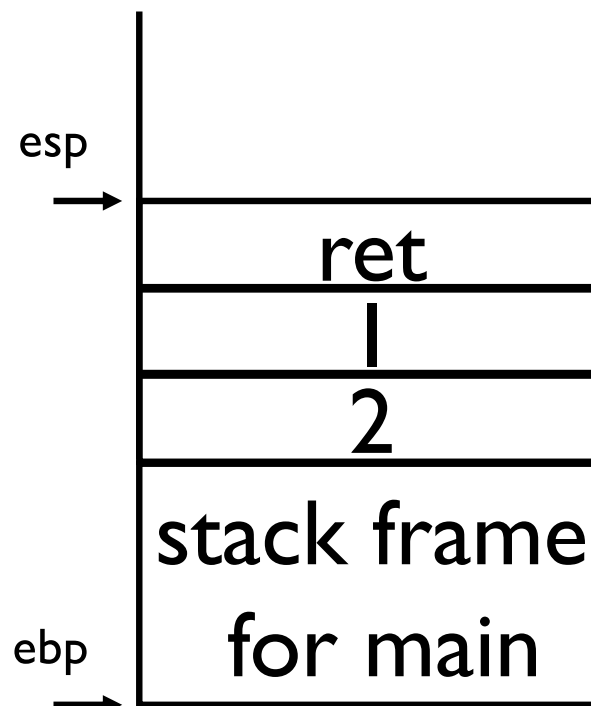
- Call stack sequence

At invocation

Before call

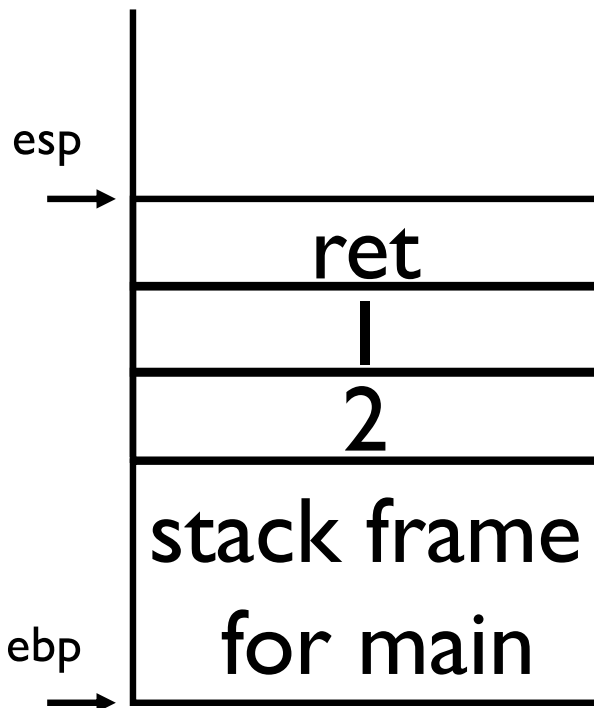


At call

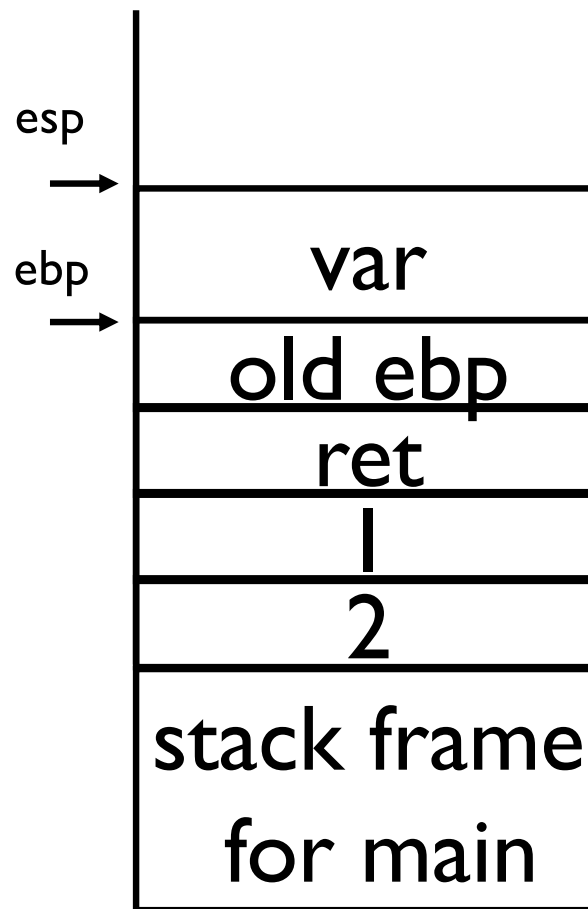


Stack Frames

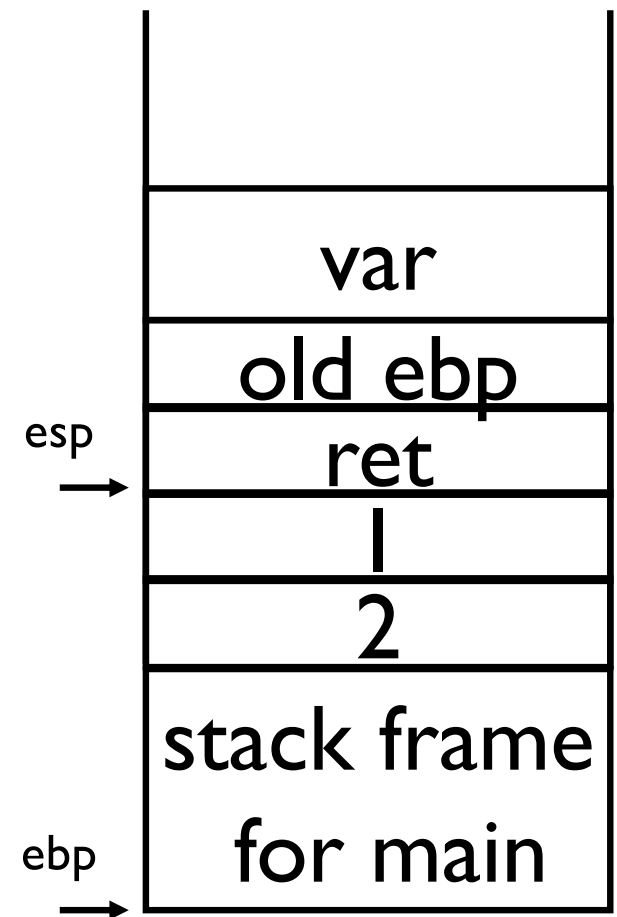
At call



At invocation



At return



- Stack stores **data** and **metadata** about runtime
 - Code addresses and stack frame addresses
- When function returns, the **code executes from the return address**
 - And stack frame is set to prior frame (value of “old ebp”)
- All **local variables and parameters** passed between functions are stored on the stack
 - Any type of variable can be present on the stack
- Stack **data persists** after return
 - Stack frames are overwritten rather than erased

- The data created via **dynamic memory allocation** resides in the **heap** segment
- Such data is created by memory allocation system calls
 - E.g., malloc, calloc
- And reclaimed by the programmer using deallocation system calls
 - E.g., free
 - NOTE: No garbage collection in C
- Again, the programmer is responsible for how memory is used by their programs

- The heap is a contiguous virtual memory region in which memory can be dynamically allocated
 - ▶ `tree_t *t = (tree_t *)malloc(sizeof(tree_t));`
 - What does the above accomplish?
- To track allocation state, heap metadata must also be stored
 - ▶ E.g., location of objects and their sizes as well as free slots in the heap and their sizes
 - ▶ Sometimes this metadata is also stored with the objects – like the stack

Heap Structure (Depends)

- Where the heap metadata is stored is allocator-specific
 - One option...

Meta Data	Object	Meta Data	Object	Meta Data
--------------	--------	--------------	--------	--------------

Heap Issues

- Heap may store **data** and **metadata** about allocation
 - Used and free slots and sizes
- Heap memory must be **reclaimed by the programmer**
 - If you forget to reclaim that creates a “**memory leak**”
- Heap memory may be **used in any function**
 - Thus, you must keep track of whether pointers to heap memory reference allocated or deallocated regions
- Heap objects may be referenced through globals, pointers in other heap objects, or stack pointers

Your Questions

- Anything you have been curious about in the C language that I have not discussed?

Take Away

- The C language gives programmers **a lot of latitude** to manage the use of memory in their programs
 - This enables you to write more efficient programs
- But, this latitude can enable you to **create unintended functionality**
 - That can lead to flaws – e.g., segmentation faults
 - And some of these bugs may be exploitable
- We will examine methods to reduce flaws and prevent exploitation of flaws