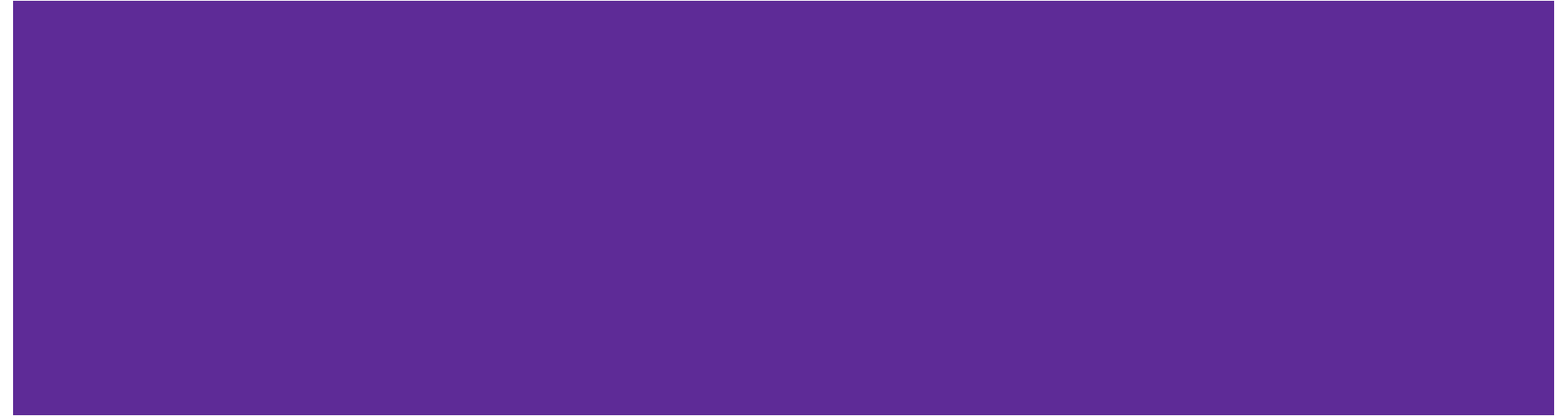# What do you think?

Work with a partner(s):

1. Write C code to declare an array with 5 elements, and fill it with the numbers 55,44,33,22 & 11

2. Write a C function that will modify the array to hold the numbers 5,4,3,2 & 1

Note: #2 might be pretty hard right now - check back at the end of lecture

CSE 374 Lecture 10

(C continued - I/O & Arguments)

I/O : Printf, scanf

Printf and scanf are two I/O functions, prototyped in stdio.h

➔ Printf (print-format)
➔ int printf(const char *format, …)
➔ 'Format' is a string that can contain format tags
➔ + additional arguments to match tags
➔ Number of arguments better match number of %
➔ Corresponding arguments better have the right types (%d, int; %f, float; %e, float (prints scientific); %s, \0- terminated char*; … Compiler might check, but not guaranteed
   ◆ best case scenario: you crash
➔ `printf("%s: %d %g\n", p, y+9, 3.0)`

➔ scanf (gets input, formatted)
➔ int scanf(const char *format, …)
➔ 'Format' is a string that can contain format tags
➔ + additional arguments to match tags - should be pointers to the right data type so input can be stored in them
➔ `scanf("%d %s", &n, str);`
➔ `scanf("%*s %d", &a);`
   ◆ %*s ignores string until space, then reads in an integer

Formatting Tricks

```
/* printf example */
#include <stdio.h>

int main()
{
    printf ("Characters: %c %c \n", 'a', 65);
    printf ("Decimals: %d %ld\n", 1977, 650000L);
    printf ("Preceding with blanks: %10d \n", 1977);
    printf ("Preceding with zeros: %010d \n", 1977);
    printf ("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
    printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
    printf ("Width trick: %*d \n", 5, 10);
    printf ("%s \n", "A string");
    return 0;
}
```

`%[flags][width][.precision][length]specifier`

Where the *specifier character* at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

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
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a `signed int`. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

flags	description
-	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see *width* sub-specifier).

width	description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	description
.number	For integer specifiers (d, i, o, u, x, X): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for *precision*, 0 is assumed.
.*	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

From cplusplus.com

Using Files

as `stdio.h` in the C language). This library uses what are called **streams** to operate with physical devices such as keyboards, printers, terminals or with any other type of files supported by the system. Streams are an abstraction to interact with these in an uniform way; All streams have similar properties independently of the individual characteristics of the physical media they are associated with.

Streams are handled in the `cstdio` library as pointers to <u>FILE</u> objects. A pointer to a <u>FILE</u> object uniquely identifies a stream, and is used as a parameter in the operations involving that stream.

```
13   char *file_to_open = argv[1];
14   FILE *file = fopen(file_to_open, "r");
15
16   // fopen will return NULL if it fails (look at man page)
17   // http://www.cplusplus.com/reference/cstdio/fopen/
18   if (file) {
19     printf("File is #%d\n", fileno(file));
20     do_something_with_a_file(file);
21     fclose(file);
22   } else {
23     perror("Oh noes! Failed to open file");  // prints a st
24     return EXIT_FAILURE;
25   }
```

Learning about iostreams

https://cplusplus.com/reference/cstdio/

fprintf
<cstdio>

```
int fprintf ( FILE * stream, const char * format, ... );
```

Write formatted data to stream

type
FILE
<cstdi

Object containing information to control a stream

Object type that identifies a stream and contains the information needed to control it, including a pointer to its buffer, its position indicator and all its state indicators.

FILE objects are usually created by a call to either fopen or tmpfile, which both return a pointer to one of these objects.

(f)scanf:

- **Whitespace character:** the function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab c... see isspace). A single whitespace in the *format* string validates any quantity of whitespace c... extracted from the *stream* (including none).
- **Non-whitespace character, except format specifier (%):** Any character that is not eith... whitespace character (blank, newline or tab) or part of a *format specifier* (which begin with a... causes the function to read the next character from the stream, compare it to this non-white... character and if it matches, it is discarded and the function continues with the next characte... the character does not match, the function fails, returning and leaving subsequent character... stream unread.
- **Format specifiers:** A sequence formed by an initial percentage sign (%) indicates a format s... which is used to specify the type and format of the data to be retrieved from the *stream* and... the locations pointed by the additional arguments.

```c
int main() {
    FILE* ptr = fopen("abc.txt", "r");
    if (ptr == NULL) {
        printf("no such file.");
        return 0;
    }


    /* Assuming that abc.txt has content in below
    format
    NAME AGE CITY
    abc  12 hyderabad
    bef  25 delhi
    cce  65 bangalore */
    char buf[100];
    while (fscanf(ptr, "%*s %*s %s ", buf) == 1) {
        printf("%s\n", buf);
    }
    return 0;
}
```

Pointer Review

```
int var = 349;
int *varptr = &var;
```

Pointers point to an address in memory
`&x` returns the address

Declare a pointer to a pointer type and it has a specific type/size of memory:

`T *x;` or `T* x;` or `T * x;` or T*x
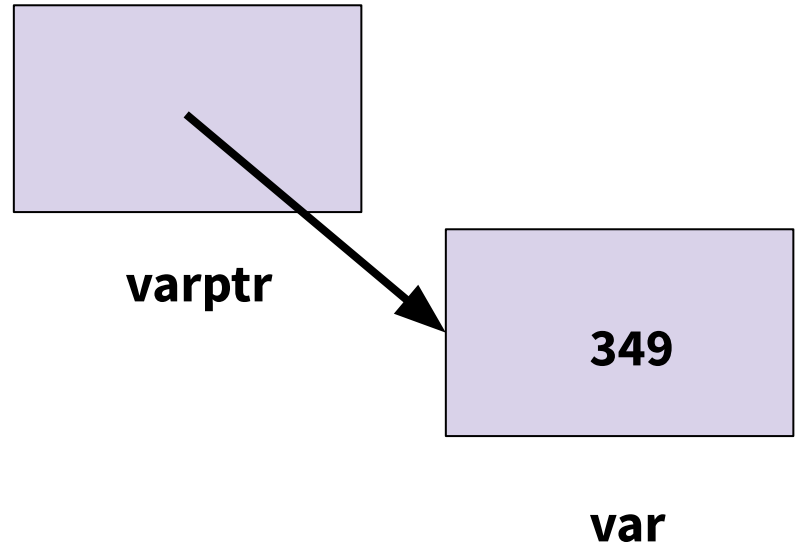(T is a type, x is a variable)

An expression to dereference a pointer
`*x` (more generally *expression)
    Dereference - get the value at the address

Arrays have an implicit pointer type
`T = x[n]` implies x is of type T*

varptr

349

var

Pointers to pointers

Levels of pointers make sense:

I.e.: `argv, *argv, **argv`

Or: `argv, argv[0], argv[0][0]`

But

`&(&p)` doesn't make sense

```
void f(int x) {
    int*p = &x;
    int**q = &p;
    // x, p, *p, q, *q, **q
}
```

Integer, pointer to integer, pointer to pointer to integer

`&p` is the address of '`p`',

`&(&p)` would be the address of the address of `p`, but that value isn't stored separately anywhere and doesn't have an address

Try using `printf ("The address of x is %p\n", &x);`

Storage

- Variables need a place to live in memory
- Get 'allocated' a physical space in memory (with an address)
- Size of memory allocation depends on datatype
- Get 'deallocated' to release the space in memory

code   *globals*   *heap ->*   *<- stack*

. . . .

Address '0'

Address '4'

Address '$2^{64}-1$' or '$2^{32}-1$'
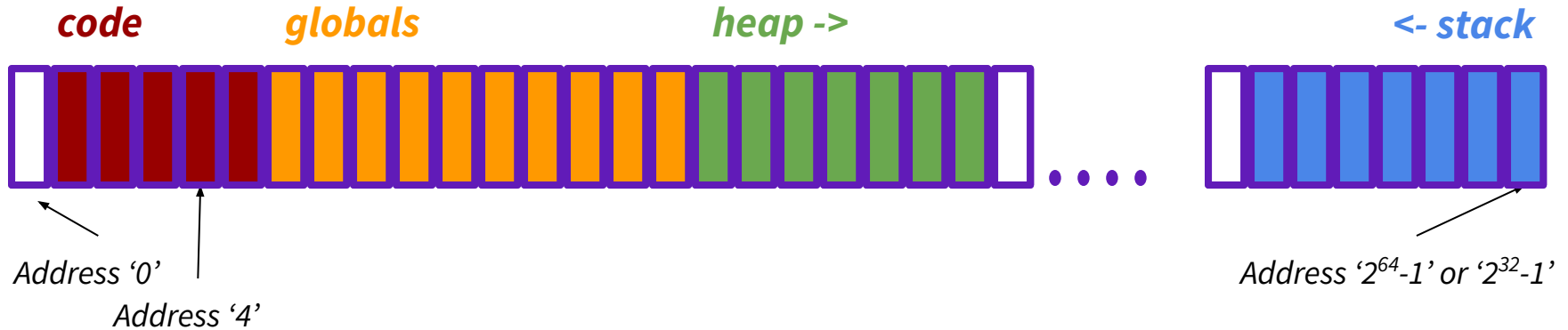
Scope

Variables may be accessed by the caller only at certain times - this is scope

Scope and storage are related, but not the same thing

- **Global variables**
  - Scope: entire program
  - Not desirable (violate encapsulation) But can be OK for truly global data like conversion tables, physical constants, etc.
- **Static global variables**
  - Scope: containing file
  - Static functions cannot be called from other files
- **Static local variables**
  - Scope: that function, rarely used
- **Local variables (automatic)**
  - Scope: that block – With recursion, multiple copies of same variable (one per stack frame/function activation)

allocated before main, deallocated after main - memory in 'global' block

allocated "when reached" deallocated "after block" - memory in frame on stack

```c
// includes for 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) { … }
```

Includes declarations & prototypes you might want to share.

Global variables & forward declarations go first.

Source File Structures

Stuff in function definitions is local to those functions

The stack

Stack stores active functions & <u>local</u> variables
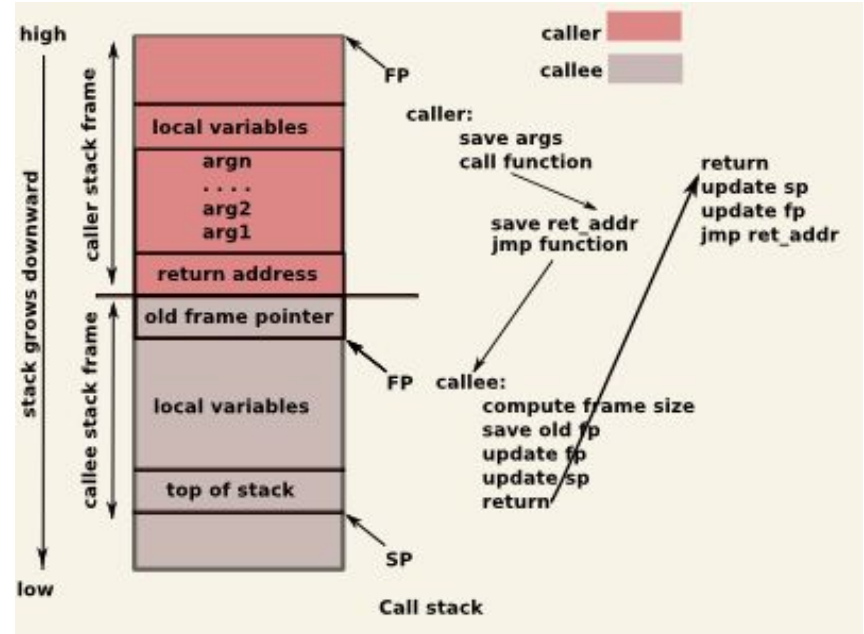
Frames deleted when function returns
    Local variables do not persist

Local variables must have defined size
    Can not make run-time adjustments
      (Arrays must have length)

caller
callee

high

caller stack frame

local variables

argn
. . . .
arg2
arg1

return address

old frame pointer

callee stack frame

local variables

top of stack

low

stack grows downward

FP

FP

SP

caller:
    save args
    call function

save ret_addr
jmp function

return
update sp
update fp
jmp ret_addr

callee:
    compute frame size
    save old fp
    update fp
    update sp
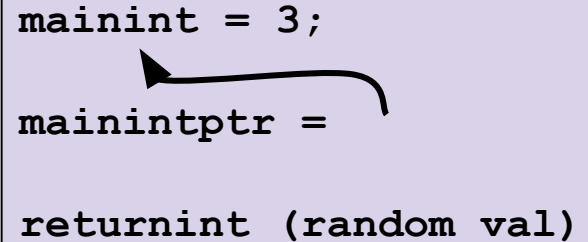   return

Call stack

Arguments

Demo

Storage allocation and variable scope is like local variables (i.e. space is part of the function frame added to the stack, and the variable may be used in the function).

All arguments passed by value. (i.e. a copy of the value is made and assigned to the variable.)

———

Stack (main)

```
26   int main (int argc, char **argv) {
27      int mainint = 3;
28      int *mainintptr = &mainint;   // hold address of mainint
29      int returnint;
30
```

```
mainint = 3;

mainintptr =

returnint (random val)
```

Stack (demoint)

d = 3;

mainint = 3;

mainintptr =

returnint (random val)

```
5   // demo functions here
6   int demoint (int d) {
7     d = d*100;
8     return d;
9   }
```

```
32        // test passing just the integer
33    returnint = demoint(mainint);
```
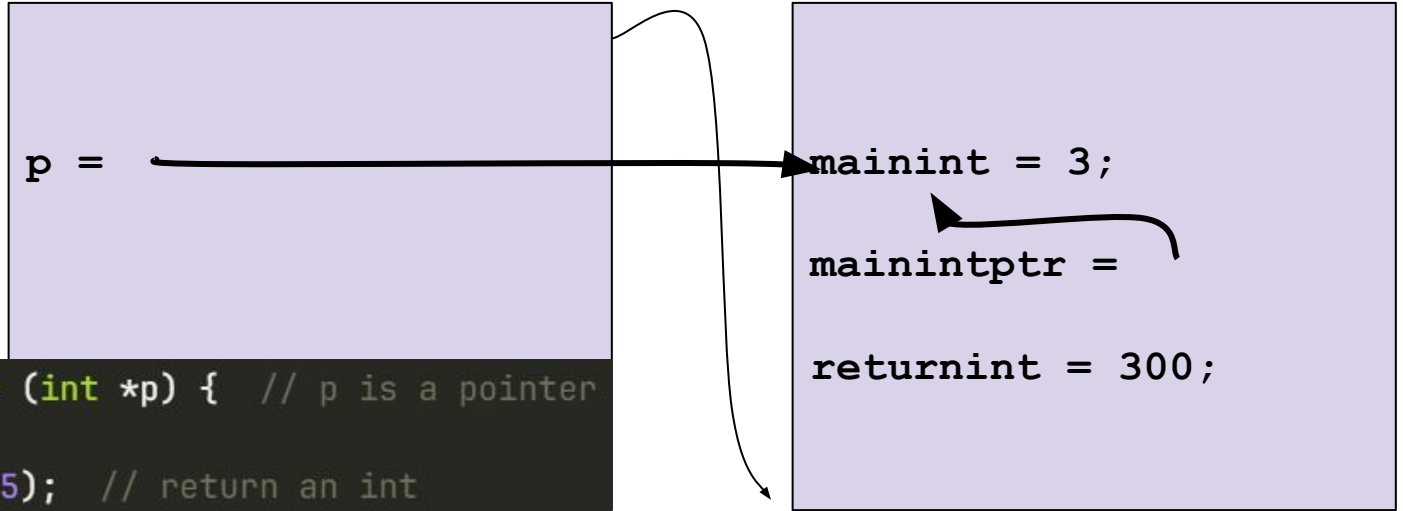
Stack (dempointer)
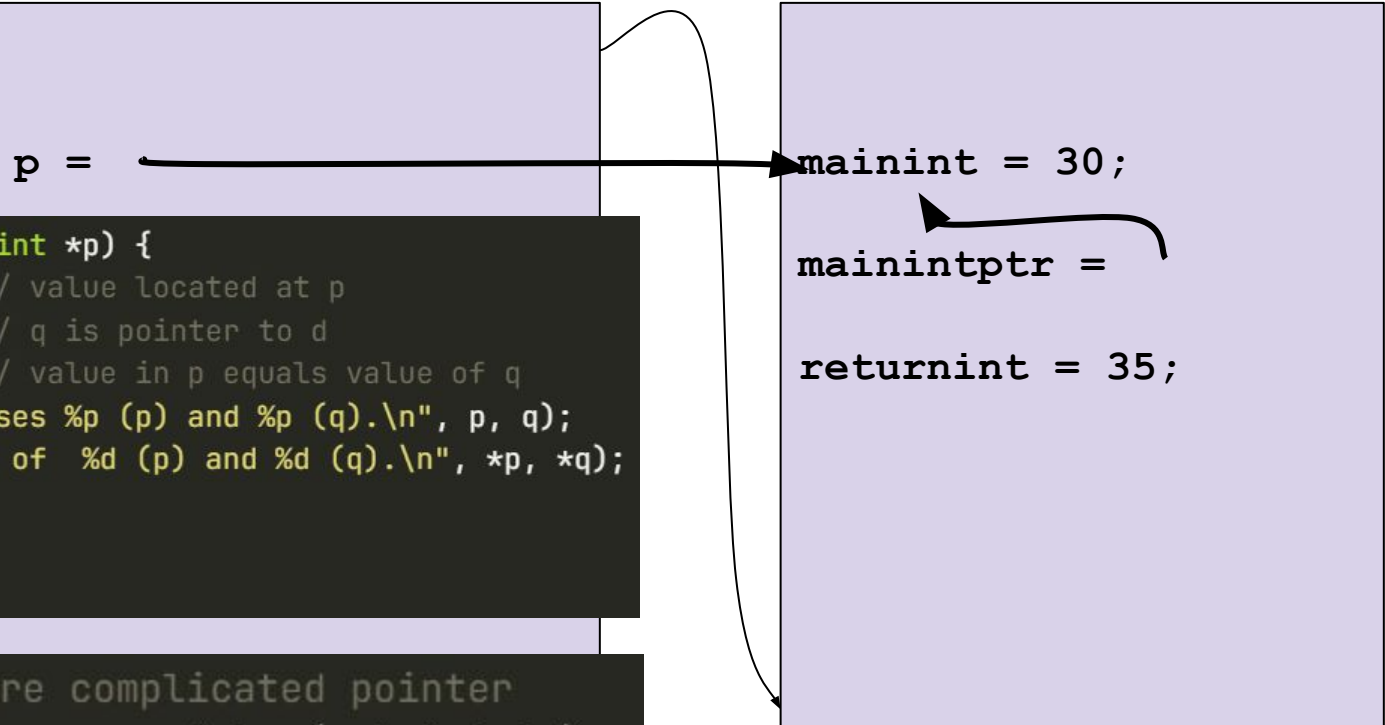
p =

mainint = 3;

mainintptr =

returnint = 300;

```
11   int demopointer (int *p) {  // p is a pointer
12     *p = *p *10;
13     return (*p + 5);  // return an int
14   }
```

```
36       // test passing a pointer
37     returnint = demopointer(mainintptr);
```

Stack (morepointer)

p =

mainint = 30;

mainintptr =

returnint = 35;

```
16  int morepointer (int *p) {
17    int d = *p;    // value located at p
18    int *q = &d;   // q is pointer to d
19    *p = *q;       // value in p equals value of q
20    printf ("Addresses %p (p) and %p (q).\n", p, q);
21    printf ("Values of  %d (p) and %d (q).\n", *p, *q);
22    *p = *q + 1;
23    return *q + 2;
24  }
```

```
40      // test more complicated pointer
41    returnint = morepointer(mainintptr);
```

Stack (morepointer)

p =

d = 30

q =

mainint = 30;

mainintptr =

returnint (random val)

```
16  int morepointer (int *p) {
17    int d = *p;      // value located at p
18    int *q = &d;     // q is pointer to d
19    *p = *q;         // value in p equals value of q
20    printf ("Addresses %p (p) and %p (q).\n", p, q);
21    printf ("Values of  %d (p) and %d (q).\n", *p, *q);
22    *p = *q + 1;
23    return *q + 2;
24  }
```
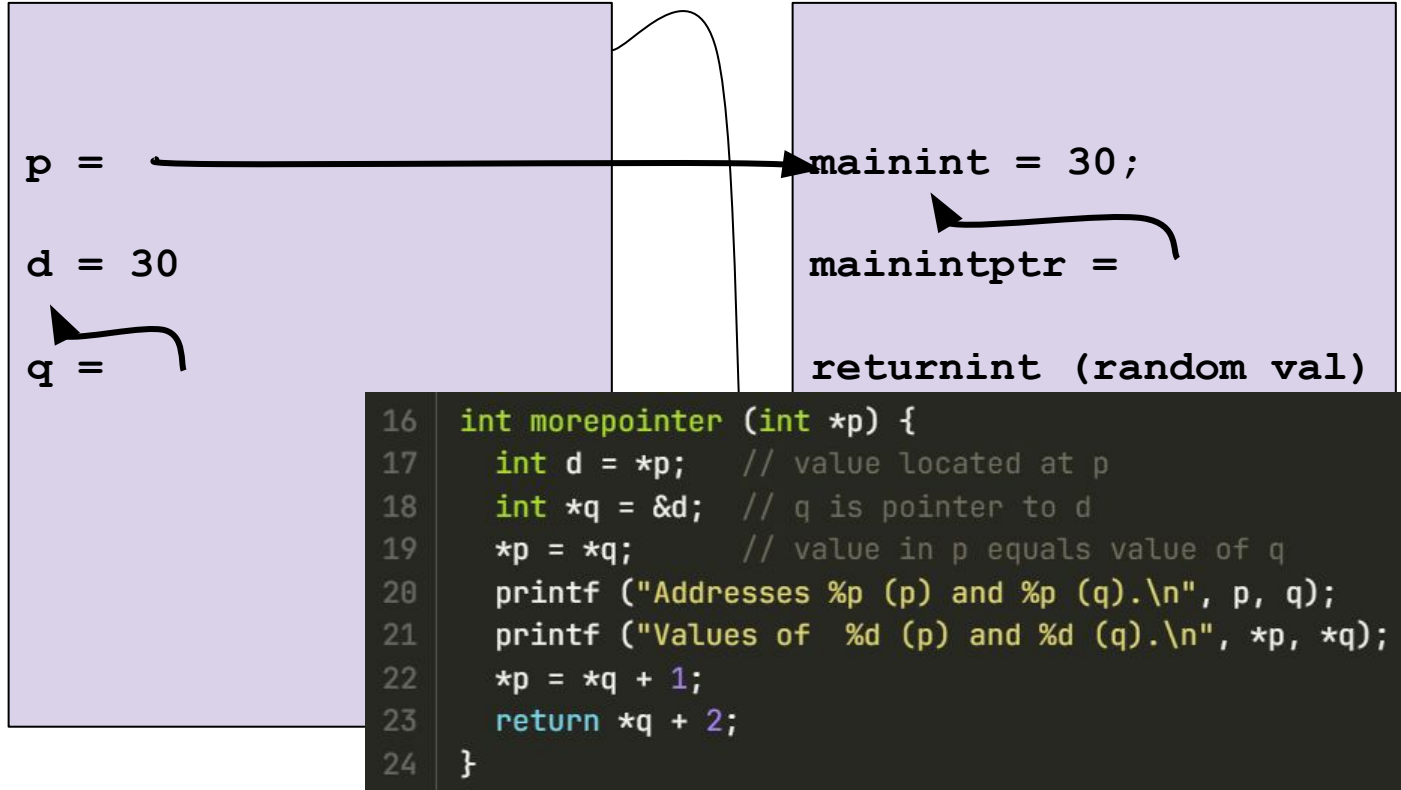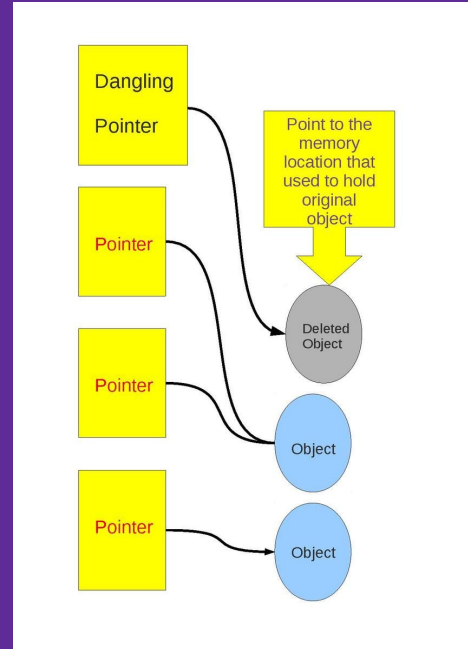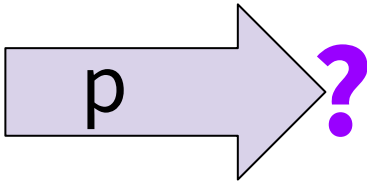
Dangling pointers

Pointers referring to memory that has been released *(Demo)*

Garbage collecting languages (like Java) only delete memory that is unreachable to avoid this problem.
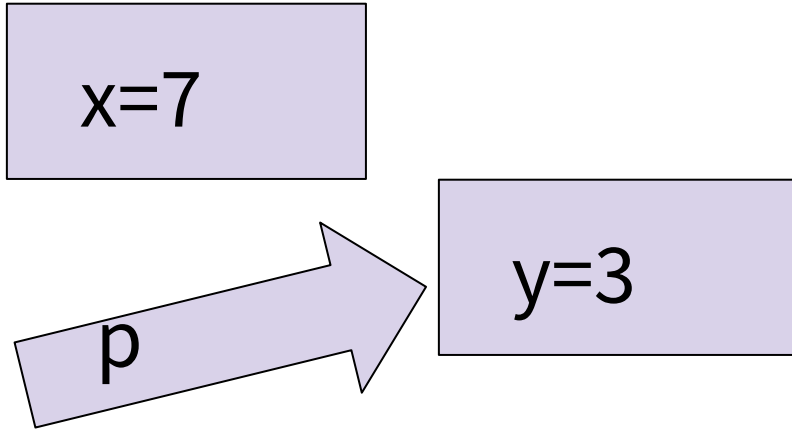
Dangling Pointers (line 4)

x=7

p ➔ **?**

```
3   int* f(int x) {
4     int *p;
5     if (x) {
6       int y = 3;
7       p = &y; /* ok */
8       printf ("y=%d and *p=%d\n", y, *p);
9     } /* ok, but p now dangling */
10    printf ("*p=%d\n", *p);
11    /* y = 4 does not compile */
12
13    *p = 7; /* could CRASH but probably not */
14    printf ("*p=%d\n", *p);
15    return p; /* uh-oh, but no crash yet */
16  }
17
18  void g(int *p) {
19    *p = 123;
20    f(123);   // now just to demonstrate the stack
21    printf ("in g: *p=%d\n", *p);
22  }
23
24  int main(int argc, char **argv) {
25    g(f(7)); /* HOPEFULLY YOU CRASH (but maybe not) */
26  }
```
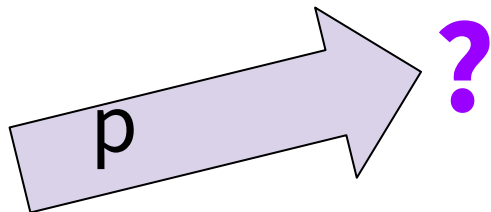
Dangling Pointers (line 7)

x=7

y=3

p

```
 3  int* f(int x) {
 4    int *p;
 5    if (x) {
 6      int y = 3;
 7      p = &y; /* ok */
 8      printf ("y=%d and *p=%d\n", y, *p);
 9    } /* ok, but p now dangling */
10    printf ("*p=%d\n", *p);
11    /* y = 4 does not compile */
12
13    *p = 7; /* could CRASH but probably not */
14    printf ("*p=%d\n", *p);
15    return p; /* uh-oh, but no crash yet */
16  }
17
18  void g(int *p) {
19    *p = 123;
20    f(123);   // now just to demonstrate the stack
21    printf ("in g: *p=%d\n", *p);
22  }
23
24  int main(int argc, char **argv) {
25    g(f(7)); /* HOPEFULLY YOU CRASH (but maybe not) */
26  }
```

Dangling Pointers (line 13)

x=7

p **?**

```
3   int* f(int x) {
4     int *p;
5     if (x) {
6       int y = 3;
7       p = &y; /* ok */
8       printf ("y=%d and *p=%d\n", y, *p);
9     } /* ok, but p now dangling */
10    printf ("*p=%d\n", *p);
11    /* y = 4 does not compile */
12
13    *p = 7; /* could CRASH but probably not */
14    printf ("*p=%d\n", *p);
15    return p; /* uh-oh, but no crash yet */
16  }
17
18  void g(int *p) {
19    *p = 123;
20    f(123);  // now just to demonstrate the stack
21    printf ("in g: *p=%d\n", *p);
22  }
23
24  int main(int argc, char **argv) {
25    g(f(7)); /* HOPEFULLY YOU CRASH (but maybe not) */
26  }
```

Arrays again

"A reference to an object of type array-of-T which appears in an expression decays (with three exceptions) into a pointer to its first element; the type of the resultant pointer is pointer-to-T."

http://c-faq.com/aryptr/aryptrequiv.html

Right: x is the array, which decays to a pointer to an int and &x returns a pointer to the entire array.

```
void f1(int* p) {  // takes a pointer
  *p = 5;
}
int* f2() {
    int x[3];     // x on stack, is pointer
    x[0] = 5;
    (&x)[0] = 5; // address of x, points to
                  // same place but different T
    *x = 5;       // put value at location x
    *(x+0) = 5;   // Also put value at x
    f1(x);
    f1(&x);      // wrong – watch types!
    x = &x[2]; // No!  X isn't really a pointer
    int *p = &x[2];
    return x; // correct type, but is a
              // dangling pointer
}
```

Pointer arithmetic

- If p has type T* or T[] and   *p has type T
- If p points to one item of type T, p+1 points to a place in memory for the next item of type T
    - So, p[0] is one item of type T, p+i = p[i]
- T[] always has type T*, even if it is declared as T[]
    - Implicit array promotion

    Result: Arrays are always passed by reference, not by value. (The information passed is the address of where the values are stored.)

https://chortle.ccsu.edu/CPuzzles/CPuzzlesMain.html