Lecture 10: Pointers

Sarah Nadi
nadi@ualberta.ca
Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology
Winter 2018

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro. Most examples taken from K.N. King's book]

Agenda

- Memory layout of a C program
- Pointer variables
- The address and indirection operators
- Pointer assignment
- Pointers as arguments
- Pointers as return values

Readings

- Textbook Chapter 11

Memory

- Main memory is divided into bytes
- A byte stores 8 bits of information
- Each byte has a unique hexadecimal address, e.g., 0xcc1ebd5cffffe7b0

Recall...

- All variables in your program are stored some place in memory in binary form.

```
int x;
x = …;
```

Value

Memory Address								
…								
0x15	1	0	1	1	0	1	0	0
0x16	0	0	0	1	0	0	0	0
0x17	0	0	0	1	0	0	0	0
0x18	1	0	1	1	0	0	0	1
…								
…								
…								

Recall…

- All variables in your program are stored some place in memory in binary form.

```
int x;
x = …;
```

Value

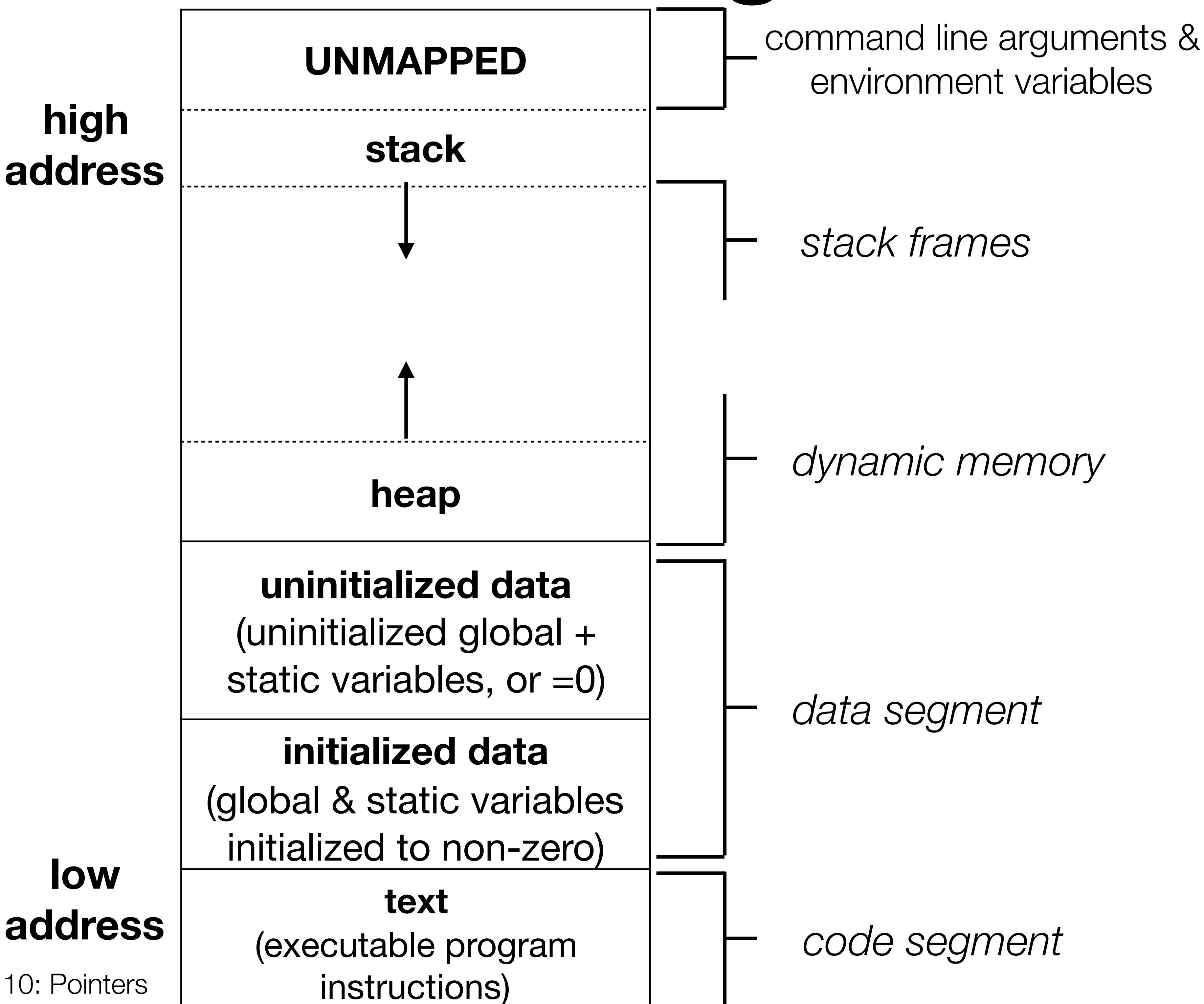
Memory Address								
…								
0x15	1	0	1	1	0	1	0	0
0x16	0	0	0	1	0	0	0	0
0x17	0	0	0	1	0	0	0	0
0x18	1	0	1	1	0	0	0	1
…								
…								
…								

`x` is stored in 4 bytes starting at address 0x15, so the address of x is 0x15 (even though it occupies 4 bytes)

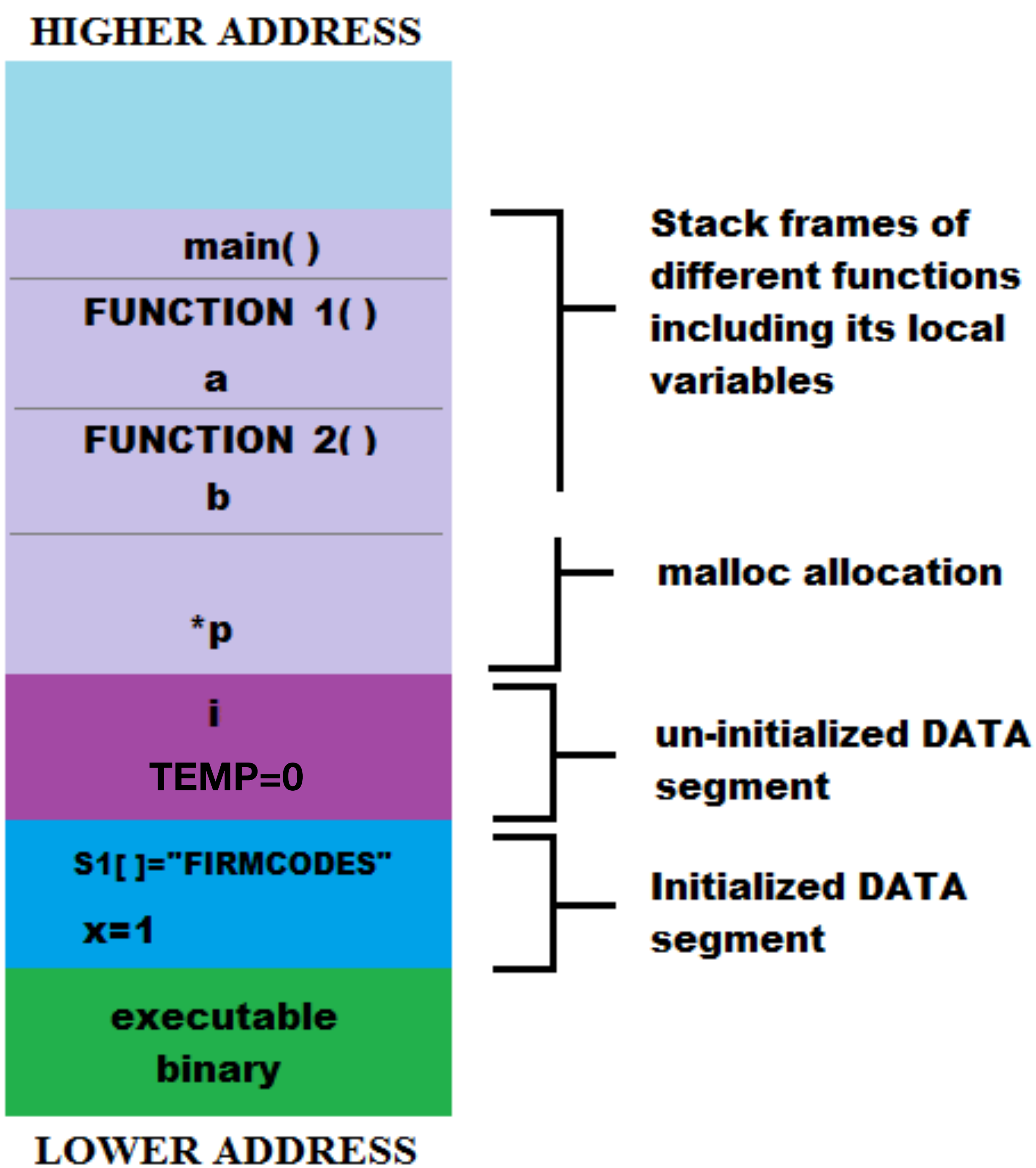
Memory *Cont'd*

- An executable program consists of code + data, stored in memory
    - each variable occupies one or more bytes (e.g., an `int` typically occupies 4 bytes)
    - The address of the first byte is the address of the variable

Memory Layout of a C Program



A Practical Example




```c
#include<stdio.h>
#include<malloc.h>

void FUNCTION_1();
void FUNCTION_2();

char S1[]="FIRMCODES";  //initialized read-write area of DATA segment
int i;                  //uninitialized DATA segment
const int x=1;          //initialized read-only area of DATA segment

int main()
{
    static int TEMP=0;  //uninitialized DATA segment

    char *p=(char*)malloc(sizeof(char)); //Heap segment

    FUNCTION_1();       //FUNCTION_1 stack frame

    return 0;
}

void FUNCTION_1()
{
    int a;          //initialized in stack frame of FUNCTION_1

    FUNCTION_2();   //FUNCTION_2 stack frame
}

void FUNCTION_2()
{
    int b;          //initialized in stack frame of FUNCTION_2
}
```

[http://www.firmcodes.com/memory-layout-c-program-2/]

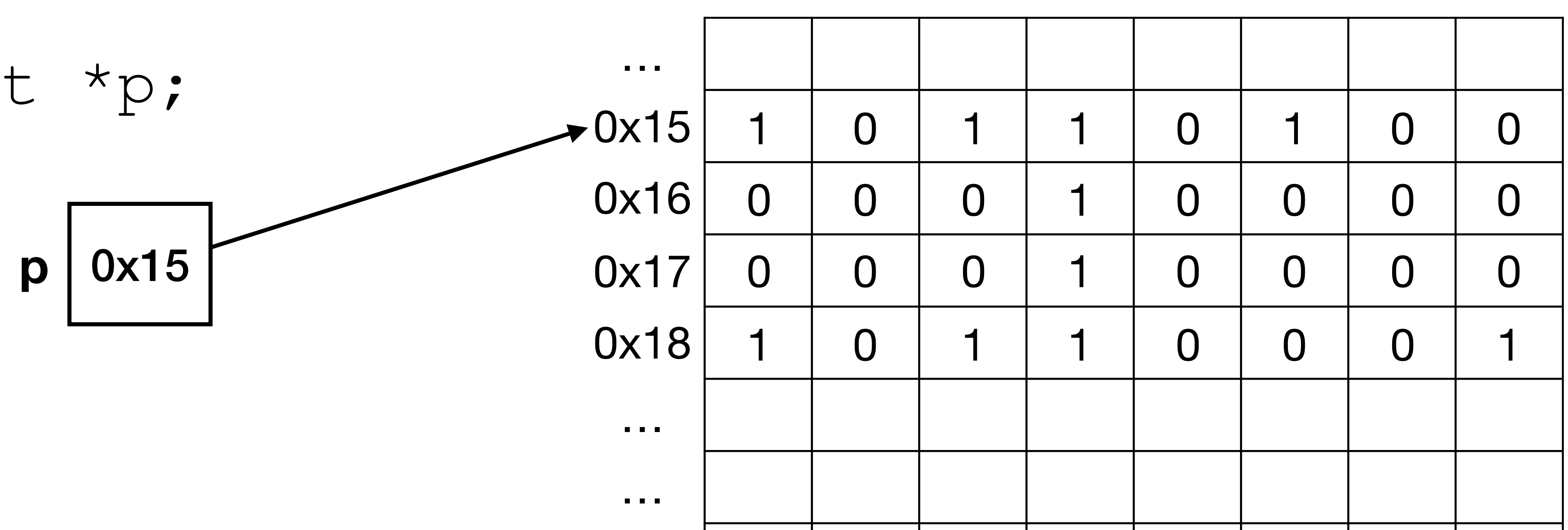
Pointer Variables

- Memory addresses are represented by (usually hexadecimal) numbers

- To differentiate them from regular integers, memory addresses are stored in *pointer variables,* denoted by the `*`

- For example, to declare a pointer `p` that points to an `int`:

    - `int *p;`



Examples of Pointer Declarations

```
int *p;
double *q;
char *r
```

The Address Operator

- The address operator `&`: gets the address of a given variable

```
int i, *p;
scanf("%d", &i);
…
…
p = &i;

scanf("%d", p);
```

The Address Operator

- The address operator `&`: gets the address of a given variable

```
int i, *p;
scanf("%d", &i);
…
…
p = &i;

scanf("%d", p);
```

scanf needs the address of `i` in order to read data into it

The Address Operator

- The address operator `&`**:** gets the address of a given variable

```
int i, *p;
scanf("%d", &i);
…
…
p = &i;

scanf("%d", p);
```

scanf needs the address of `i` in order to read data into it

this statement assigns the address of `i` to the pointer variable `p`. In other words, `p` now points to `i`

The Address Operator

- The address operator `&:` gets the address of a given variable

```
int i, *p;
scanf("%d", &i);
…
…
p = &i;

scanf("%d", p);
```

scanf needs the address of `i` in order to read data into it

this statement assigns the address of `i` to the pointer variable `p`. In other words, `p` now points to `i`

now we can use `p` (not `&p`) here, because `p` is itself an address

The Indirection Operator

- The indirection operator *: allows accessing the content stored in the object pointed to by the pointer

```
int i, j, *p, *q;

i = 9;

p = &i;

printf("%d", i);

printf("%d", *p);

j = *&i;

*q = 1;
```

The Indirection Operator

- The indirection operator *: allows accessing the content stored in the object pointed to by the pointer

```
int i, j, *p, *q;

i = 9;

p = &i;

printf("%d", i);

printf("%d", *p);

j = *&i;

*q = 1;
```

assign address of `i` to `p` so now `p` points to `i`

The Indirection Operator

- The indirection operator *: allows accessing the content stored in the object pointed to by the pointer

```
int i, j, *p, *q;

i = 9;

p = &i;

printf("%d", i);

printf("%d", *p);

j = *&i;

*q = 1;
```

assign address of `i` to `p` so now `p` points to `i`

`i` is an `int` so we can just print it directly

The Indirection Operator

- The indirection operator *: allows accessing the content stored in the object pointed to by the pointer

```
int i, j, *p, *q;

i = 9;

p = &i;

printf("%d", i);

printf("%d", *p);

j = *&i;

*q = 1;
```

assign address of `i` to `p` so now `p` points to `i`

`i` is an `int` so we can just print it directly

`p` is a pointer so we need to apply the indirection operator first (a.k.a de-reference the pointer)

The Indirection Operator

- The indirection operator *: allows accessing the content stored in the object pointed to by the pointer

```
int i, j, *p, *q;

i = 9;

p = &i;

printf(“%d”, i);

printf(“%d”, *p);

j = *&i;

*q = 1;
```

assign address of `i` to `p` so now `p` points to `i`

`i` is an `int` so we can just print it directly

`p` is a pointer so we need to apply the indirection operator first (a.k.a de-reference the pointer)

equivalent to `j = i;`

The Indirection Operator

- The indirection operator *: allows accessing the content stored in the object pointed to by the pointer

```
int i, j, *p, *q;

i = 9;

p = &i;

printf("%d", i);

printf("%d", *p);

j = *&i;

*q = 1;
```

assign address of `i` to `p` so now `p` points to `i`

`i` is an `int` so we can just print it directly

`p` is a pointer so we need to apply the indirection operator first (a.k.a de-reference the pointer)

equivalent to `j = i;`

WRONG!!! q is uninitialized so it doesn't contain any memory address that it can de-reference

Pointer Assignment

```
int i, *p, *q;
i = 10;
p = &i;
q = p;
```

Pointer Assignment

```
int i, *p, *q;
i = 10;
p = &i;
q = p;
```

This is a pointer assignment (I'm assigning one pointer to the other).. What do you think this assignment does?

Pointer Assignment

```
int i, *p, *q;
i = 10;
p = &i;
q = p;
```

This is a pointer assignment (I'm assigning one pointer to the other).. What do you think this assignment does?

- Let's think about what assignment in general does: it copies the value of an expression into the lvalue on the left-hand side

Pointer Assignment

```
int i, *p, *q;
i = 10;
p = &i;
q = p;
```

This is a pointer assignment (I'm assigning one pointer to the other).. What do you think this assignment does?

- Let's think about what assignment in general does: it copies the value of an expression into the lvalue on the left-hand side
- Thus, `q = p` copies the value of `p` into `q`.. but what is the value of `p`?

Pointer Assignment

```
int i, *p, *q;
i = 10;
p = &i;
q = p;
```

This is a pointer assignment (I'm assigning one pointer to the other).. What do you think this assignment does?

- Let's think about what assignment in general does: it copies the value of an expression into the lvalue on the left-hand side
- Thus, `q = p` copies the value of `p` into `q`.. but what is the value of `p`?
- The value of `p` is a memory address. In this case, it is the address of variable `i`

Pointer Assignment

```
int i, *p, *q;
i = 10;
p = &i;
q = p;
```

This is a pointer assignment (I'm assigning one pointer to the other).. What do you think this assignment does?

- Let's think about what assignment in general does: it copies the value of an expression into the lvalue on the left-hand side
- Thus, `q = p` copies the value of `p` into `q`.. but what is the value of `p`?
- The value of `p` is a memory address. In this case, it is the address of variable `i`
- Therefore, the assignment will copy the value of this memory address into `q`

Pointer Assignment

```
int i, *p, *q;
i = 10;
p = &i;
q = p;
```

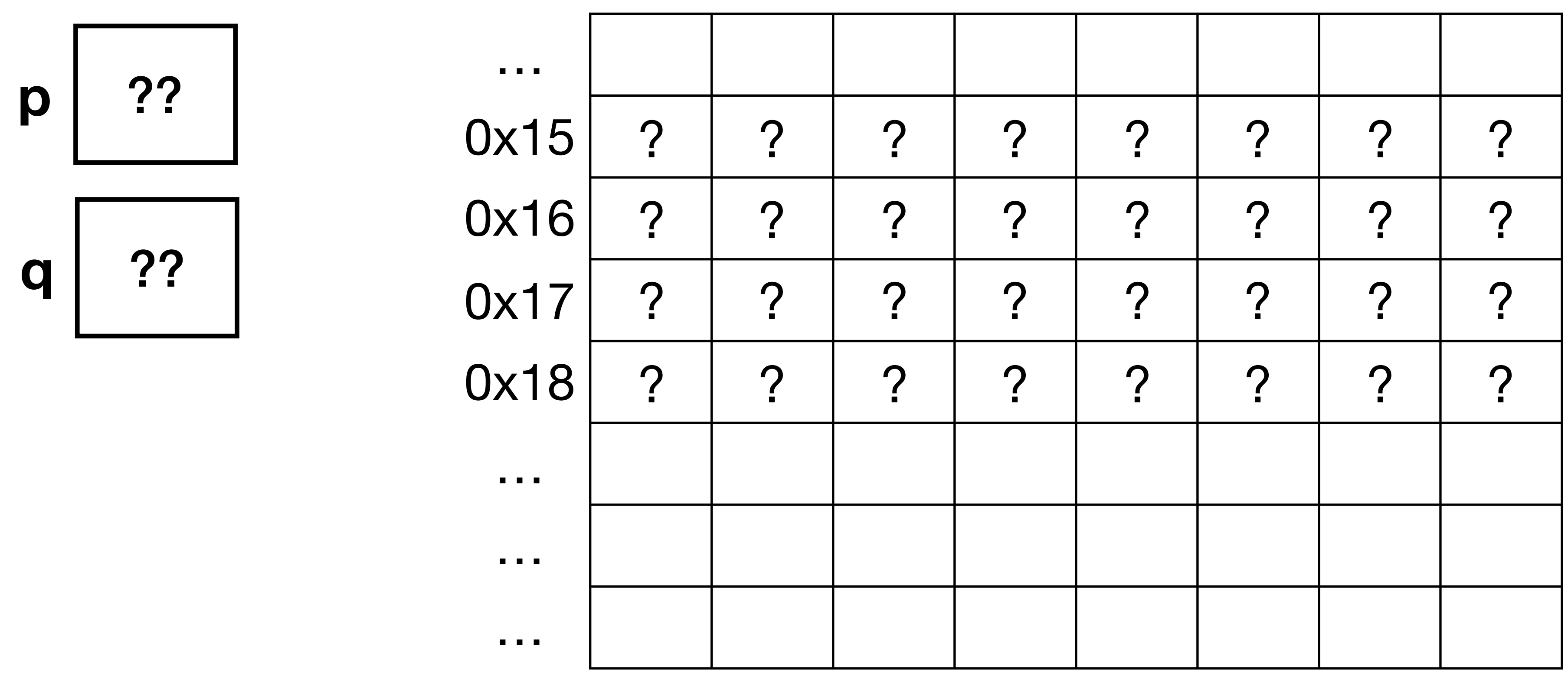
This is a pointer assignment (I'm assigning one pointer to the other).. What do you think this assignment does?

- Let's think about what assignment in general does: it copies the value of an expression into the lvalue on the left-hand side
- Thus, `q = p` copies the value of `p` into `q`.. but what is the value of `p`?
- The value of `p` is a memory address. In this case, it is the address of variable `i`
- Therefore, the assignment will copy the value of this memory address into `q`
- In other words, now `p` and `q` point to the same memory location. This means that `*p` and `*q` are exactly the same value

Let’s visualize this

Let’s visualize this

```
int i, *p, *q;
```



Let’s visualize this

```
int i, *p, *q;
i = 10;
```

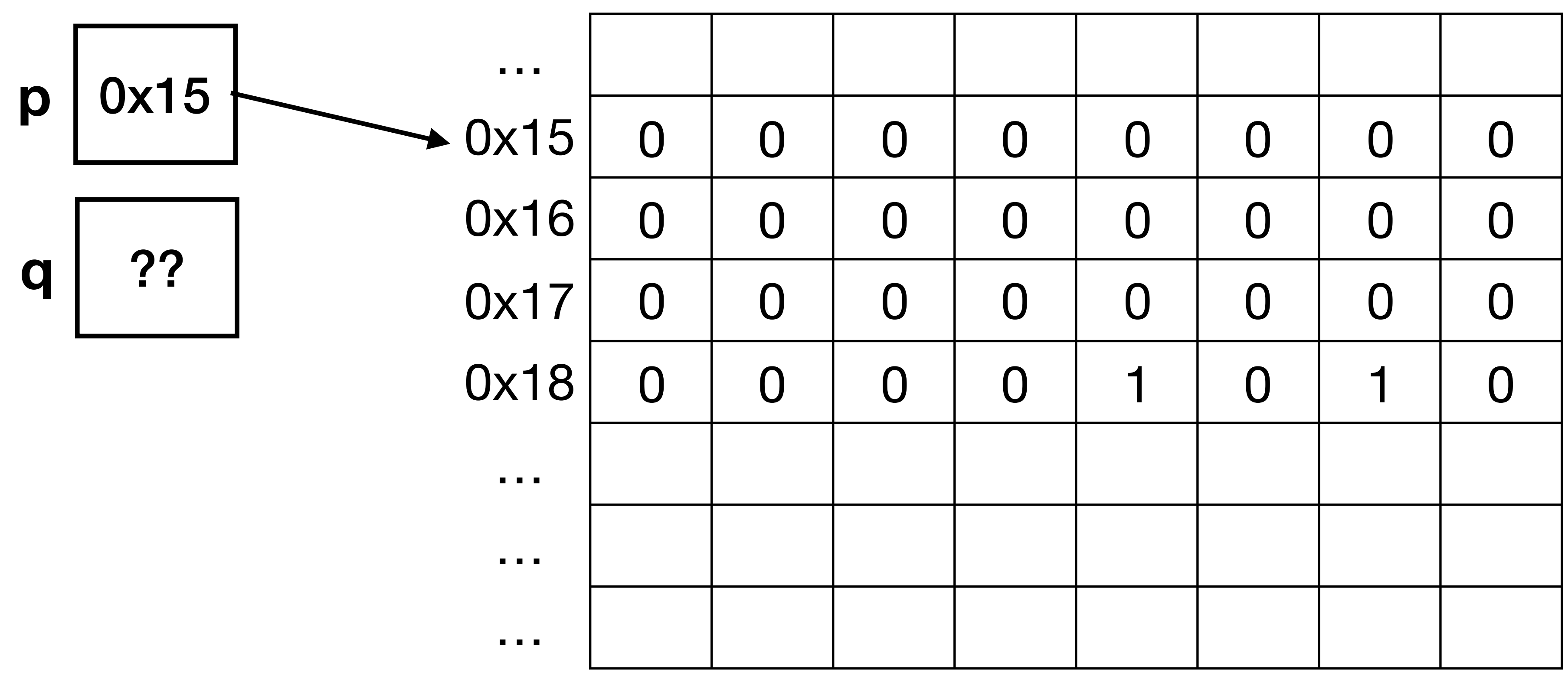
p ??

q ??

…								
0x15	0	0	0	0	0	0	0	0
0x16	0	0	0	0	0	0	0	0
0x17	0	0	0	0	0	0	0	0
0x18	0	0	0	0	1	0	1	0
…								
…								
…								

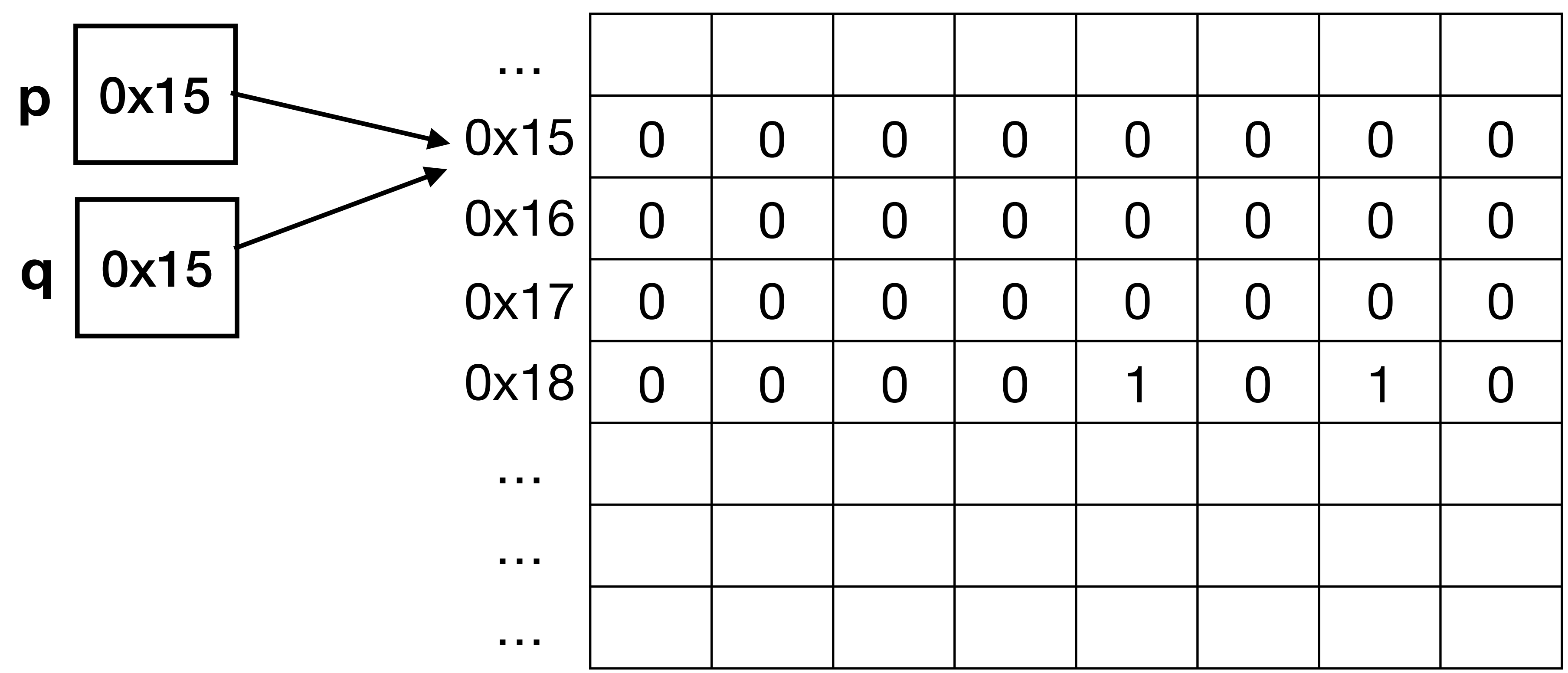
Let’s visualize this

```
int i, *p, *q;
i = 10;
p = &i;
```



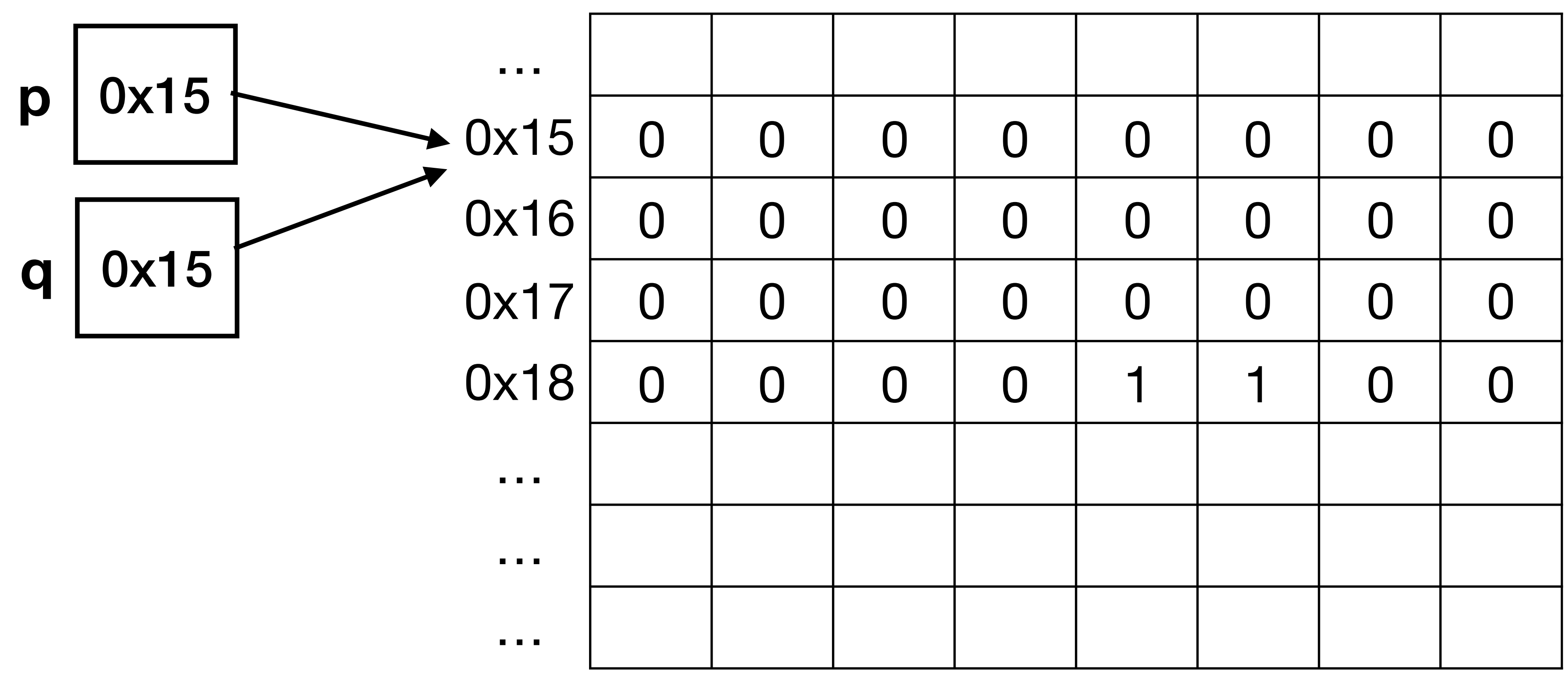
Let’s visualize this

```
int i, *p, *q;
i = 10;
p = &i;
q = p;
```



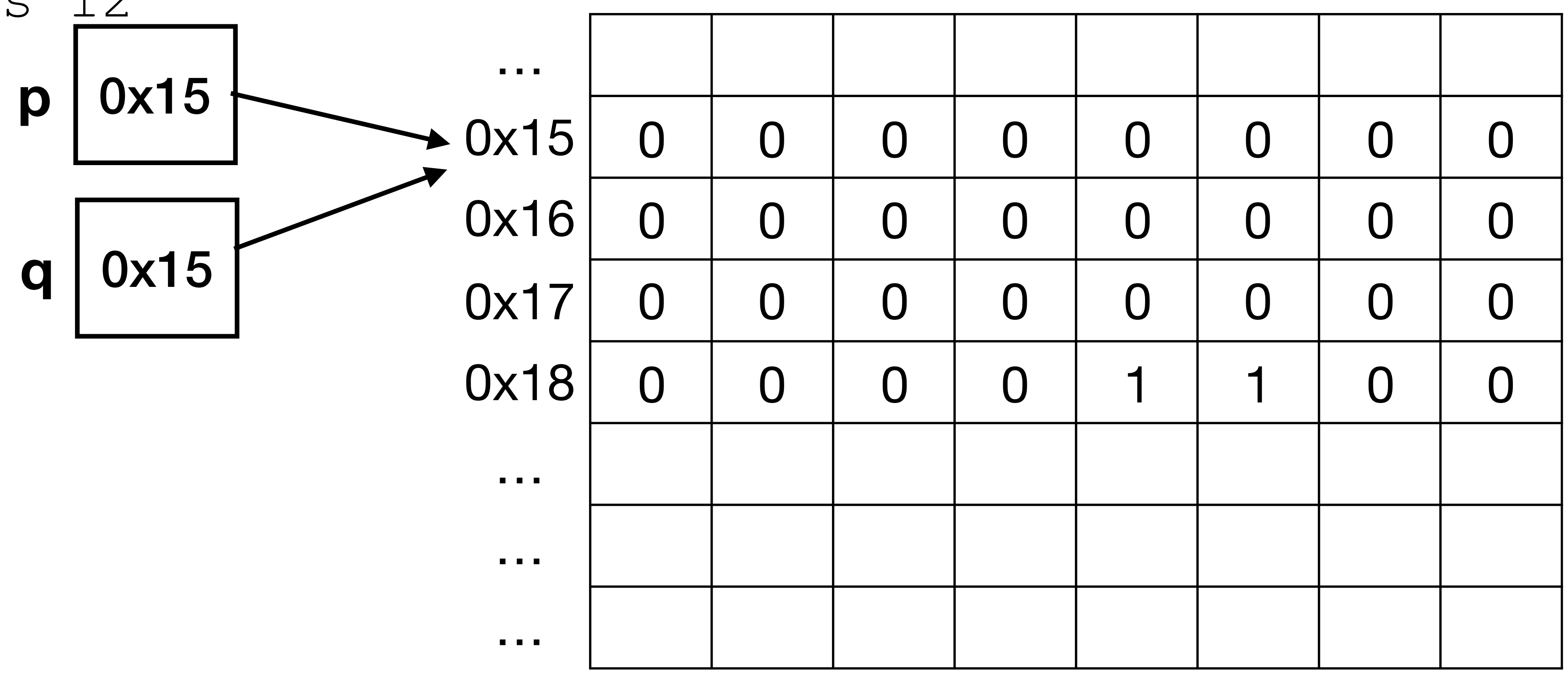
Let’s visualize this

```
int i, *p, *q;
i = 10;
p = &i;
q = p;
*q = 12;
```



Let's visualize this

```
int i, *p, *q;
i = 10;
p = &i;
q = p;
*q = 12;
//next line prints 12
printf("%d", *p);
```



More on Pointer Assignment

- If you want to copy the content of the object pointed to by a pointer into the object pointed to by another pointer:

```
int i = 10, j = 20, *p, *q;
p = &i;
q = &j;
*p = *q; /*same as saying i = j. Essentially, now the value of the
              object at memory address p is 20 */
```

Pointers as Arguments

```
void decompose(double x, long int_part, double frac_part) {
   int_part = (long) x;
   frac_part = x - int_part;
}
```

Pointers as Arguments

- Recall: arguments are passed by value to parameters

```
void decompose(double x, long int_part, double frac_part) {
   int_part = (long) x;
   frac_part = x - int_part;
}
```

Pointers as Arguments

- Recall: arguments are passed by value to parameters

```
void decompose(double x, long int_part, double frac_part) {
   int_part = (long) x;
   frac_part = x - int_part;
}
```

Pointers as Arguments

- Recall: arguments are passed by value to parameters

```
void decompose(double x, long int_part, double frac_part) {
   int_part = (long) x;
   frac_part = x - int_part;
}
```

Pointers as Arguments

- Recall: arguments are passed by value to parameters

```
void decompose(double x, long int_part, double frac_part) {
   int_part = (long) x;
   frac_part = x - int_part;
}
```

- So how can we use these two calculated parts **outside** of the function decompose?

Pointers as Arguments

- Recall: arguments are passed by value to parameters

```
void decompose(double x, long int_part, double frac_part) {
   int_part = (long) x;
   frac_part = x - int_part;
}
```

- So how can we use these two calculated parts **outside** of the function decompose?
    - we can have a return value but that's limited to one so we have to choose which of these values to return

Pointers as Arguments

- Recall: arguments are passed by value to parameters

```
void decompose(double x, long int_part, double frac_part) {
   int_part = (long) x;
   frac_part = x - int_part;
}
```

- So how can we use these two calculated parts **outside** of the function decompose?
  - we can have a return value but that's limited to one so we have to choose which of these values to return
  - we know that functions cannot return arrays

Pointers as Arguments

- Recall: arguments are passed by value to parameters

```
void decompose(double x, long int_part, double frac_part) {
   int_part = (long) x;
   frac_part = x - int_part;
}
```

- So how can we use these two calculated parts **outside** of the function decompose?
    - we can have a return value but that's limited to one so we have to choose which of these values to return
    - we know that functions cannot return arrays
    - we can have these parameters as two arrays since the value of array members can be changed in the function, but that's just complicated and we will need to add the size of the array to the parameter list

Let’s Use Pointers to Solve this Problem!

```
void decompose(double x, long *int_part, double *frac_part) {
   *int_part = (long) x;
   *frac_part = x - *int_part;
}
```

```
int main(){
  double x = 10.34
  long int_prt;
  double frac_prt;
  decompose(x, &int_prt, &frac_prt);
  //int_prt and frac_prt now have new values
}
```

Let's Use Pointers to Solve this Problem!

```
void decompose(double x, long *int_part, double *frac_part) {
   *int_part = (long) x;
   *frac_part = x - *int_part;
}
```

```
int main(){
  double x = 10.34
  long int_prt;
  double frac_prt;
  decompose(x, &int_prt, &frac_prt);
  //int_prt and frac_prt now have new values
}
```

x (address = 0x30)

10.34

Let's Use Pointers to Solve this Problem!

```
void decompose(double x, long *int_part, double *frac_part) {
   *int_part = (long) x;
   *frac_part = x - *int_part;
}
```

```
int main(){
 double x = 10.34
 long int_prt;
 double frac_prt;
 decompose(x, &int_prt, &frac_prt);
 //int_prt and frac_prt now have new values
}
```

int_prt (address = 0x50)

??

x (address = 0x30)

10.34

Let's Use Pointers to Solve this Problem!

```
void decompose(double x, long *int_part, double *frac_part) {
   *int_part = (long) x;
   *frac_part = x - *int_part;
}
```

```
int main(){
  double x = 10.34
  long int_prt;
  double frac_prt;
  decompose(x, &int_prt, &frac_prt);
  //int_prt and frac_prt now have new values
}
```

int_prt (address = 0x50)

??

frac_prt (address = 0x58)

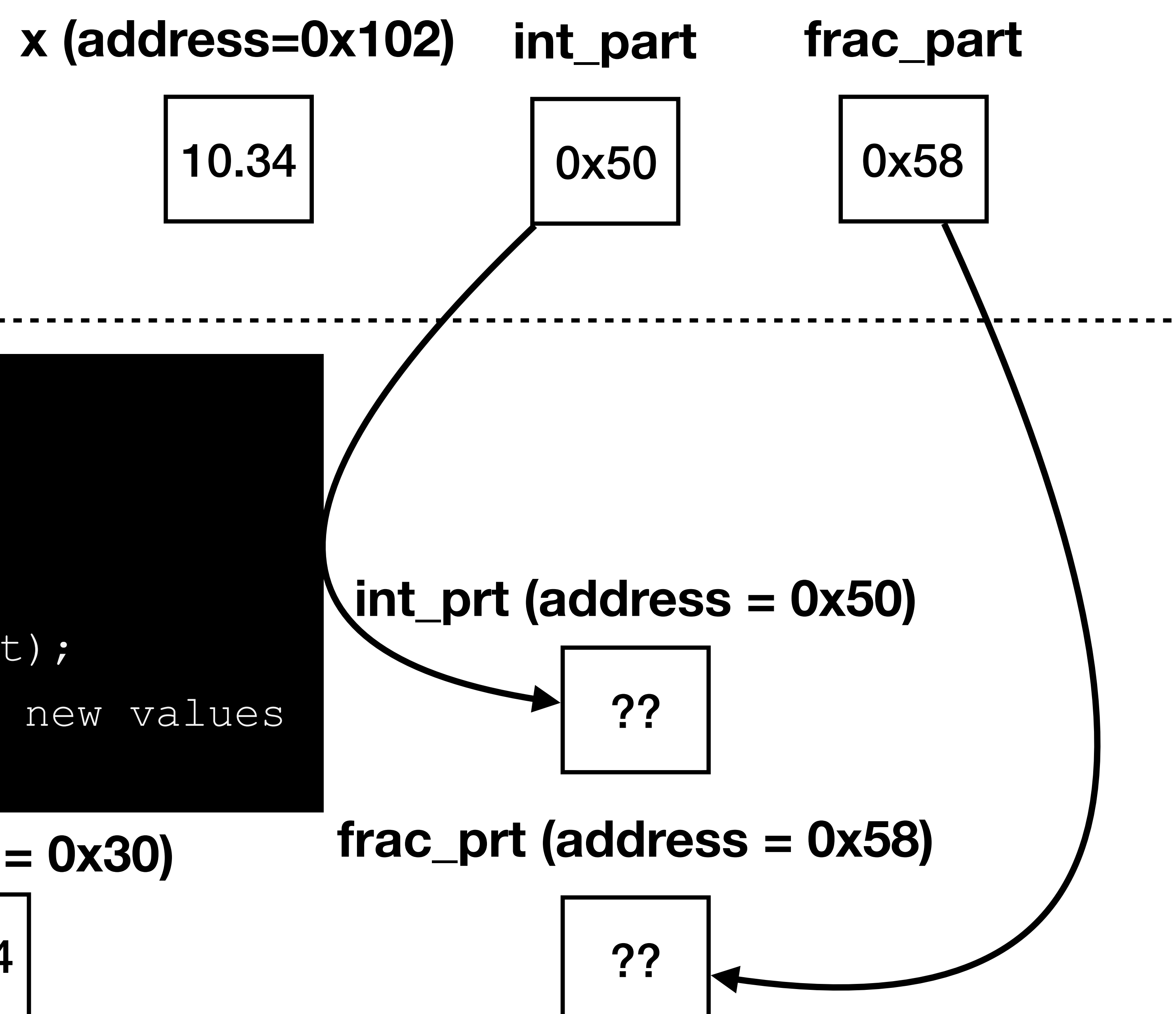
??

x (address = 0x30)

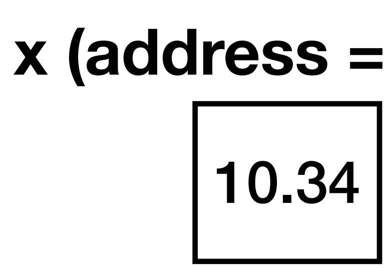
10.34

Let’s Use Pointers to Solve this Problem!

```
void decompose(double x, long *int_part, double *frac_part) {
   *int_part = (long) x;
   *frac_part = x - *int_part;
}
```

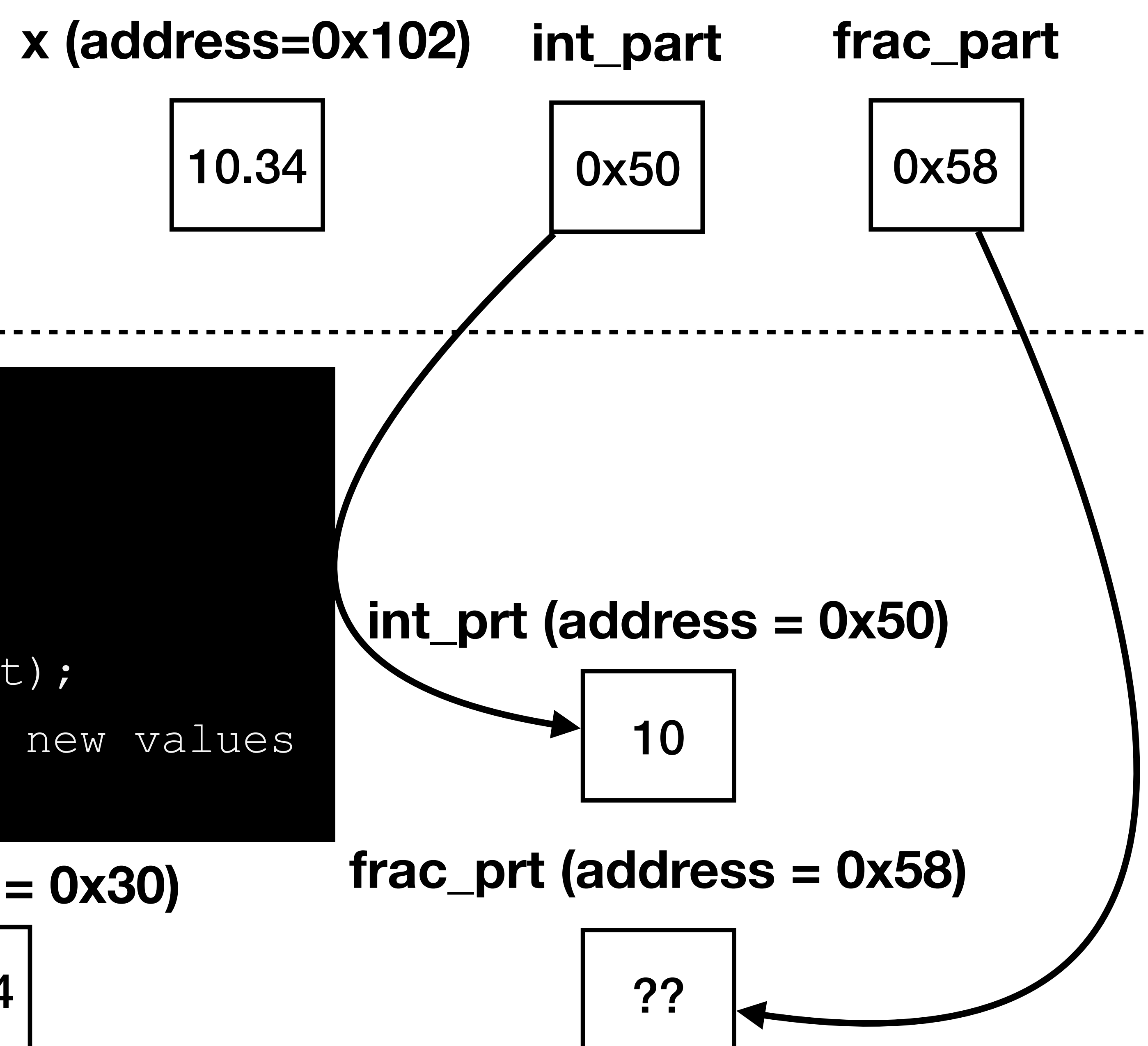



```
int main(){
  double x = 10.34
  long int_prt;
  double frac_prt;
▶ decompose(x, &int_prt, &frac_prt);
  //int_prt and frac_prt now have new values
}
```

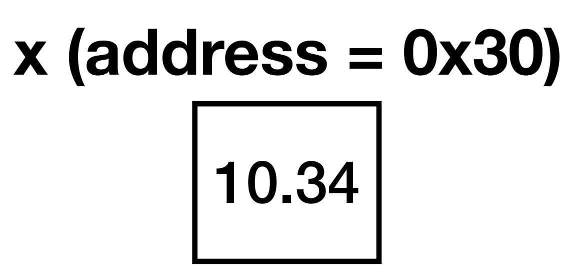


Let's Use Pointers to Solve this Problem!

```
void decompose(double x, long *int_part, double *frac_part) {
   *int_part = (long) x;
   *frac_part = x - *int_part;
}
```




```
int main(){
  double x = 10.34
  long int_prt;
  double frac_prt;
  decompose(x, &int_prt, &frac_prt);
  //int_prt and frac_prt now have new values
}
```



Let’s Use Pointers to Solve this Problem!

```
void decompose(double x, long *int_part, double *frac_part) {
   *int_part = (long) x;
   *frac_part = x - *int_part;
}
```

x (address=0x102)	int_part	frac_part
10.34	0x50	0x58

```
int main(){
  double x = 10.34
  long int_prt;
  double frac_prt;
  decompose(x, &int_prt, &frac_prt);
  //int_prt and frac_prt now have new values
}
```

int_prt (address = 0x50)

10

frac_prt (address = 0x58)

0.34

x (address = 0x30)

10.34

Let’s Use Pointers to Solve this Problem!

```
void decompose(double x, long *int_part, double *frac_part) {
    *int_part = (long) x;
    *frac_part = x - *int_part;
}
```

```
int main(){
  double x = 10.34
  long int_prt;
  double frac_prt;
  decompose(x, &int_prt, &frac_prt);
  //int_prt and frac_prt now have new values
}
```

int_prt (address = 0x50)

10

x (address = 0x30)

10.34

frac_prt (address = 0x58)

0.34

Example: Swapping Numbers

```
void swap(int a, int b){
  int temp;
  temp = a;
  a = b;
  b = temp;
  return;
}

int main(){
  int a = 10, b = 8;
  if (a > b)
    swap(&a, &b);
    return 0;
}
```

X

will not actually swap the numbers

```
void swap(int *a, int *b){
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
  return;
}

int main(){
  int a = 10, b = 8;
  if (a > b)
    swap(&a, &b);
}
```

Protecting Objects

- Use the `const` keyword to document that a function will not change an object whose address is passed to the function

```
void f(const int *p){
   *p = 0; /*** WRONG ***/
}
```

Pointer as Return Values

- A function can also return a pointer

```
int* swap_if_larger(int *a, int *b){
  int temp;
  if(*a > *b){
    temp = *a;
    *a = *b;
    *b = temp;
  }
  return b; //returns the larger number
}
```

This example is ok, because b is actually a memory address that was passed to the function by its caller. Thus, when execution returns to the caller of this function, this memory address still exists

[https://www.iconfinder.com/icons/118940/dialog_warning_icon#size=256]

Pointer as Return Values

- A function can also return a pointer

```
int* swap_if_larger(int *a, int *b){
  int temp;
  if(*a > *b){
    temp = *a;
    *a = *b;
    *b = temp;
  }
  return b; //returns the larger number
}
```

This example is ok, because b is actually a memory address that was passed to the function by its caller. Thus, when execution returns to the caller of this function, this memory address still exists

do **NOT** return a pointer to an **automatic local variable**!

you **CAN** return a pointer to an **external/global variable** or to a **STATIC local variable**

[https://www.iconfinder.com/icons/118940/dialog_warning_icon#size=256]

Debugging Programs with gdb

I got an abort 6 trap error message

I got a segmentation fault

Stack smashing detected …
Aborted (core dumped)

My program is outputting funny values

[http://moziru.com/explore/Overview%20clipart%20confused%20face/]

Most of the time, the answer is: you are accessing memory you should not be accessing

GDB

```
GDB(1) GNU Development Tools GDB(1)

NAME
       gdb - The GNU Debugger

SYNOPSIS
       gdb [-help] [-nh] [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps]
           [-tty=dev] [-s symfile] [-e prog] [-se prog] [-c core] [-p procID]
           [-x cmds] [-d dir] [prog|prog procID|prog core]

DESCRIPTION
       The purpose of a debugger such as GDB is to allow you to see
       what is going on "inside" another program while it executes -- or
       what another program was doing at the moment it crashed.
```

Check the gdb manual using `man gdb`

Debugging a Given Program

```
#include <stdio.h>
#include <limits.h>

void find_largest(int array[], int size){

    int start=0,end=size-1;
    int largest = INT_MIN;

    while(end > start){
        if(array[end] > largest)
            largest = array[end];

        if(array[start] > largest)
            largest = array[start];

        end +=1;
        start += 1;
    }

    printf("largest=%d\n", largest);
}


int main(){
    int array[] = {1,10,4,40,2,3,9,2,9,10};

    find_largest(array, 10);

}
```

Debugging a Given Program

```c
#include <stdio.h>
#include <limits.h>

void find_largest(int array[], int size){

    int start=0,end=size-1;
    int largest = INT_MIN;

    while(end > start){
        if(array[end] > largest)
            largest = array[end];

        if(array[start] > largest)
            largest = array[start];

        end +=1;
        start += 1;
    }

    printf("largest=%d\n", largest);
}


int main(){
    int array[] = {1,10,4,40,2,3,9,2,9,10};

    find_largest(array, 10);

}
```

```
Lecture10>./search
Segmentation fault (core dumped)
```

Debugging a Given Program

```c
#include <stdio.h>
#include <limits.h>

void find_largest(int array[], int size){

    int start=0,end=size-1;
    int largest = INT_MIN;

    while(end > start){
        if(array[end] > largest)
            largest = array[end];

        if(array[start] > largest)
            largest = array[start];

        end +=1;
        start += 1;
    }

    printf("largest=%d\n", largest);
}


int main(){
    int array[] = {1,10,4,40,2,3,9,2,9,10};

    find_largest(array, 10);

}
```

```
Lecture10>./search
Segmentation fault (core dumped)
```

Let's use gdb to debug this:

```
gdb ./test
```

Example

```
nadi@ug12:~/CMPUT201/cmput201-lecture-demos/Lecture10>gdb ./search
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./test...(no debugging symbols found)...done.
(gdb)
```

??

Let’s Check the Compiler

```
GCC(1) GNU GCC(1)

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-Wpedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] [@file] infile...

       Only the most useful options are listed here; see below for the remainder.
       ...

       Debugging Options
       ...
       -ggdb
```

Let's Change the Makefile to include the -ggdb option

```
search: search.c
	gcc -Wall -ggdb -std=c99 -o search search.c

clean:
	rm -f search
```

Let's try this again after re-compiling

```
nadi@ug12:~/CMPUT201/cmput201-lecture-demos/Lecture10>gdb ./search
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
..
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./search...done.
(gdb) run
Starting program: /cshome/nadi/CMPUT201/cmput201-lecture-demos/Lecture10/search

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005d2 in find_largest (array=0x7fffffffe660, size=10)
    at search.c:10
10        if(array[end] > largest)
(gdb)
```

Some of most frequently used gdb commands

```
break [file:]functiop
    Set a breakpoint at function (in file).

run [arglist]
    Start your program (with arglist, if specified).

bt  Backtrace: display the program stack.

print expr: Display the value of an expression.

c   Continue running your program (after stopping, e.g. at a breakpoint).

next
    Execute next program line (after stopping); step over any function calls in the line.

edit [file:]function
    look at the program line where it is presently stopped.

list [file:]function
    type the text of the program in the vicinity of where it is presently stopped.

step
    Execute next program line (after stopping); step into any function calls in the line.

help [name]
    Show information about GDB command name, or general information about using GDB.

quit
    Exit from GDB.
```

Fixing the Problem

```c
#include <stdio.h>
#include <limits.h>

void find_largest(int array[], int size){

    int start=0,end=size-1;
    int largest = INT_MIN;

    while(end > start){
        if(array[end] > largest)
            largest = array[end];

        if(array[start] > largest)
            largest = array[start];

        end +=1;
        start += 1;
    }

    printf("largest=%d\n", largest);
}


int main(){
    int array[] = {1,10,4,40,2,3,9,2,9,10};

    find_largest(array, 10);

}
```

A debugger only helps you locate the problem.. it doesn't tell you how to fix it

Fixing the Problem

```c
#include <stdio.h>
#include <limits.h>

void find_largest(int array[], int size){

    int start=0,end=size-1;
    int largest = INT_MIN;

    while(end > start){
        if(array[end] > largest)
            largest = array[end];

        if(array[start] > largest)
            largest = array[start];

        end +=1;
        start += 1;
    }

    printf("largest=%d\n", largest);
}


int main(){
    int array[] = {1,10,4,40,2,3,9,2,9,10};

    find_largest(array, 10);

}
```

A debugger only helps you locate the problem.. it doesn't tell you how to fix it