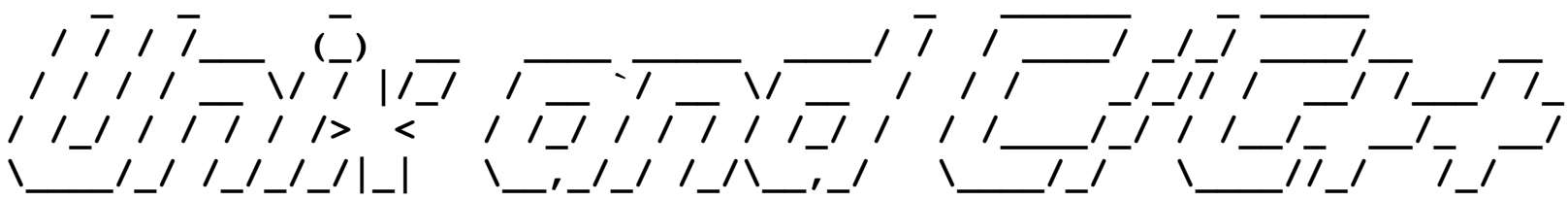
UNIVERSITY OF OREGON

CIS 607:



Lecture 2:
More on C, More on Memory

Oct 4th, 2021

Hank Childs, University of Oregon

Let’s Review Project 2A

- Next 14 slides are for reference if we need them

Motivation: Project 2A

Assignment: fill out this worksheet.

Location	0x8000	0x8004	0x8008	0x800c	0x8010	0x8014	0x8018
Value	0	1	1	2	3	5	8
Location	0x801c	0x8020	0x8024	0x8028	0x802c	0x8030	0x8034
Value	13	21	34	55	89	144	233
Location	0x8038	0x803c	0x8040	0x8044	0x8048	0x804c	0x8050
Value	377	610	987	1597	2584	4181	6765

Code:

```
int *A = 0x8000;
int *B[3] = { A, A+7, A+14 };
```

Note: "NOT ENOUGH INFO" is a valid answer.

Variable	Your Answer	Variable	Your Answer
A	0x8000	(A+6)-(A+3)	
&A	NOT ENOUGH INFO	*(A+6)-*(A+4)	
A[2]	1	A[5]-*(A+4)	
*A		(A+6)-B[0]	

Important Context

- Different types have different sizes:
  - int: 4 bytes
  - float: 4 bytes
  - double: 8 bytes
  - char: 1 byte
  - unsigned char: 1 byte

Important Memory Concepts in C (1/9): Stack versus Heap

- You can allocate variables that only live for the invocation of your function
  - Called stack variables (will talk more about this later)
- You can allocated variables that live for the whole program (or until you delete them)
  - Called heap variables (will talk more about this later as well)

Important Memory Concepts in C (2/9): Pointers

- Pointer: points to memory location
  - Denoted with '*'
  - Example: "int *p"
    - pointer to an integer
  - You need pointers to get to heap memory
- Address of: gets the address of memory
  - Operator: '&'
  - Example:

```
int x;
int *y = &x;
```

Important Memory Concepts in C (3/9): Memory allocation

- Special built-in function to allocate memory from heap: malloc
  - Interacts with operating system
  - Argument for malloc is how many bytes you want
- Also built-in function to deallocate memory: free

malloc/free example

Enables compiler to see functions that aren't in this file. More on this next week.

```
#include <stdlib.h>
int main()
{
    /* allocates memory */
    int *ptr = malloc(2*sizeof(int));

    /* deallocates memory */
    free(ptr);
}
```

sizeof is a built in function in C. It returns the number of bytes for a type (4 bytes for int).

don't have to say how many bytes to free … the OS knows

Important Memory Concepts in C (4/9): Arrays

- Arrays lie in contiguous memory
  - So if you know address to one element, you know address of the rest
- int *a = malloc(sizeof(int)*1);
  - a single integer
  - … or an array of a single integer
- int *a = malloc(sizeof(int)*2);
  - an array of two integers
  - first integer is at ‘a’
  - second integer is at the address ‘a+4’
    - Tricky point here, since C/C++ will refer to it as ‘a+1’

Important Memory Concepts in C (5/9): Dereferencing

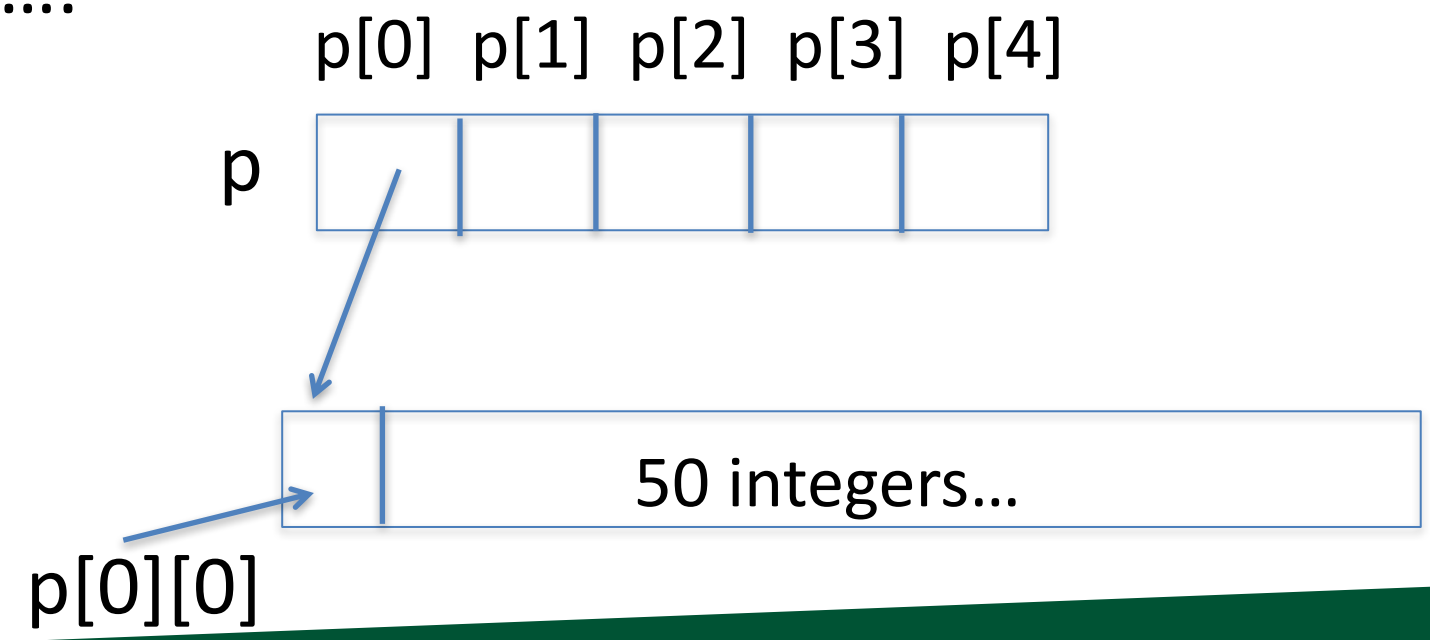
- There are two operators for getting the value at a memory location: *, and []
  - This is called deferencing
    - * = “dereference operator”
- int *p = malloc(sizeof(int)*1);
- *p = 2; /* sets memory p points to to have value 2 */
- p[0] = 2; /* sets memory p points to to have value 2 */

Important Memory Concepts in C (6/9): pointer arithmetic

- int *p = malloc(sizeof(int)*5);
- C/C++ allows you to modify pointer with math operations
  - called pointer arithmetic
  - "does the right thing" with respect to type
    - int *p = malloc(sizeof(int)*5);
    - p+1 is 4 bytes bigger than p!!
- Then:
  - "p+3" is the same as "&(p[3])" (ADDRESSES)
  - "*(p+3)" is the same as "p[3]" (VALUES)

Important Memory Concepts in C (7/9)
Pointers to pointers

- int **p = malloc(sizeof(int *)*5);
- p[0] = malloc(sizeof(int)*50);
- ….



Important Memory Concepts in C (8/9): Hexadecimal address

- Addresses are in hexadecimal
- int *A = 0x8000;
- Then A+1 is 0x8004. (Since int is 4 bytes)

Important Memory Concepts in C (9/9) NULL pointer

- int *p = NULL;
- often stored as address 0x0000000
- used to initialize something to a known value
  - And also indicate that it is uninitialized…

New Material

Actions taken by a computer

- A computer operates by loading a special kind of file called a “program”
  - This file has a sequence of instructions in it
- Instructions are very primitive:
  - Add, subtract, <lots of math stuff>
  - Load from memory, store to memory
  - “Jump”
  - Some other things…

Instructions

- Every cycle get an instruction -- “bytes that explain what to do”
- If byte == 000, then add two numbers
- If byte == 001, then subtract two numbers
- If byte == 002, then multiply two numbers
- …

- (and of course the real conventions are very different)

Computer programs can do iteration

- Line 100: X=0
- Line 101: A=location in memory
- Line 102: if X==10, jump to line 106
- Line 103: A = 2*A (for C programmers: *A=2**A)
- Line 104: X = X+1
- Line 105: jump to line 102
- Line 106: …. (other stuff)

With iteration, we can write finite length programs that can run forever…

Billions and Billions of Instructions Can Add Up to Do Great Things

Von Neumann Architecture

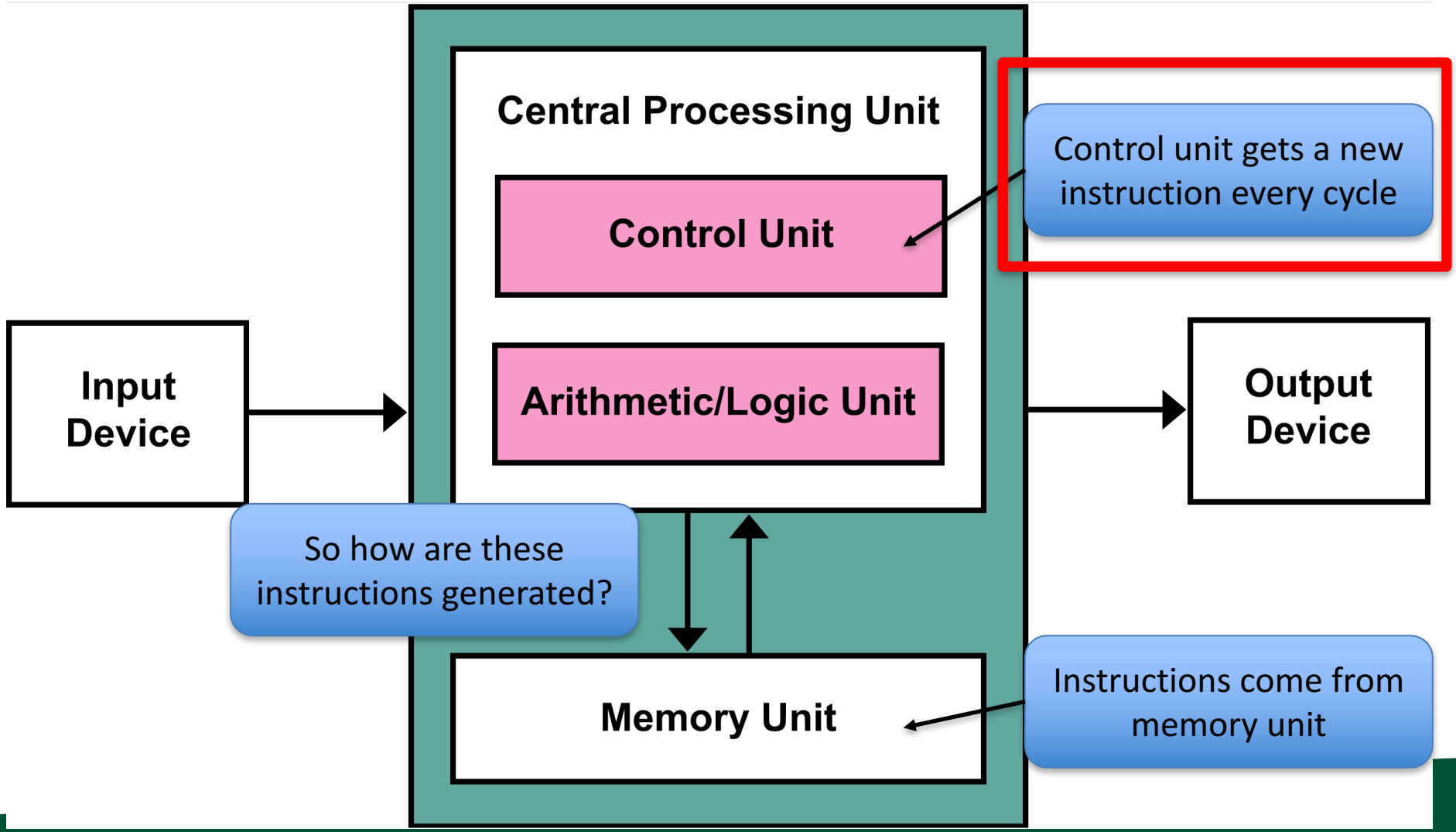


Image source: wikipedia

How computers do work?

- Instructions come from programs, which are loaded into memory.
- Programs contain a sequence of instructions that CPU can understand (“opcodes”) and those instructions are fetched one by one.
  - The correct term is technically machine code and not opcodes. Opcodes refer to the instruction to perform (plus, minus, times, etc), while machine code augment opcodes with other info.

So where do programs come from?

- (Answer coming in a few slides)

Deep thought: Python is a program

- Python gets input from the keyboard
- It “interprets” the input
- It translates the input to something it understands
- It converts that input to instructions for the CPU
- It sends the instructions to the CPU

Python example

- >>> 4+6
- Python recognizes this as two numbers (4 and 6) and a known instruction (+)
- It sends the numbers and the opcode for + to the CPU as an instruction
- The CPU does the math
- Python takes the result (10) and prints it

Python example

- How did Python know that “4+6” was two numbers and a known instruction?
- Answer: the + instruction and “4” and “6” arguments were the LAST things it did.
- Before that, it used other instructions to understand the string and determine what it was supposed to do.
- Typing “4+6” into the terminal actually requires thousands of instructions (or more) to carry out the 1 instruction you wanted!

Python is an interpreter

- It accepts strings that you type
- It determines your intended actions
- It converts those intended actions into instructions for CPU
- It sends the instructions to the CPU
- It displays the results
- … and it does this until you quit.

Deep thought: where did Python come from?

- Python is a program.
- Actually a very complex program.
- Something created that program.
- The thing that creates that program is called a compiler.

The Workflow With Compilers

- 1) start with source code
- 2) invoke compiler, which takes source code and generates executable
  - The executable is made up of instructions for the CPU (opcode)
- 3) invoke executable
  - The instructions for the CPU are fed to the CPU one at a time

Hello World Example

```
Hanks-iMac:Downloads hank$ cat hello.c
#include <stdio.h>
int main()
{
   printf("Hello world\n");
}
Hanks-iMac:Downloads hank$ gcc hello.c
.Hanks-iMac:Downloads hank$ ./a.out
Hello world
```

Notes on previous example

- Ignore for now: #include <stdio.h>
- “main” is the first function called in C.
  - (Sort of. There are other things called before main to set up the program. But you won’t be mucking with those.)
- gcc is a C compiler. It is short for GNU Compiler Collection. GNU is an open source effort.
- The output is called “a.out”
- We have to add “./”, as it “./a.out”
  - “.” is the directory the shell is in. Therefore “./a.out” says run the program called a.out in the current directory.

More Deep Thoughts

- The compiler is a program. Its job is to make other programs.
- The operating system is a program. Its job is to run other programs (and provide an environment for them).
- Where did the first compiler come from?

If your job is to…	How do you feel about it? (*)	Is it fast?	How does it become a program?
By hand, write binary files full of opcodes		Yes	It already is a program
Write "assembly code" (the instructions in English, not opcodes)	 Not Great	Yes	Assembler
Write C programs	prettygood	Yes, although assembly can be faster	Compiler
Write Python programs		Almost certainly not, unless it is calling subroutines in C	Python is already a program

* = I feel AMAZING about C, and only pretty good about Python

As

```
Hanks-iMac:Downloads hank$
#include <stdio.h>
int main()
{
   int X = 3*3+3;
   printf("X is %d\n", X);
}
Hanks-iMac:Downloads hank$
```

```
Hanks-iMac:Downloads hank$ cat simple_math.s
        .section        __TEXT,__text,regular,pure_instructions
        .macosx_version_min 10, 12
        .globl  _main
        .p2align        4, 0x90
_main:                                  ## @main
        .cfi_startproc
## BB#0:
        pushq   %rbp
Ltmp0:
        .cfi_def_cfa_offset 16
Ltmp1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp2:
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        leaq    L_.str(%rip), %rdi
        movl    $12, -4(%rbp)
        movl    -4(%rbp), %esi
        movb    $0, %al
        callq   _printf
        xorl    %esi, %esi
        movl    %eax, -8(%rbp)          ## 4-byte Spill
        movl    %esi, %eax
        addq    $16, %rsp
        popq    %rbp
        retq
        .cfi_endproc

        .section        __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
        .asciz  "X is %d\n"


.subsections_via_symbols
```

Notes

- Assembly code varies from architecture to architectures
- C does not
  - You can code in C, and the compiler will make your code work anywhere
- Just about every command in C corresponds to a small handful of assembly instructions
  - This means C will be fast
- This is not true in Python

If your job is to…	How do you feel about it? (*)	Is it fast?	How does it become a program?
By hand, write binary files full of opcodes		Yes	It already is a program
Write "assembly code" (the instructions in English, not opcodes)	 Not Great	Yes	Assembler
Write C programs		Yes, although assembly can be faster	Compiler
Write Python programs		Almost certainly not, unless it is calling subroutines in C	Python is already a program

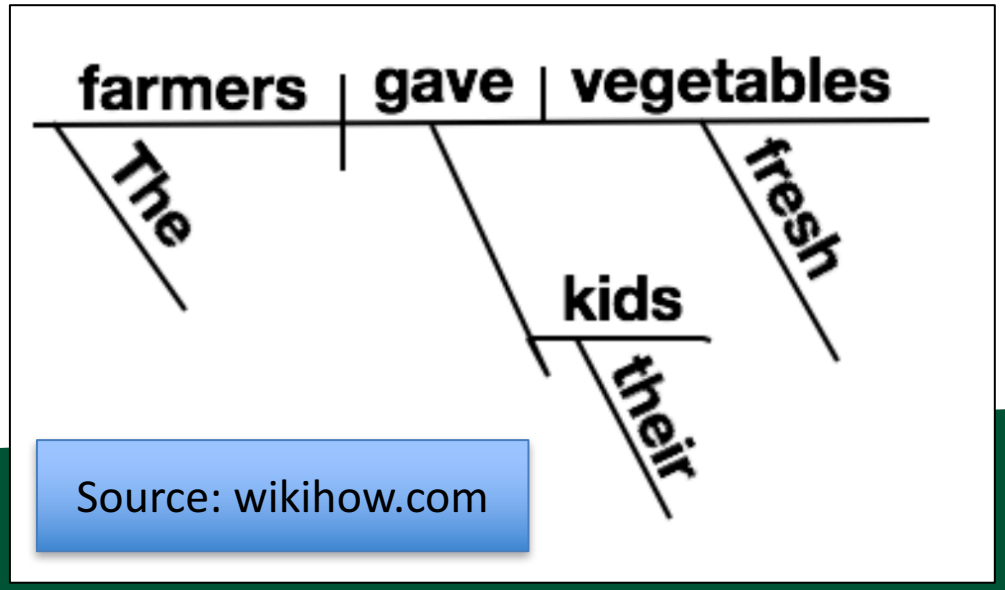
One reason I like C is the compiler. It looks at all of my code, and tells me right away if there is a syntax problem.

* = I feel AMAZING about C, and only pretty good about Python

Aside: diagramming sentences

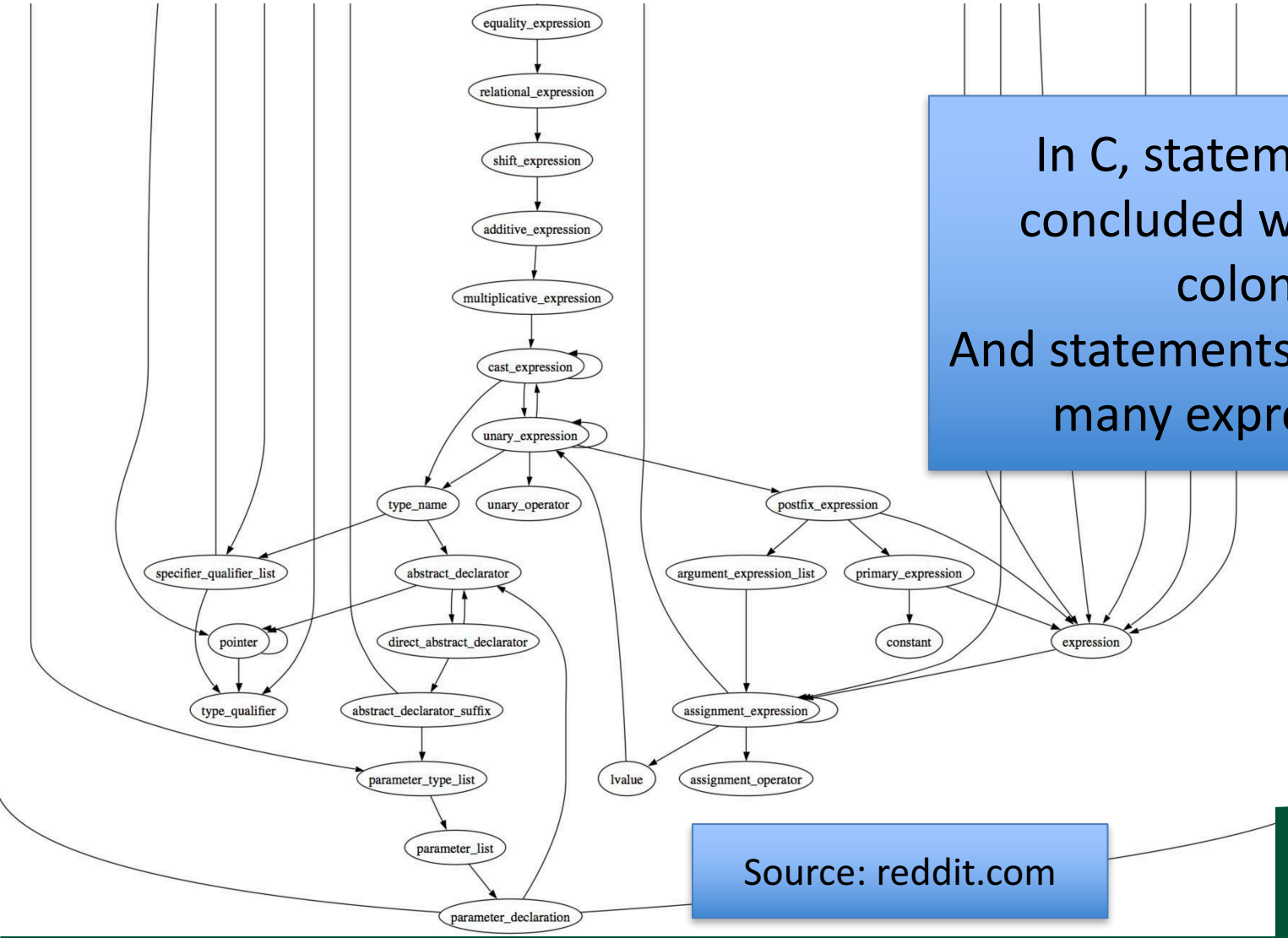
- English language is made up of sentences.
- There are different formulas for a sentence.
- Sentences always end with punctuation.
  - This punctuation reduces ambiguity and helps the reader.

We don't try learn languages (spoken or programming) by looking at the grammar that defines them. We do it by example….



Source: wikihow.com

C grammar



In C, statements are concluded with semi-colons.
And statements can contain many expressions.

Source: reddit.com

More C: variable types

- When you declare variables, you have to declare their type.
  - int, float, double, char, unsigned char
- These types affect how bits/bytes are interpreted and what part of the ALU is used to process them

More C: functions

- As you would expect, you can define your own functions and call them.

```
Hanks-iMac:Downloads hank$ cat function.c
#include <stdio.h>
int foo1() { return 3; };
int foo2() { return 4; };
int main()
{
    printf("%d\n", foo1()*foo2());
}
Hanks-iMac:Downloads hank$ gcc function.c
Hanks-iMac:Downloads hank$ ./a.out
12
```

More C: every function has its own "scope" and its variables live within that scope

```
Hanks-iMac:Downloads hank$ cat scope.c
#include <stdio.h>

int foo1()
{
    int X = 3;
    return X;
}

int main()
{
    int Y = foo1();
    printf("X is %d\n", X);
}
Hanks-iMac:Downloads hank$ gcc scope.c
scope.c:12:25: error: use of undeclared identifier 'X'
    printf("X is %d\n", X);
                        ^
1 error generated.
```

You can also use more { and } within a function. This affects variable scope.

```
Hanks-iMac:Downloads hank$ cat scope2.c
#include <stdio.h>

int main()
{
    int Y = 3;
    {
        int X = 2;
    }
    printf("X is %d\n", X);
}
Hanks-iMac:Downloads hank$ gcc scope2.c
scope2.c:9:25: error: use of undeclared identifier 'X'
    printf("X is %d\n", X);
                        ^
1 error generated.
```

Iteration is done with constructs “for” and “while”

```
Hanks-iMac:Downloads hank$ cat while.c
#include <stdio.h>

int main()
{
    int X = 0;
    while (X < 83)
       X += 5;
    printf("X is %d\n", X);
}
Hanks-iMac:Downloads hank$ gcc while.c
Hanks-iMac:Downloads hank$ ./a.out
X is 85
```

What's the answer?

```
Hanks-iMac:Downloads hank$ cat while.c
#include <stdio.h>

int main()
{
    int X = 0;
    while (X < 8)
       X += 5;
       X += 2;
    printf("X is %d\n", X);
}
Hanks-iMac:Downloads hank$ gcc while.c
./Hanks-iMac:Downloads hank$ ./a.out
```

```
Hanks-iMac:Downloads hank$ cat while.c
#include <stdio.h>

int main()
{
    int X = 0;
    while (X < 8)
    {
    X += 5;
    X += 2;
    }
    printf("X is %d\n", X);
}
Hanks-iMac:Downloads hank$ gcc while.c
Hanks-iMac:Downloads hank$ ./a.out
X is 14
```

X is 12

The for loop

- Three main components:
  - Initialization
  - Termination
  - What to do each step
- Often used with a loop variable (example: i)
  - Initialization: i = 0
  - Termination: i < 10
  - What to do each step: i = i+1

For loop in practice

```
Hanks-iMac:Downloads hank$ cat sum.c
#include <stdio.h>
int main()
{
    int sum = 0;
    int i;
    for (i = 0 ; i < 10 ; i = i+1)
    {
        sum = sum+i;
    }
    printf("Sum is %d\n", sum);
}
Hanks-iMac:Downloads hank$ gcc sum.c
Hanks-iMac:Downloads hank$ ./a.out
Sum is 45
```

Handy: increment operator / decrement operator

```
Hanks-iMac:Downloads hank$ cat increment.c
#include <stdio.h>
int main()
{
    int X = 0;
    X++;  /* Now X is 1 */
    X++;  /* Now X is 2 */
    X--;  /* Now X is 1 again */
    printf("X is %d\n", X);
}
Hanks-iMac:Downloads hank$ gcc increment.c
Hanks-iMac:Downloads hank$ ./a.out
X is 1
```

For loop in practice (with increment operator)

```
Hanks-iMac:Downloads hank$ cat sum.c
#include <stdio.h>
int main()
{
    int sum = 0;
    int i;
    for (i = 0 ; i < 10 ; i++)
    {
        sum = sum+i;
    }
    printf("Sum is %d\n", sum);
}
Hanks-iMac:Downloads hank$ gcc sum.c
Hanks-iMac:Downloads hank$ ./a.out
Sum is 45
```

More on Memory

Memory Segments

- Von Neumann architecture: one memory space, for both instructions and data
- → so break memory into “segments”
  - … creates boundaries to prevent confusion
- 4 segments:
  - Code segment
  - Data segment
  - Stack segment
  - Heap segment

Code Segment

- Contains assembly code instructions
- Also called text segment
- This segment is modify-able, but that's a bad idea
  - "Self-modifying code"
    - Typically ends in a bad state very quickly.

Data Segment

- Contains data not associated with heap or stack
  - global variables
  - statics (to be discussed later)
  - character strings you've compiled in

```
char *str = "hello world\n"
```

Stack: data structure for collection

- A stack contains things
- It has only two methods: push and pop
  - Push puts something onto the stack
  - Pop returns the most recently pushed item (and removes that item from the stack)
- LIFO: last in, first out

Imagine a stack of trays.
You can place on top (push).
Or take one off the top (pop).

Stack

- Stack: memory set aside as scratch space for program execution
- When a function has local variables, it uses this memory.
  - When you exit the function, the memory is lost

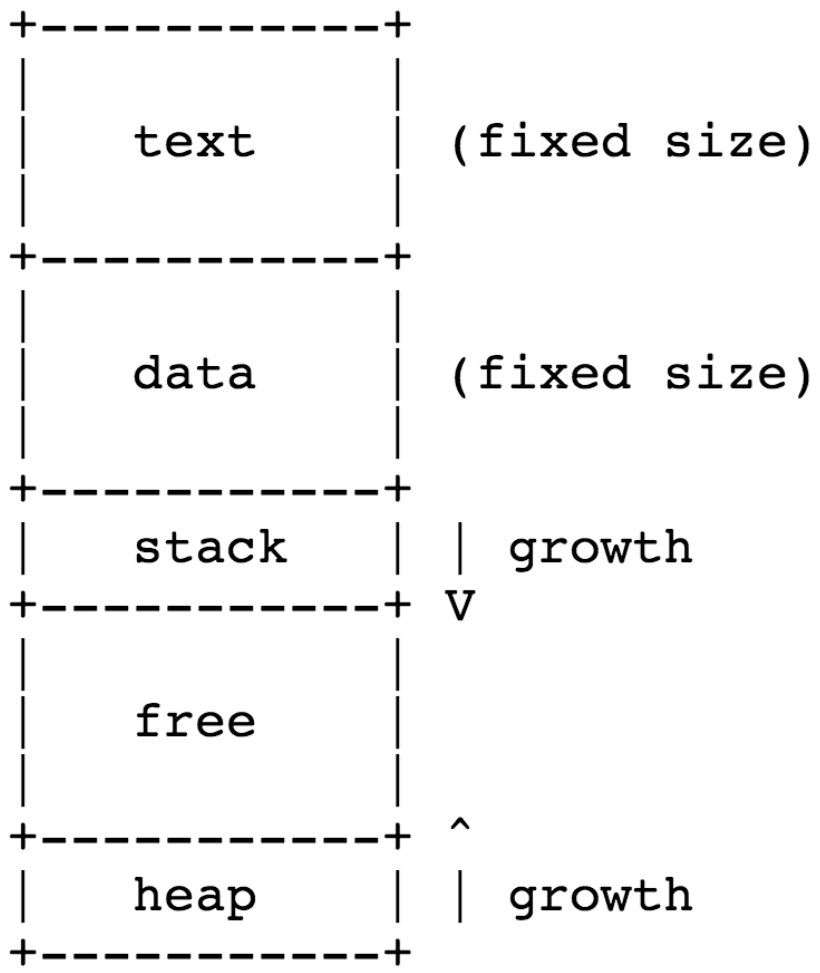
Stack

- The stack grows as you enter functions, and shrinks as you exit functions.
  - This can be done on a per variable basis, but the compiler typically does a grouping.
    - Some exceptions (discussed later)
- Don’t have to manage memory: allocated and freed automatically

Heap

- Heap (data structure): tree-based data structure
- Heap (memory): area of computer memory that requires explicit management (malloc, free).
- Memory from the heap is accessible any time, by any function.
  - Contrasts with the stack

Memory Segments



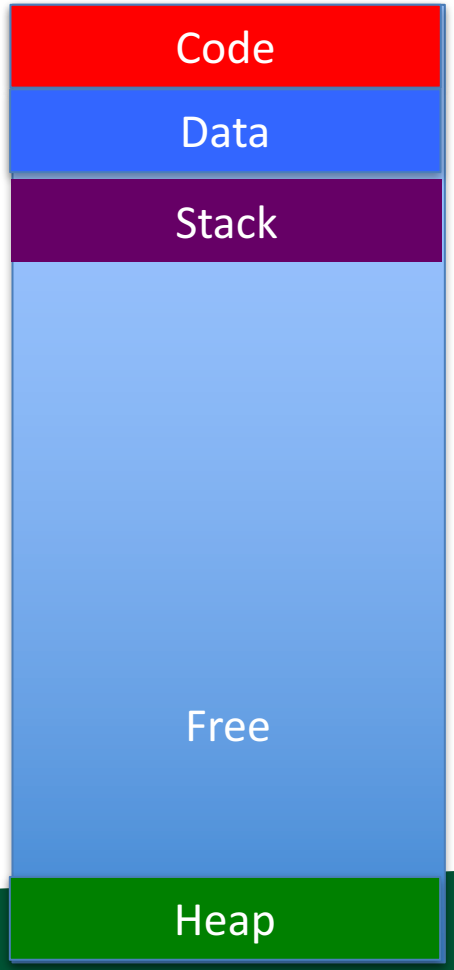
Source: http://www.cs.uwm.edu/classes/cs315/Bacon/

Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/Deal location	Automatic	Explicit

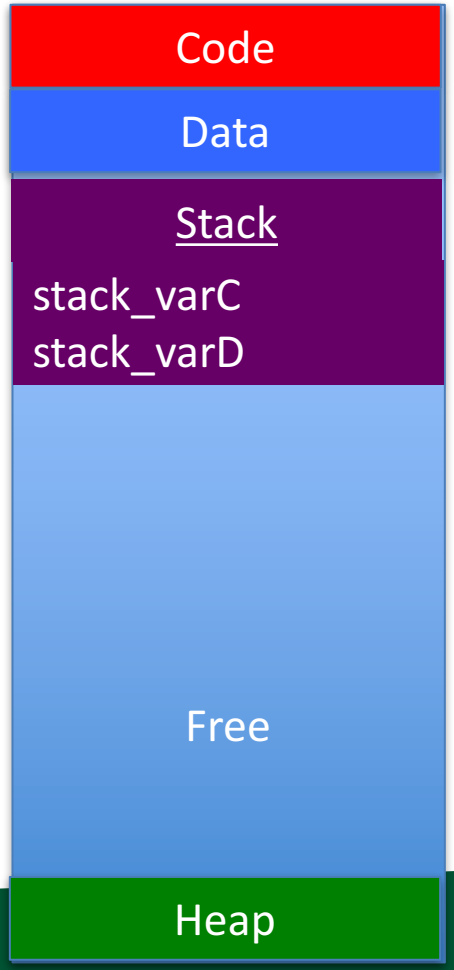
How stack memory is allocated into Stack Memory Segment

```
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```



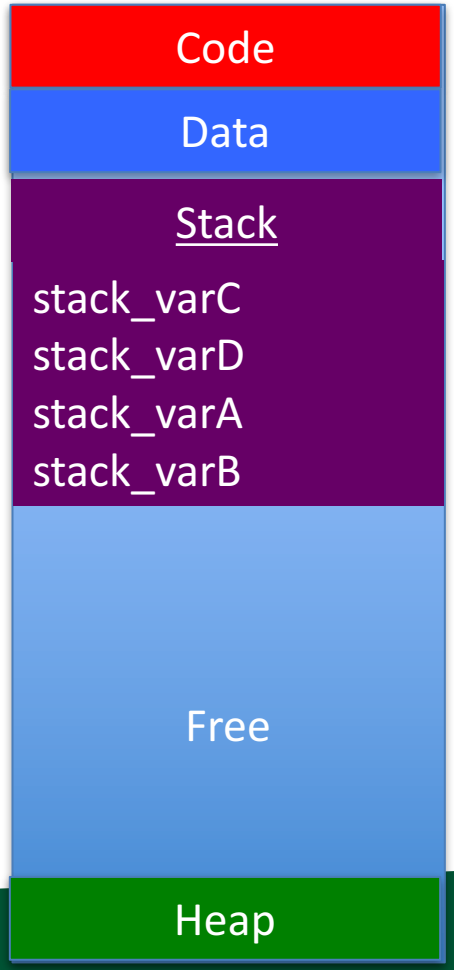
How stack memory is allocated into Stack Memory Segment

```
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```



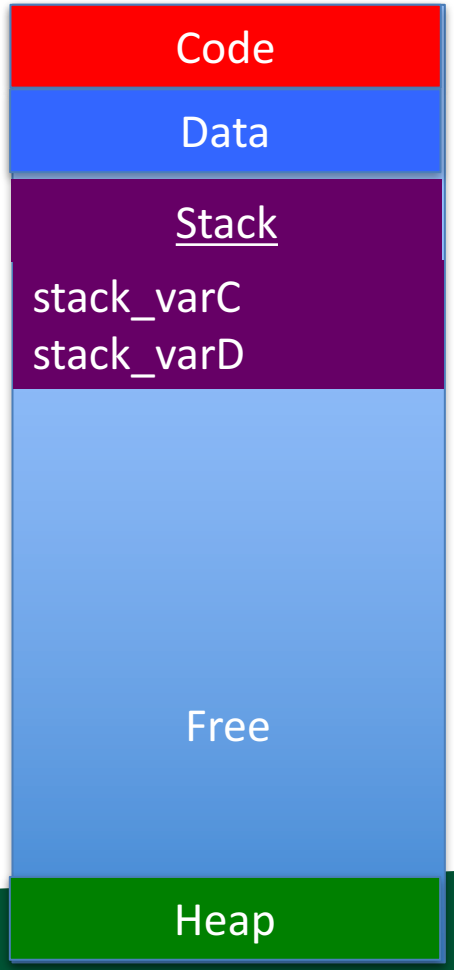
How stack memory is allocated into Stack Memory Segment

```
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```



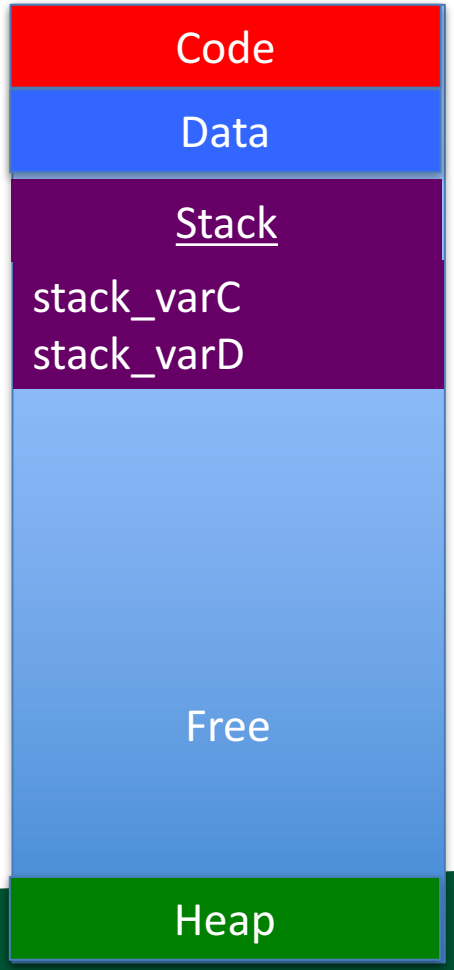
How stack memory is allocated into Stack Memory Segment

```
void foo()
{
    int stack_varA;
    int stack_varB;
}

int main()
{
    int stack_varC;
    int stack_varD;
    foo();
}
```



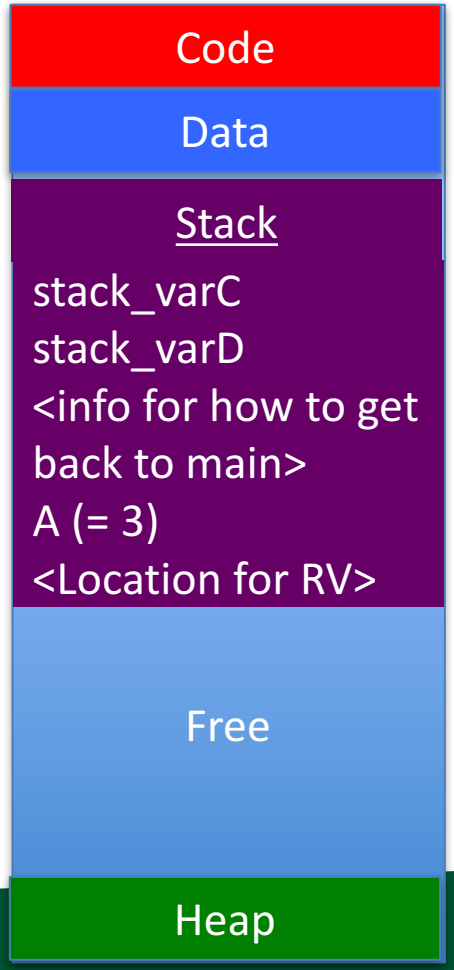
How stack memory is allocated into Stack Memory Segment

```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```



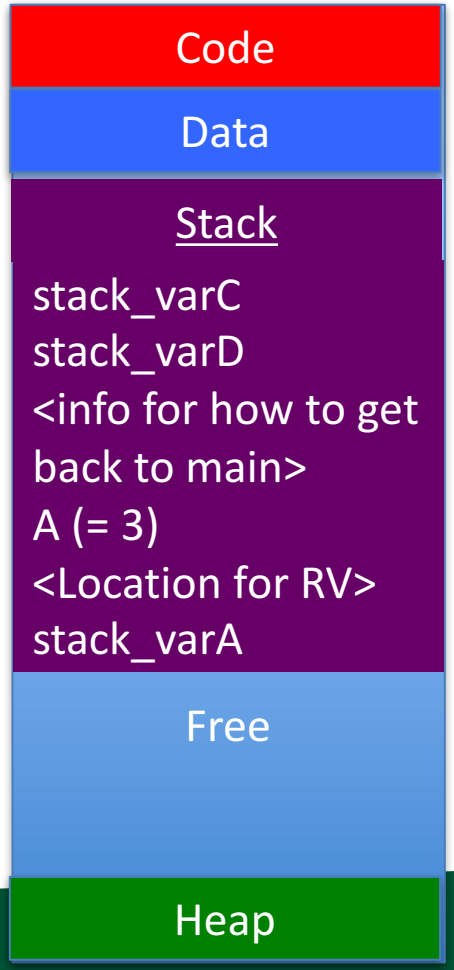
How stack memory is allocated into Stack Memory Segment

```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```



How stack memory is allocated into Stack Memory Segment

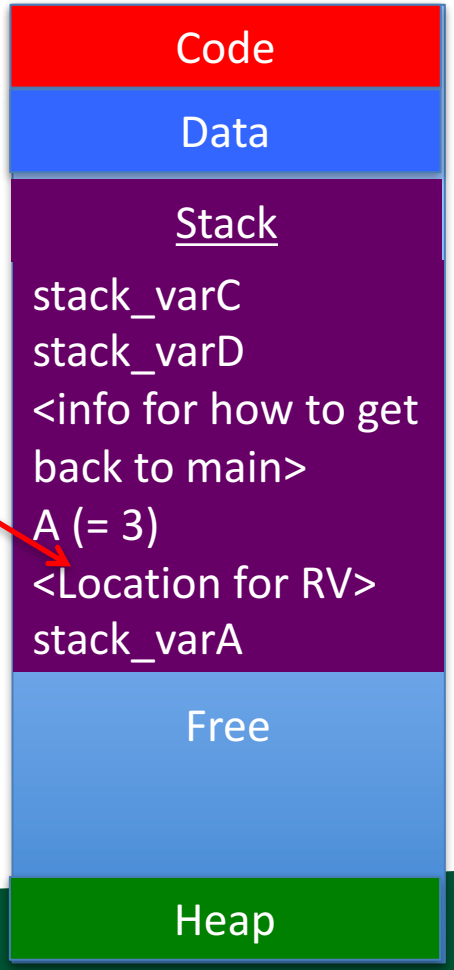
```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```



How stack memory is allocated into Stack Memory Segment

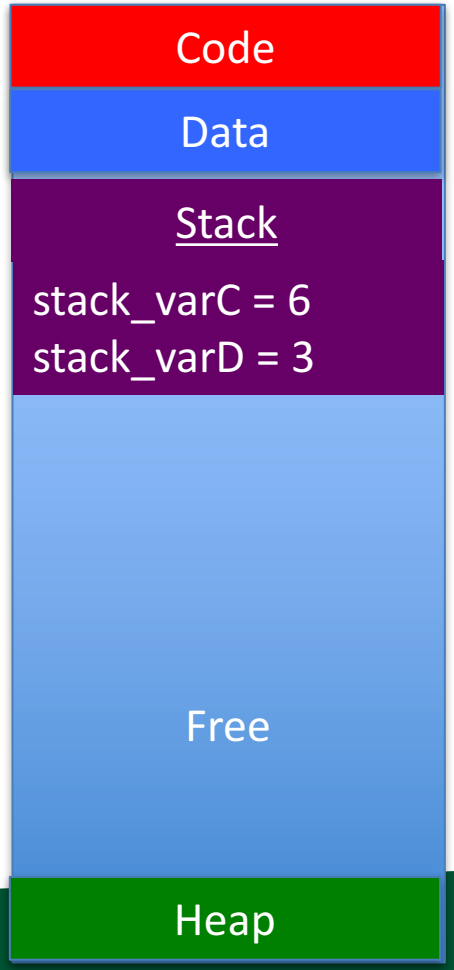
```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```

Return copies into location specified by calling function



How stack memory is allocated into Stack Memory Segment

```
int doubler(int A)
{
    int stack_varA;
    stack_varA = 2*A;
    return stack_varA;
}

int main()
{
    int stack_varC;
    int stack_varD = 3;
    stack_varC = doubler(stack_varD);
}
```



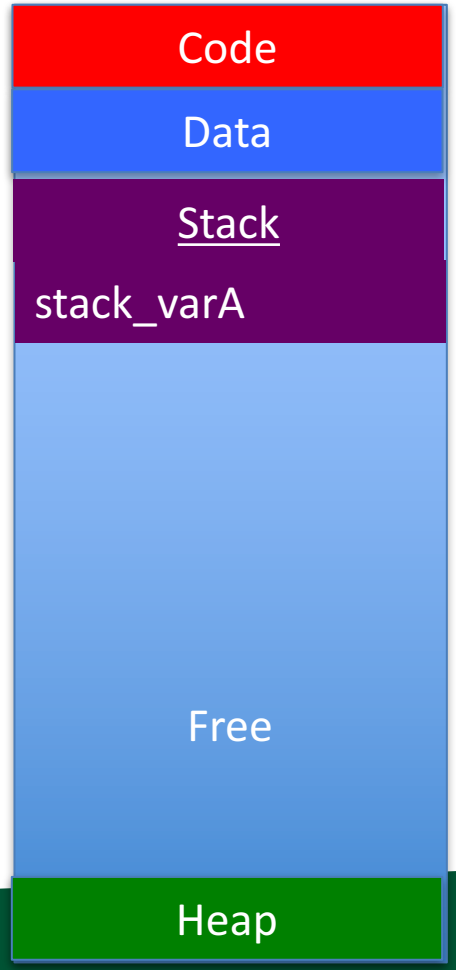
This code is very problematic … why?

```
int *foo()
{
    int stack_varC[2] = { 0, 1 };
    return stack_varC;
}
int *bar()
{
    int stack_varD[2] = { 2, 3 };
    return stack_varD;
}

int main()
{
    int *stack_varA, *stack_varB;
    stack_varA = foo();
    stack_varB = bar();
    stack_varA[0] *= stack_varB[0];
}
```

foo and bar are returning addresses that are on the stack … they could easily be overwritten
(and bar's stack_varD overwrites foo's stack_varC in this program)

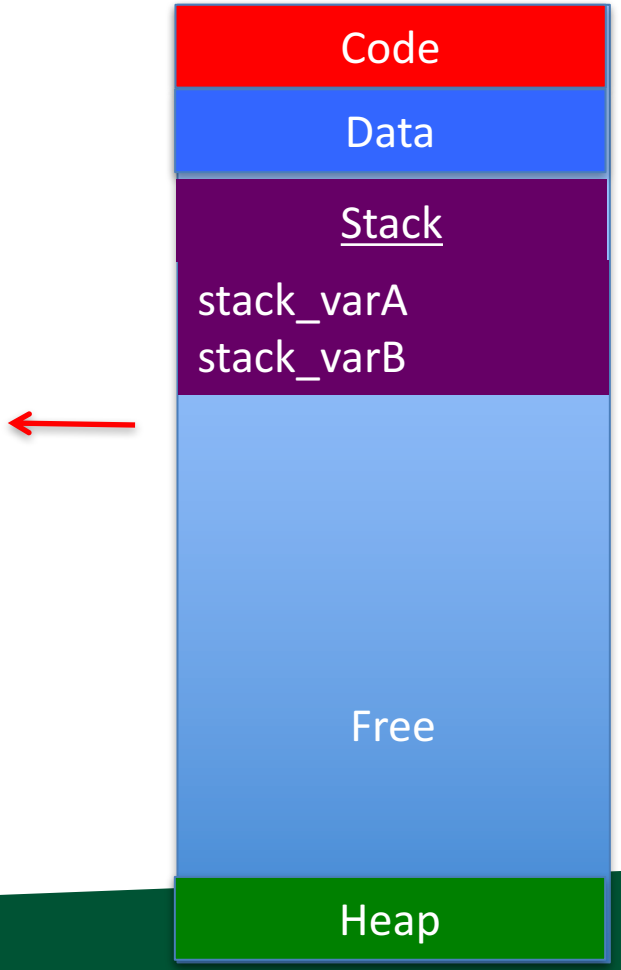
Nested Scope

```
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```



Nested Scope

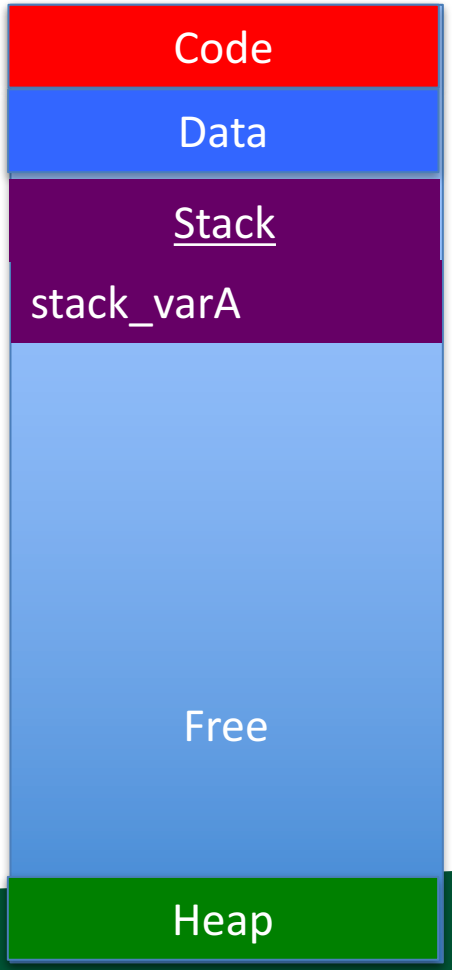
```
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```



Nested Scope

```
int main()
{
    int stack_varA;
    {
        int stack_varB = 3;
    }
}
```

You can create new scope within a function by adding '{' and '}'.



Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/Deal location	Automatic	Explicit
Access	Fast	Slower

Memory pages associated with stack are almost always immediately available.

Memory pages associated with heap may be located anywhere ... may be caching effects

Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/Deal location	Automatic	Explicit
Access	Fast	Slower
Variable scope	Limited	Unlimited

Variable scope: stack

```
int *foo()
{
    int stack_varA[2] = { 0, 1 };
    return stack_varA;
}

int *bar()
{
    int *heap_varB;
    heap_varB = malloc(sizeof(int)*2);
    heap_varB[0] = 2;
    heap_varB[1] = 2;
    return heap_varB;
}

int main()
{
    int *stack_varA;
    int *stack_varB;
    stack_varA = foo(); /* problem */
    stack_varB = bar(); /* still good */
}
```

foo is bad code … never return memory on the stack from a function

bar returned memory from heap

The calling function – i.e., the function that calls bar – must understand this and take responsibility for calling free.

If it doesn't, then this is a "memory leak".

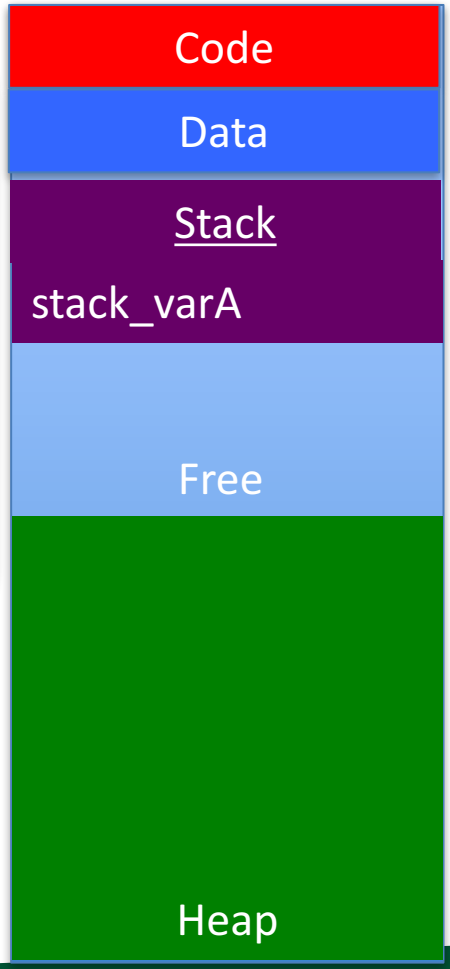
Memory leaks

It is OK that we are using the heap … that's what it is there for

The problem is that we lost the references to the first 49 allocations on heap

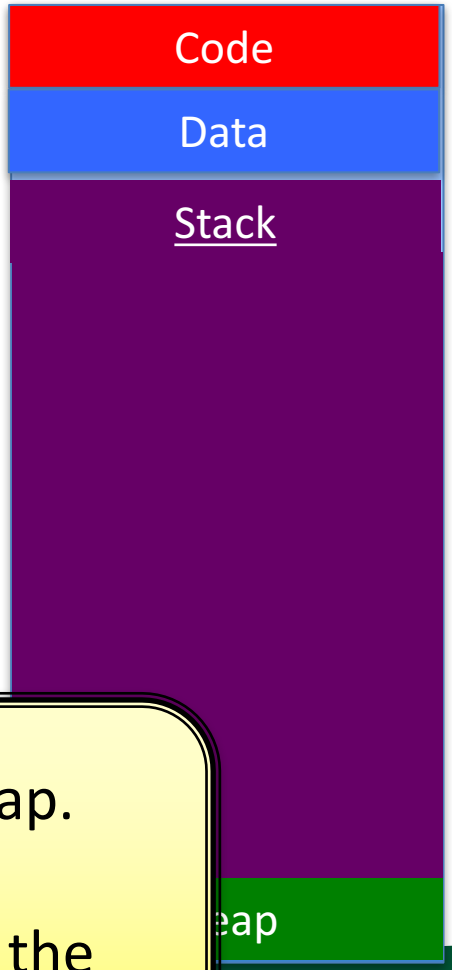
The heap's memory manager will not be able to re-claim them … we have effectively limited the memory available to the program.

```
{
    int i;
    int stack_varA;
    for (i = 0 ; i < 50 ; i++)
       stack_varA = bar();
}
```



Running out of memory (stack)

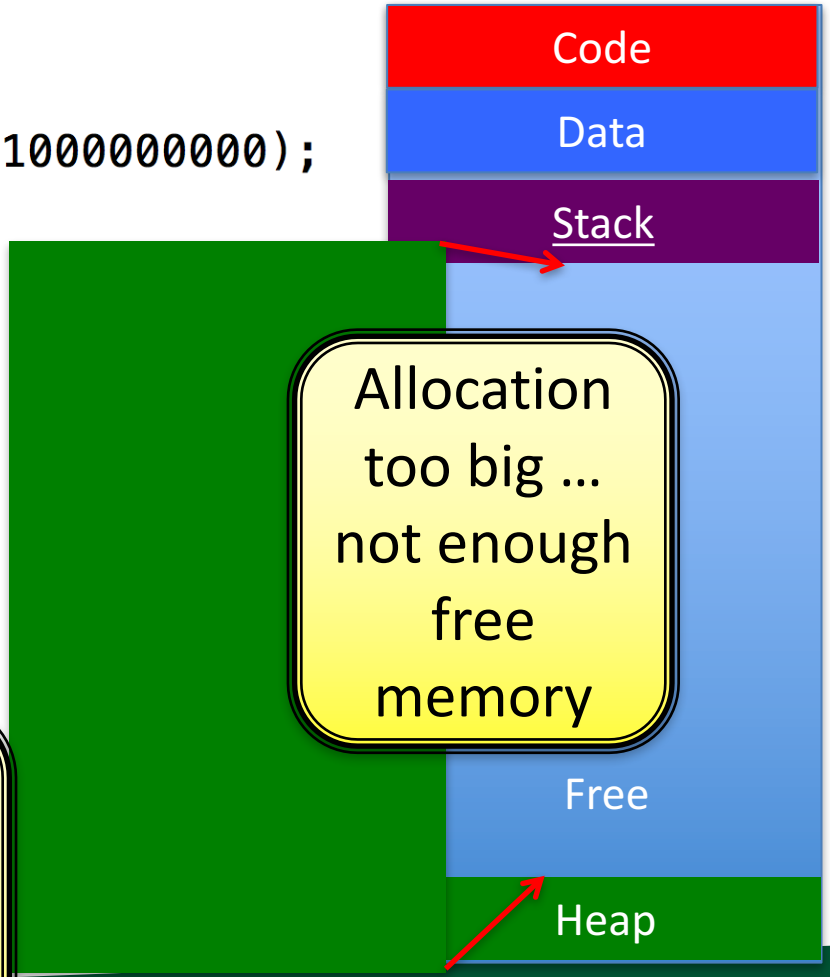
```
int endless_fun()
{
    endless_fun();
}

int main()
{
    endless_fun();
}
```



stack overflow: when the stack runs into the heap.
There is no protection for stack overflows.
(Checking for it would require coordination with the heap's memory manager on every function calls.)

Running out of memory (heap)

```
int *heaps_o_fun()
{
    int *heap_A = malloc(sizeof(int)*1000000000);
    return heap_A;
}

int main()
{
    int *stack_A;
    stack_A = heaps_o_fun();
}
```



If the heap memory manager doesn't have room to make an allocation, then malloc returns NULL …. a more graceful error scenario.

Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/Deal location	Automatic	Explicit
Access	Fast	Slower
Variable scope	Limited	Unlimited
Fragmentation	No	Yes

Memory Fragmentation

- Memory fragmentation: the memory allocated on the heap is spread out of the memory space, rather than being concentrated in a certain address space.

Memory Fragmentation

```
int *bar()
{
   int *heap_varA;
   heap_varA = malloc(sizeof(int)*2);
   heap_varA[0] = 2;
   heap_varA[1] = 2;
   return heap_varA;
}

int main()
{
   int i;
   int stack_varA[50];
   for (i = 0 ; i < 50 ; i++)
      stack_varA[i] = bar();
   for (i = 0 ; i < 25 ; i++)
      free(stack_varA[i*2]);
}
```

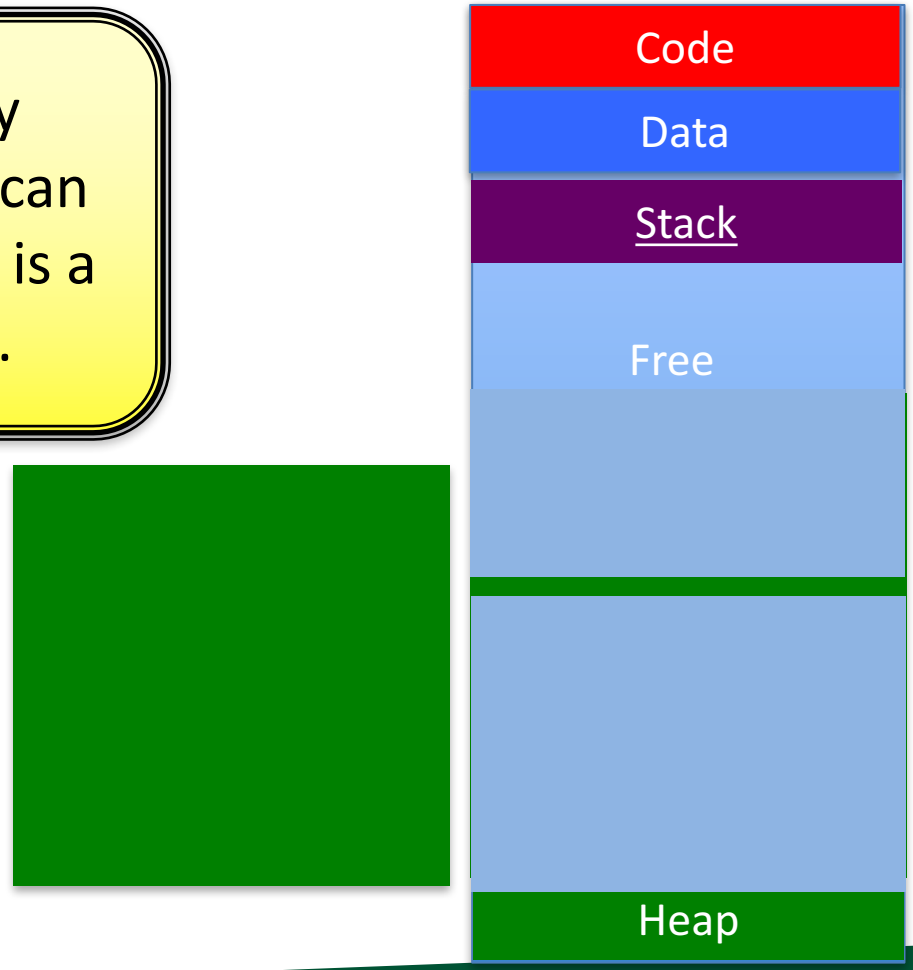
Negative aspects of fragmentation?

(1) can't make big allocations

(2) losing cache coherency

Fragmentation and Big Allocations

Even if there is lots of memory available, the memory manager can only accept your request if there is a big enough contiguous chunk.



Stack vs Heap: Pros and Cons

	Stack	Heap
Allocation/Deal location	Automatic	Explicit
Access	Fast	Slower
Variable scope	Limited	Unlimited
Fragmentation	No	Yes

Memory Errors

- Array bounds read

```
int main()
{
    int var;
    int arr[3] = { 0, 1, 2 };
    var=arr[3];
}
```

- Array bounds write

```
int main()
{
    int var = 2;
    int arr[3];
    arr[3]=var;
}
```

Memory Errors

- Free memory read / free memory write

```
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    var[0] = var[1];
}
```

When does this happen in real-world scenarios?

Memory Errors

- Freeing unallocated memory

```
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    free(var);
}
```

When does this happen in real-world scenarios?

Vocabulary: “dangling pointer”: pointer that points to memory that has already been freed.

Memory Errors

- Freeing non-heap memory

```
int main()
{
    int var[2]
    var[0] = 0;
    var[1] = 2;
    free(var);
}
```

When does this happen in real-world scenarios?

Memory Errors

- NULL pointer read / write

```
int main()
{
    char *str = NULL;
    printf(str);
    str[0] = 'H';
}
```

- NULL is never a valid location to read from or write to, and accessing them results in a "segmentation fault"
  - …. remember those memory segments?

When does this happen in real-world scenarios?

Memory Errors

- Unitialized memory read

```
int main()
{
    int *arr = malloc(sizeof(int)*10);
    int V2=arr[3];
}
```

When does this happen in real-world scenarios?