Globals, Statics, and Pointers

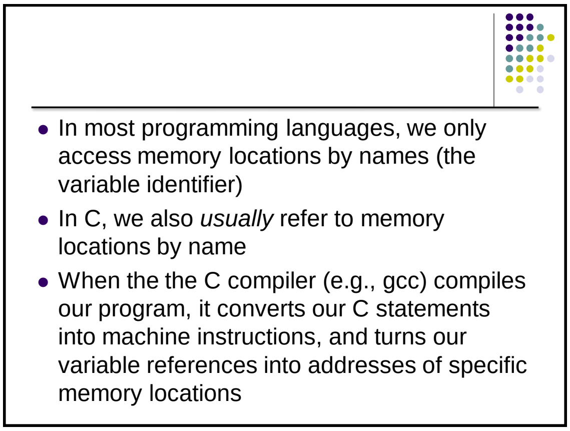
CMSC 104, Fall 2012
John Y. Park

Globals, Statics, and Pointers

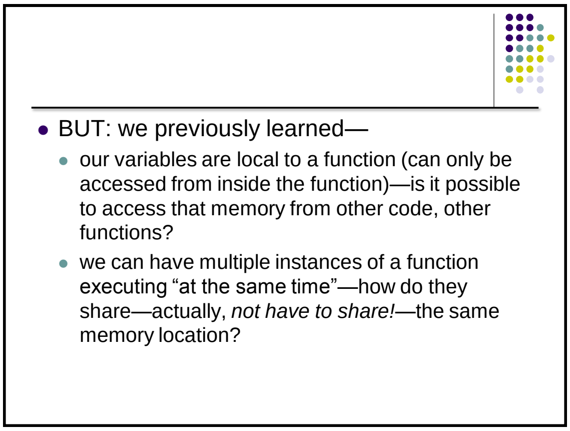
- Different kinds of variable scopes
- What global variables are
- What the static declaration does
- Pointers!
- Strings

Scope of a Variable

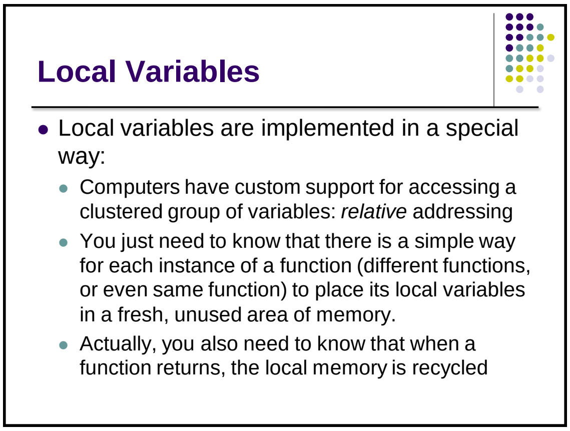
- A variable is simply a location in memory
- What matters is:
  - What type you interpret the contents as (integer? floating point? address?)
  - Who is using it?
  - Who will (re)use that location in the future?



- In most programming languages, we only access memory locations by names (the variable identifier)
- In C, we also *usually* refer to memory locations by name
- When the the C compiler (e.g., gcc) compiles our program, it converts our C statements into machine instructions, and turns our variable references into addresses of specific memory locations



- BUT: we previously learned—
  - our variables are local to a function (can only be accessed from inside the function)—is it possible to access that memory from other code, other functions?
  - we can have multiple instances of a function executing “at the same time”—how do they share—actually, *not have to share!*—the same memory location?



Local Variables

- Local variables are implemented in a special way:
  - Computers have custom support for accessing a clustered group of variables: *relative* addressing
  - You just need to know that there is a simple way for each instance of a function (different functions, or even same function) to place its local variables in a fresh, unused area of memory.
  - Actually, you also need to know that when a function returns, the local memory is recycled

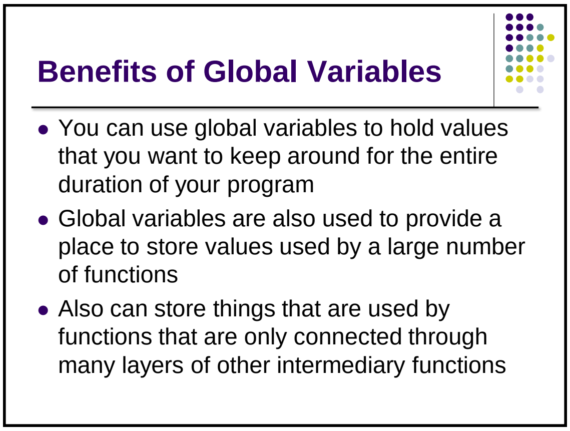
- Are there any other kinds of variables?
- It would be useful to have variables that:
  - can be referenced by name from any function
  - exist independent of any function’s execution
- We could use such variables to:
  - share information between functions
    - Not just between functions calling other functions
  - store things away from a function that will still be there when the function is called again

Global Variables

- A *global variable* is a variable that is defined outside any function
- It is allocated (a memory space is reserved for it) by the C compiler in a public area of memory.
- This space is not ever reclaimed for use for some other function or purpose

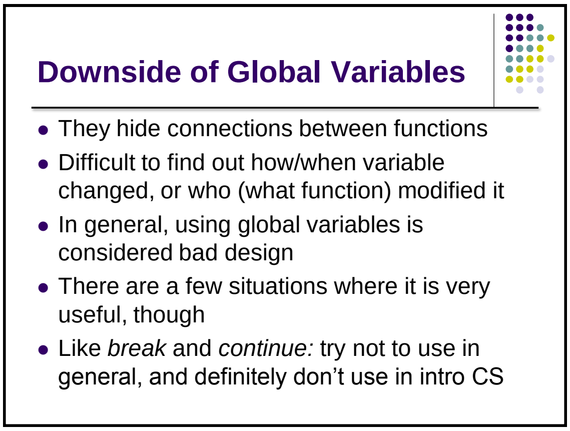
What Will This Output?

```
#include <stdio.h>

int Global_var;

int main() {
    int local_var = 10;

    /* Notice: I can reference global vars here */
    Global_var = 42;
    func_x();
    printf(“local=%d, global=%d\n”, local_var, Global_var);
}

void func_x(void) {
    int local_var;

    local_var = 1;
    Global_var = 47
}
```



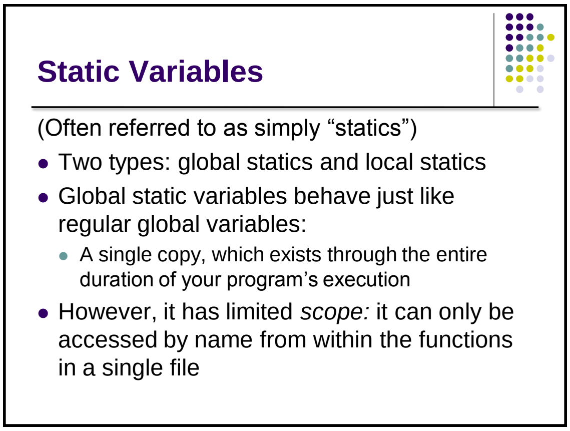
Benefits of Global Variables

- You can use global variables to hold values that you want to keep around for the entire duration of your program
- Global variables are also used to provide a place to store values used by a large number of functions
- Also can store things that are used by functions that are only connected through many layers of other intermediary functions



Downside of Global Variables

- They hide connections between functions
- Difficult to find out how/when variable changed, or who (what function) modified it
- In general, using global variables is considered bad design
- There are a few situations where it is very useful, though
- Like *break* and *continue:* try not to use in general, and definitely don’t use in intro CS



Static Variables

(Often referred to as simply “statics”)

- Two types: global statics and local statics
- Global static variables behave just like regular global variables:
  - A single copy, which exists through the entire duration of your program’s execution
- However, it has limited *scope:* it can only be accessed by name from within the functions in a single file

Compiling in Pieces

This needs further explanation:

- Your program can be written in several pieces (files)
- Each piece can be compiled separately, but then does not produce a complete working program by itself
- When all the pieces are compiled, they can be *linked* into a single *executable* [file]

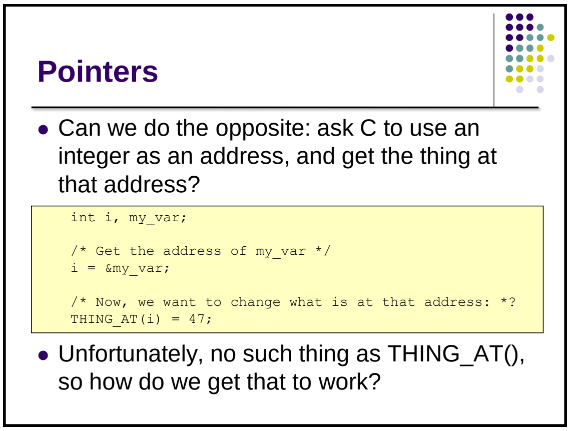
Local Static Variables

- Similar to global statics variables:
  - A single copy of the variable, which exists through the entire duration of your program's execution
  - This implies it continues to exist after a function returns
  - Unlike local vars, if a function calls *recurses* (i.e., calls itself) it does not get a new copy of local statics—those are shared across instances of a function
  - Scope is even more limited: to just inside the function it is declared in

Pointers

- We previously discussed getting a *reference* to (i.e., the address of) a variable, by using the ‘&’ operator:

```
int i;

/* pass the address of i to scanf(), so that scanf can
 * copy the number read in into that variable
 */
scanf(“%d”, &i);
```

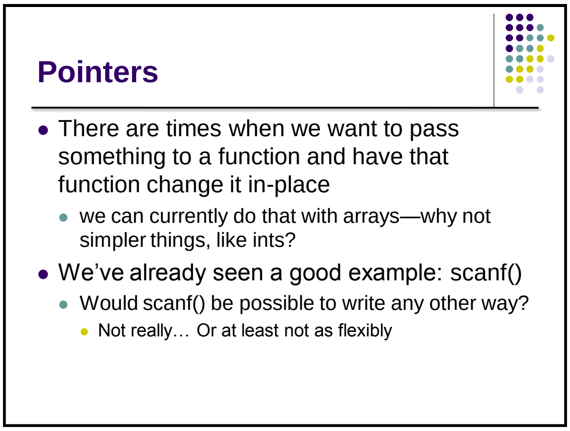


Pointers

- Can we do the opposite: ask C to use an integer as an address, and get the thing at that address?

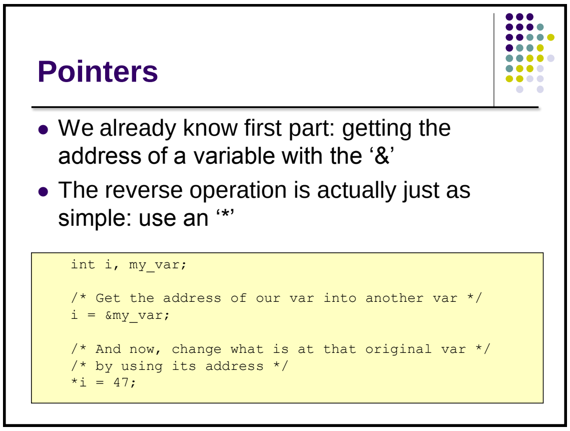
```
int i, my_var;

/* Get the address of my_var */
i = &my_var;

/* Now, we want to change what is at that address: *?
THING_AT(i) = 47;
```

- Unfortunately, no such thing as THING_AT(), so how do we get that to work?



Pointers

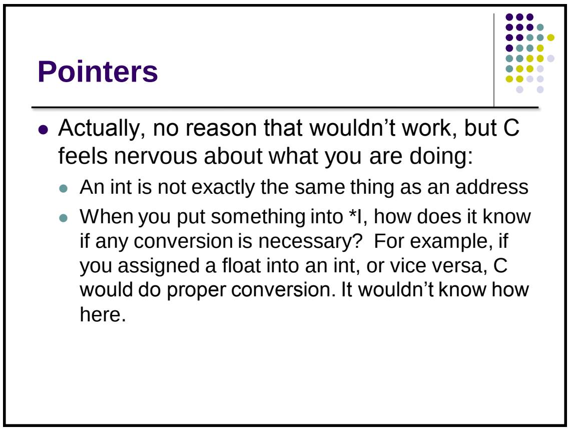
- There are times when we want to pass something to a function and have that function change it in-place
  - we can currently do that with arrays—why not simpler things, like ints?
- We've already seen a good example: scanf()
  - Would scanf() be possible to write any other way?
    - Not really… Or at least not as flexibly



Pointers

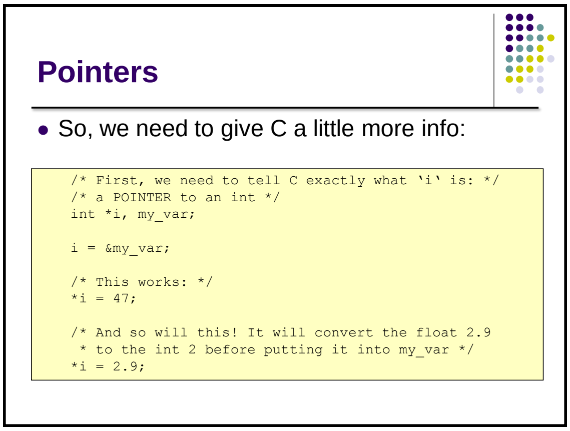
- We already know first part: getting the address of a variable with the '&'
- The reverse operation is actually just as simple: use an '*'

```
int i, my_var;

/* Get the address of our var into another var */
i = &my_var;

/* And now, change what is at that original var */
/* by using its address */
*i = 47;
```



Pointers

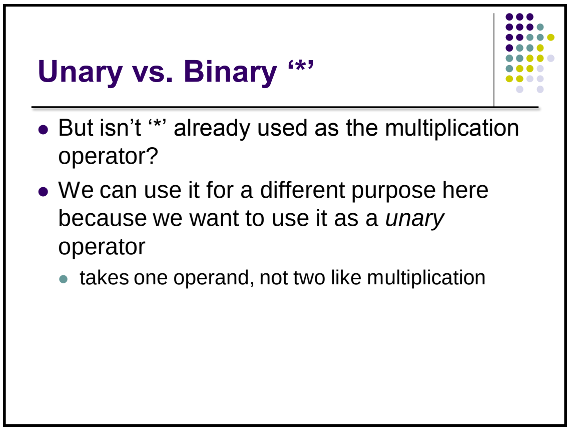
- Actually, no reason that wouldn't work, but C feels nervous about what you are doing:
  - An int is not exactly the same thing as an address
  - When you put something into *I, how does it know if any conversion is necessary? For example, if you assigned a float into an int, or vice versa, C would do proper conversion. It wouldn't know how here.



Pointers

- So, we need to give C a little more info:

```
    /* First, we need to tell C exactly what ‘i‛ is: */
    /* a POINTER to an int */
    int *i, my_var;

    i = &my_var;

    /* This works: */
    *i = 47;

    /* And so will this! It will convert the float 2.9
     * to the int 2 before putting it into my_var */
    *i = 2.9;
```



Unary vs. Binary '*'

- But isn't '*' already used as the multiplication operator?
- We can use it for a different purpose here because we want to use it as a *unary* operator
  - takes one operand, not two like multiplication

Unary vs. Binary ‘*’

- So if we see:

  x = 2 * y

  We know it’s a binary operator, because interpreting it as a unary would make the sentence syntactically incorrect
- Similarly, the following only has legal interpretation:

  x = 2 + * y

  We can reinforce this in the reader’s mind with better spacing:

  x = 2 + *y

Final Thoughts

- That’s really all there is to it!
- Recap:
  - We can get the address of *any variable* just by prepending an ‘&’ before the reference
    - This can be a simple variable (“&I”), or an element in an array (“&my_array[9]”, or even “&num[i + 2]”)
  - We can then use that address by prepending a ‘*’ in front of the value
    - This can be the value in a pointer variable (“i = &num; *i = 47”), or an expression (“*(i + 6)” or even “*&num”)
- Best to keep it simple, though…