CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

Subject Matter

- More I/O
- The Stack
- Arguments and return values

Your Goals

- Look up a function on cplusplus.com
- Homework 3

<stdio.h: printf>

specifier	**Output**	**Example**
d *or* i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
c	Character	a
S	String of characters	sample
P	Pointer address	b8000000
%	A % followed by another % character will write a single % to the stream.	%

- int printf (const char * format, ...);
- Format %[flags][width][.precision][length]specifier
  - - : left justify
  - + : requires sign on number
  - 0: left pads
- Returns:
  - Total number of characters written
  - Error code is negative number

<stdio.h: scanf>

- int scanf (const char * format, ...);
- Format %[*][width][length]specifier
  - * : read but don't store
  - width : # of characters to read
  - length: storage type
- Returns:
  - Total number of arguments filled
  - Or EOF (end of file)

specifier	Description	Characters extracted
`i`	Integer	Any number of digits, optionally preceded by a sign (+ or -).
`d` *or* `u`	Decimal integer	Any number of decimal digits (`0-9`), optionally preceded by a sign (+ or -). `d` is for a *signed* argument, and `u` for an *unsigned*.
`o`	Octal integer	Any number of octal digits (`0-7`),
`x`	Hexadecimal integer	Any number of hexadecimal digits (`0-9`, `a-f`, `A-F`), optionally preceded by `0x` or `0X`, and all optionally preceded by a sign (+ or -).
`f`, `e`, `g` `a`	Floating point number	A series of decimal digits
`c`	Character	The next character. If a *width* other than `1` is specified, the function reads exactly *width* characters and stores them in the successive locations of the array passed as argument.
`s`	String of characters	Any number of non-whitespace characters, with \o
`p`	Pointer address	A sequence of characters representing a pointer.
`[`*characters*`]`	Scanset	Any number of the characters specified between the brackets.
`[^`*characters*`]`	Negated scanset	Any number of characters none of them specified as *characters* between the brackets.
`n`	Count	No input is consumed. The number of characters read so far from `stdin` is stored in the pointed location.
`%`	`%`	A `%` followed by another `%` matches a single `%`.

File Streams

- Steams are a way to interact with hardware or data files in an abstract way
  - Don't care what it is, OS does that
- C handles these in `FILE` objects
  - `fprintf`
  - `fscanf`
  - `fputs`
  - `fgets`

```
FILE * fopen ( const char * filename,
const char * mode );
#include <stdio.h>
int main () {
  FILE * pFile;
  pFile = fopen ("myfile.txt","w");
  if (pFile≠NULL) {
    fputs ("fopen example",pFile);
    fclose (pFile);
  }
  return 0;
}
```

Getting arguments

BASH

```
#!/bin/bash
echo "0: $0"
for file in "$@"; do
    echo "$file"
done


while [ $# -gt 0 ]; do
  echo "$1"
  shift
done
```

C (printargs.c)

```
#include <stdio.h>


int main(int argc, char ** argv) {
  int k;
  printf("argc = %d\n", argc);
  for (k = 0; k < argc; k++)
   printf("argv[%d] = %s\n", k, argv[k]);
  return 0;
}
```

Working Memory - zones

- Lowest memory stores program instructions, then global variables (static constants, string literals)
- 'Heap' holds dynamically allocated variables ('new' or 'malloc' variables)
- 'Stack' holds current instructions, each function in a frame
  - 'Stack' memory implies that a frame is added, and then the last frame added is removed first
- The heap and stack grow dynamically. Meet in the middle ?= 'out of memory' error

Declare

Specifies Name & Datatype

No memory allocated

Can happen many times

```
extern int x;
int fun(float a);
```

Define

Allocates Memory

Functions get body

Can only happen once

```
int x;
int fun(float a)
{return (int)a;}
```

Initialize

Gives a value

Must be defined (have memory)

Could be re-set

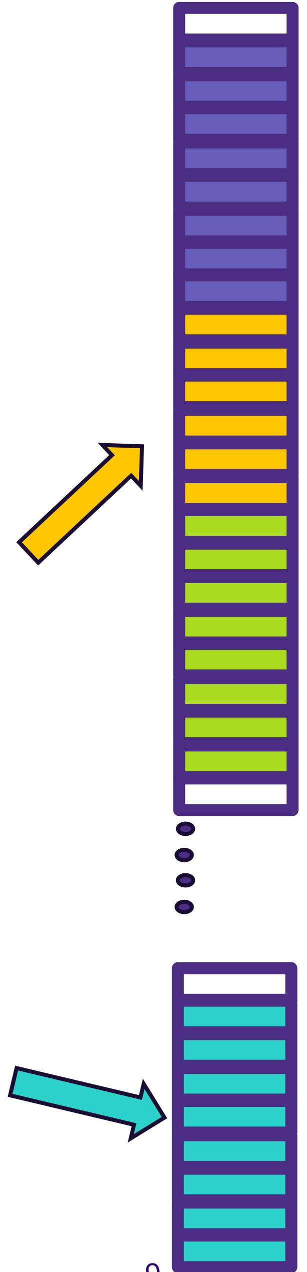
```
int x = 10;
```

Variable Scoping

- Variables (named things) accessible at certain times
- Scope and storage are related but not the same thing.

- Global variables
  - Scope: entire program
  - Violate encapsulation, but can be okay for truly global data (such as conversion factors)
- Static Global Variables
  - Scope: containing file/module
  - Can not be called from other files
- Static local variables
  - Scope: that function

- Local variables
  - Scope: within the block

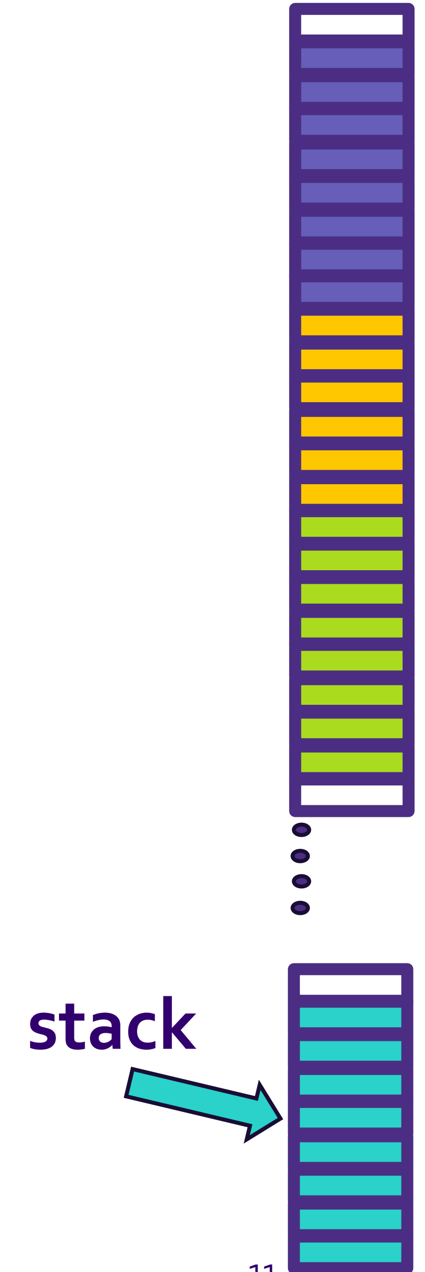


Source File Structure

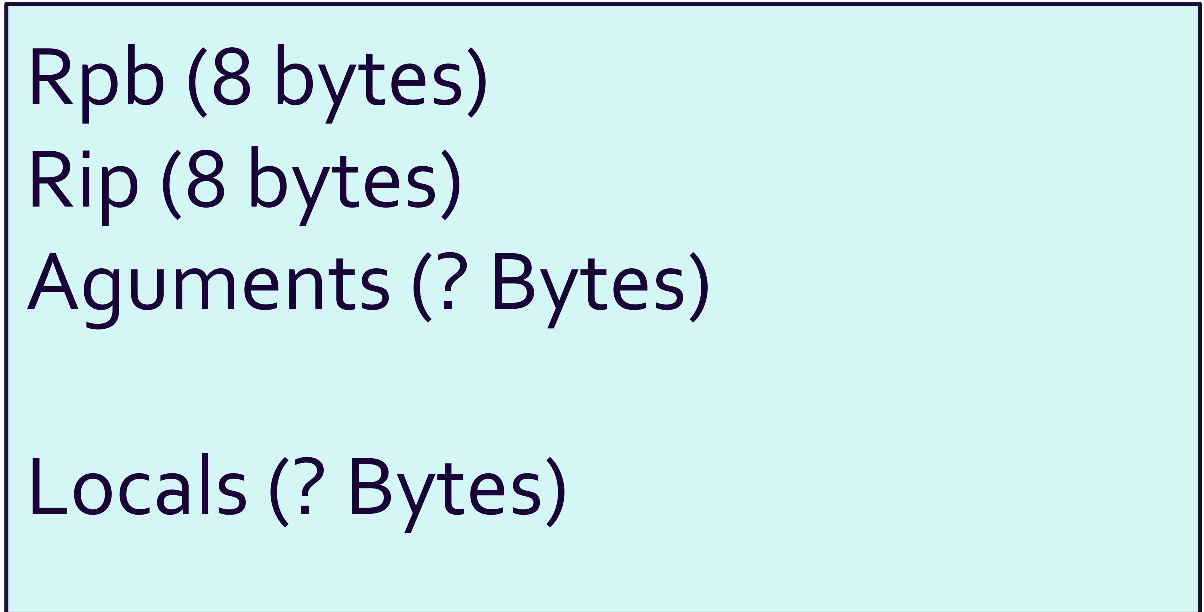
```
// includes functions & types defined elsewhere
#include <stdio.h>
#include "localstuff.h"
// symbolic constants
#define MAGIC 42
// global variables (if any)
static int days_per_month[ ] = { 31, 28, 31, 30,
  …};
// function prototypes
// (to handle "declare before use")
void some_later_function(char, int);
// function definitions
void do_this( ) { … }
char *return_that(char s[ ], int n) { … }
int main(int argc, char ** argv) { … }
```

The Stack (local variables)

- The stack stores active functions and local variables
- Each newly called code block pushes a **frame** to the stack
  - Frame includes **`rip`** - return instruction pointer
    - (next instruction)
  - **`rpb`** - address of previous frame on stack
  - Address of argument list: space to store all the arguments
  - Address of locals: space to store the local variables
- When the code block ends, the **frame** is popped off the stack
  - Arguments and locals popped off stack – no longer in memory
  - Size of variables must be known (to allocate frame size)
    - Arrays must be of a defined size



The Stack (memory allocation)



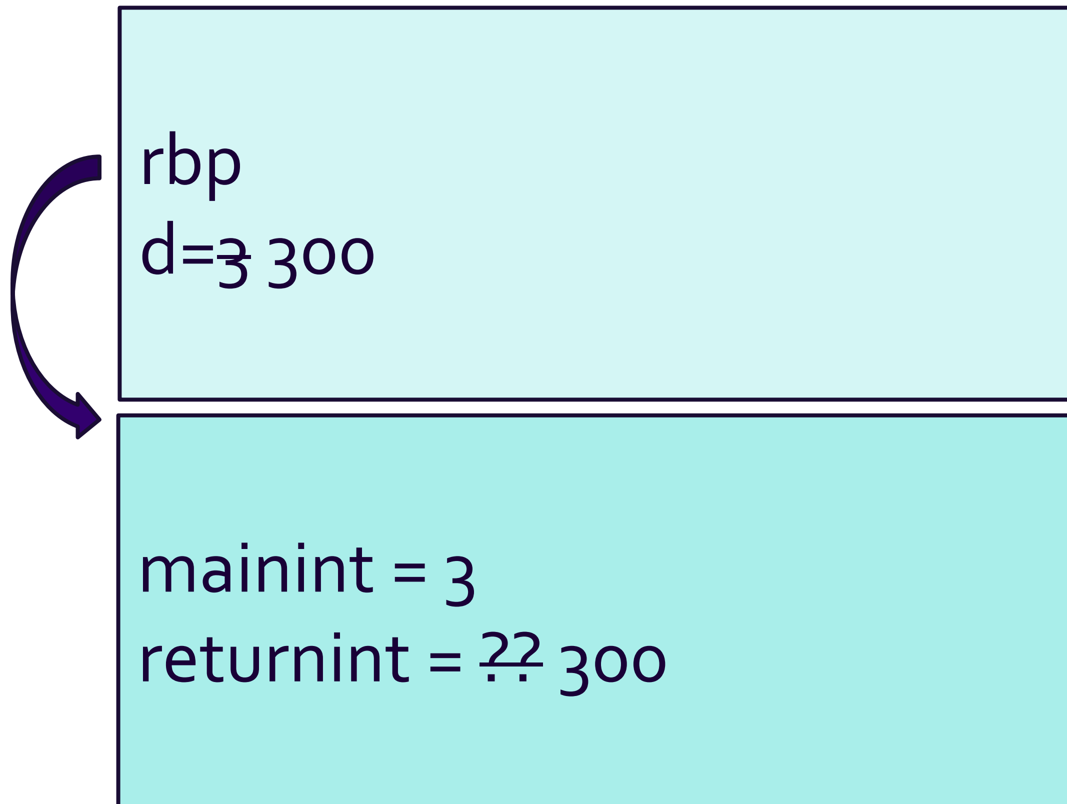
Callee frame

Caller frame

stack

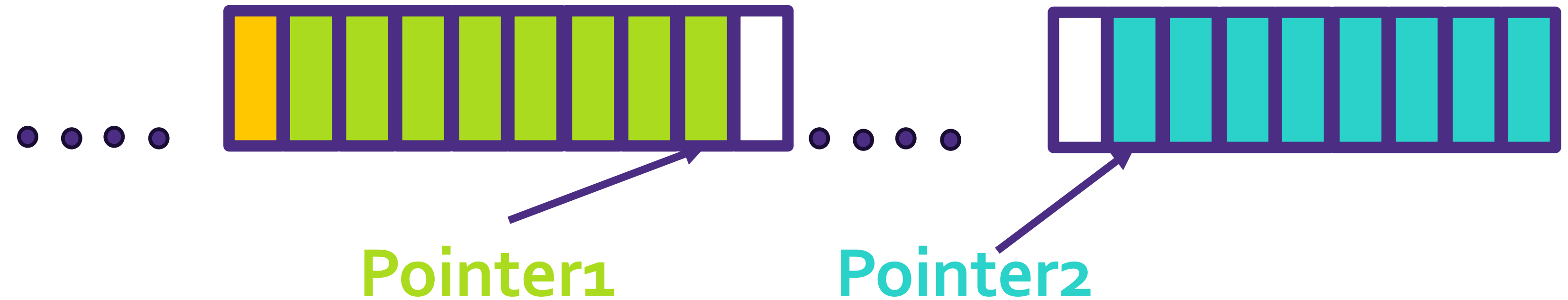
Function arguments

'local' variables in the new stack frame.




```
int demoint (int d) {
  d = d*100;
  return d;
}

int main () {
  int mainint = 3;
  int *mainintptr = &mainint;
  int returnint;
  // test passing the integer
  returnint = demoint(mainint);
```

So, what is a pointer?

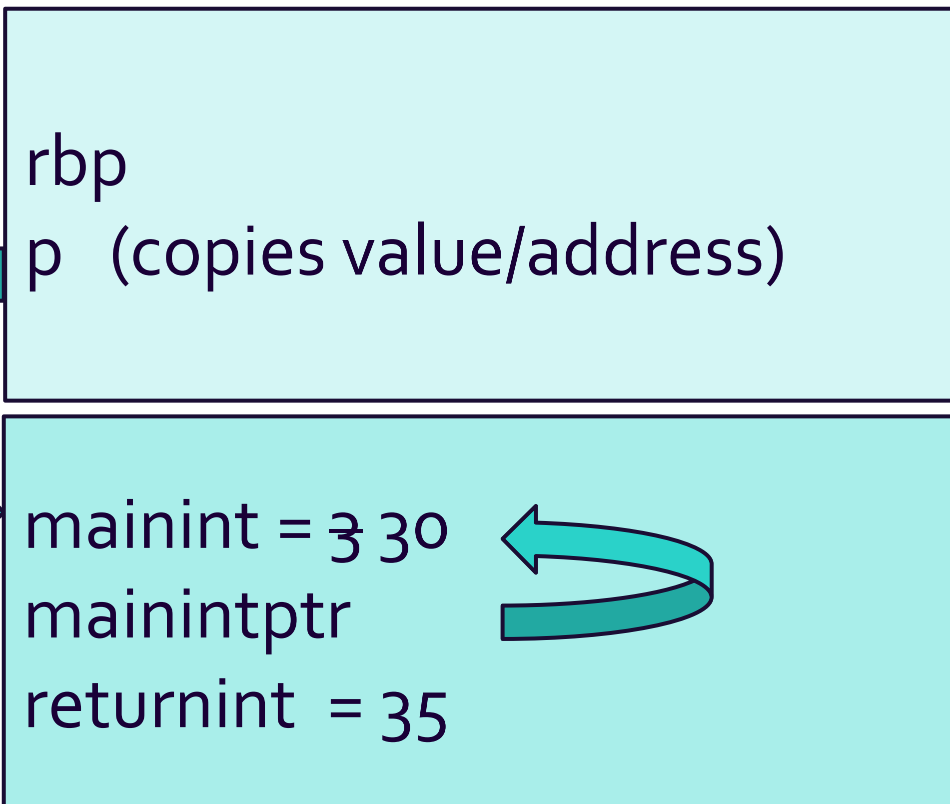


- Point to an address in memory

Code	Meaning
`int x = 4;`	Variable called 'x' of type 'int' storing value of '4'
`int *xPtr = &x;`	Variable called 'xPtr' ot type 'pointer to int', given value of 'the location of x'
`int xCopy = *xPtr;`	Variable called 'xCopy' of type 'int' storing value at the location pointed to by xPtr
`int *noPtr = NULL;`	Variable called 'noPtr' of type 'pointer to int' correctly set to 'no location'

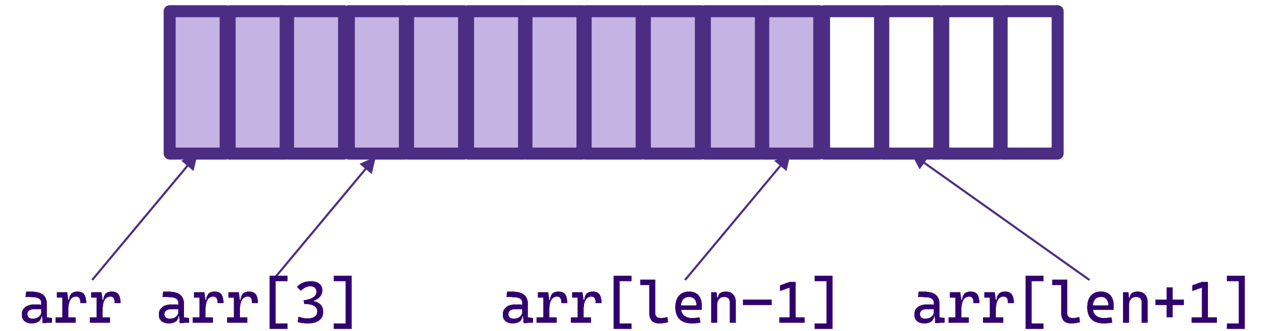
Pointer arguments

'local' variables in the new stack frame.




```
int demopointer (int *p) {
  *p = *p *10;
  return (*p + 5);  // return an int
}


int main (int argc, char **argv) {
  int mainint = 3;
  int *mainintptr = &mainint;
  int returnint;
  returnint = demopointer(mainintptr);
```

Array definition



- Arrays are contiguous blocks of memory
  - `Datatype arr[len]`
  - Has type: `Datatype*`
- Definition means allocating memory
  - You MUST tell the array how many elements it has
  - How much memory to allocate
- If you need to have an array for which you don't know the size
  - Declare a pointer, set it to NULL
  - Dynamically allocate

Array arguments

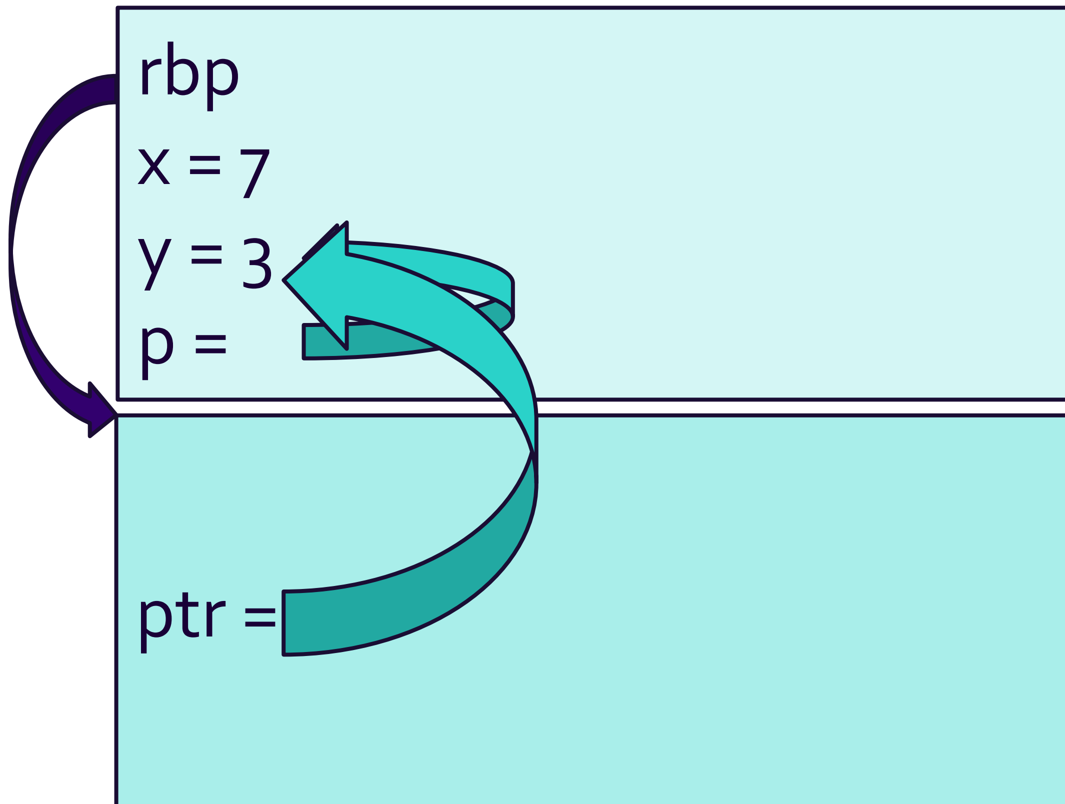
rbp
myarray (copies value/address)
myarraylen = 7

len = 7
mainarray = (address of 7x4 bytes of memory, 11)

```
void arrfunc (int *myarray, int
 myarraylen) {

  for(int i = 0; i < myarraylen; i++) {

    myarray[i] = 10*myarray[i];

}}

int main() {

  int len = 7;

  int mainarray[] = {11, 22, 33, 44,
 55, 66, 77}; // easy initialization

  arrfunc (mainarray, len);
```

Dangling Pointers

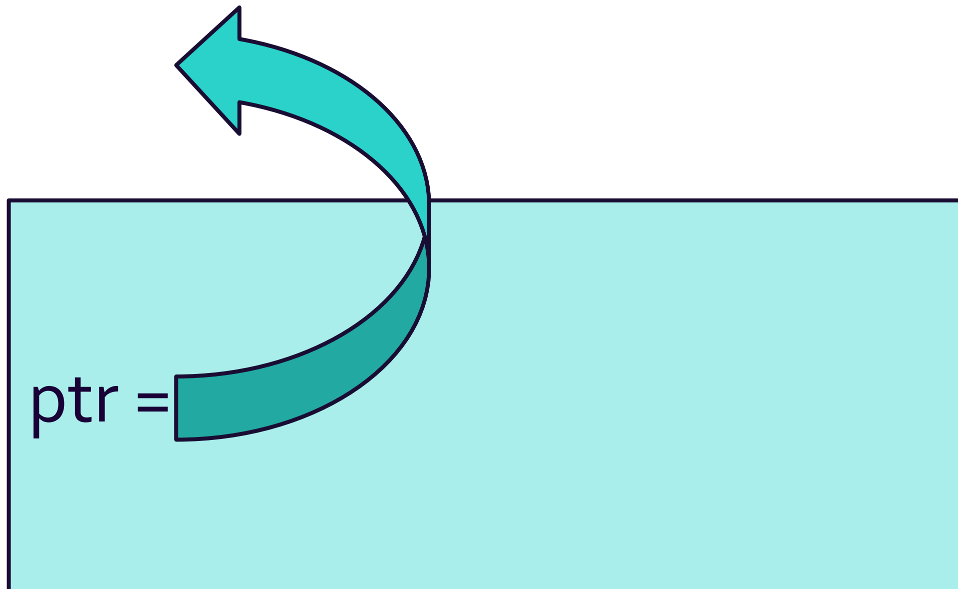
Pointers hold address, don't know if memory is allocated




```
int* f(int x) {
  int *p;
  int y = 3;
  p = &y; /* ok */
  return p;
}

int main(int argc, char **argv) {
  int* ptr;
  ptr = f(7);
  printf ("in main: *p=%d\n", *ptr);
}
```

Dangling Pointers 2

Pointers hold address, don't know if memory is allocated




```
int* f(int x) {
  int *p;
  int y = 3;
  p = &y; /* ok */
  return p;
}

int main(int argc, char **argv) {
  int* ptr;
  ptr = f(7);
  printf ("in main: *p=%d\n", *ptr);
}
```

BACKUPS

Hello World

- Compile at a bash command line with `gcc hello.c`
  - Creates an executable `a.out`
- `gcc -Wall -std=c11 -o hello hello.c`
  - `Wall` turns on all warnings
  - `C11` specifies C11 standard
  - Creates executable hello
- Run: `./a.out` or `./hello`
- Exits with `0` (`return 0;`)

```
echo 'alias ccomp="gcc -Wall -std=c11 -o"' >> .bashrc
```

```c
#include <stdio.h>
#define REPS 5


/**
 * Compile this file with:
 *      gcc -o hello hello.c
 * Run the program:
 *      ./hello
 */
int main(int argc, char **argv) {
  for (int i=0; i<REPS; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```