Memory Allocation

Lesson 1.1

Learning Objectives

LO 1.1.1 **Write** valid C code that utilizes pointers

LO 1.1.2 **Visualize** conceptually the configuration of user-defined variables at any point of the code implementation

LO 1.1.3 **Apply** multiple levels of indirections using pointers correctly

LO 1.1.4 **Allocate** memory dynamically

Stack Memory

- The **stack** memory is the place where all local variables are stored
  - —a local variable is declared in some scope, example:

```
int x; // creates the variable x on the stack
```

- As soon as the scope ends, all local variables declared in that scope end
  - —the variable name and its space are gone
  - —this happens **implicitly** – the user has no control over it

Heap Memory

- The **heap** memory is an area of memory that the user handles explicitly
  - user **allocates** and **deallocates** the memory through system calls
  - if a user forgets to release memory, it doesn't get destroyed, it just uses up extra memory

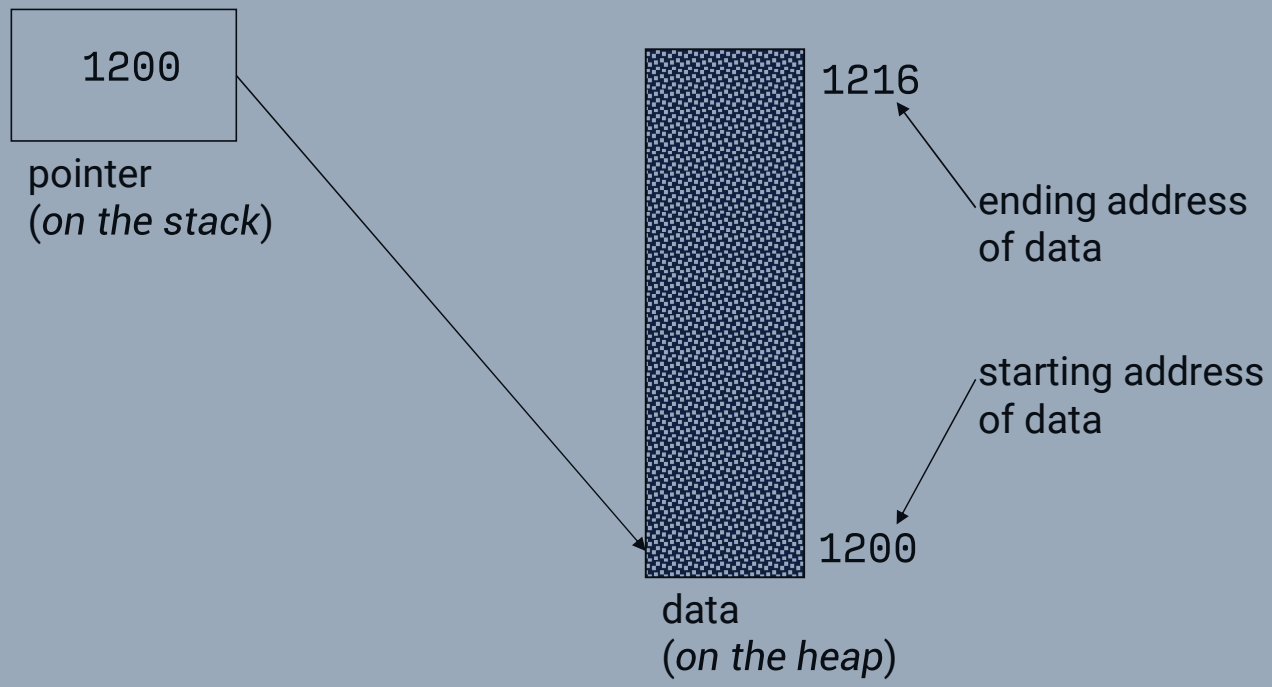
- A user maintains a handle on memory allocated in the heap with a pointer

Pointers

- A **pointer** is simply a local variable that refers to a memory location on the heap

- Accessing the pointer, actually **references** the memory on the heap

Basic Idea


1200
pointer
(*on the stack*)
1216
ending address
of data
starting address
of data
1200
data
(*on the heap*)

Pointer Declaration

- Declaring a pointer is easy
  - declared like regular variable except that an asterisk (*) is placed in front of the variable, example:

```
int *x; //using this pointer now would be
        very dangerous because x points
        to some random piece of data
```

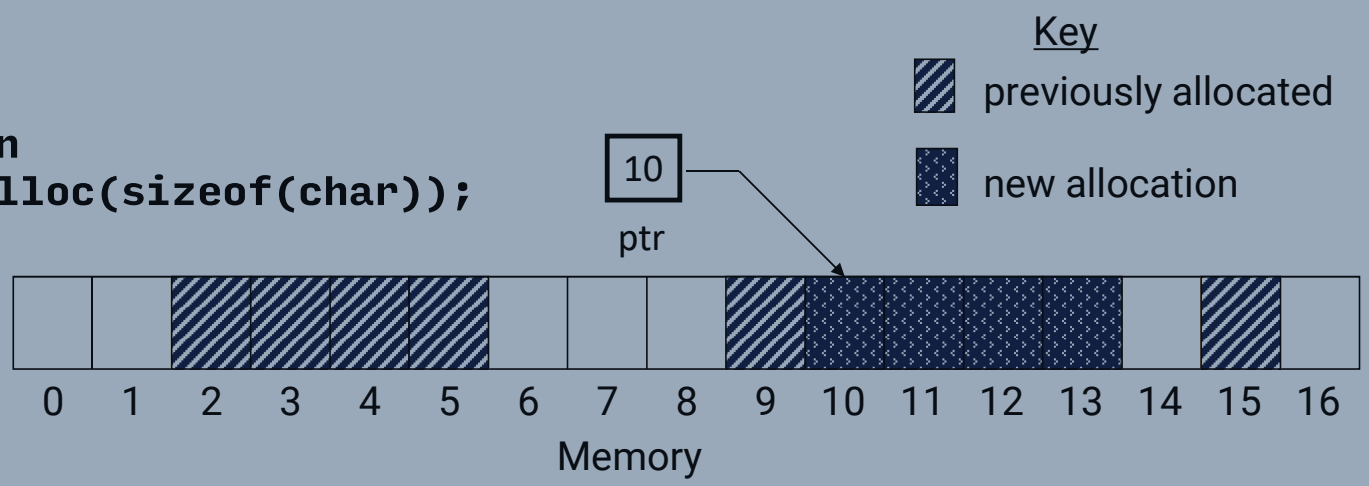
Pointer Declaration

- Declaring a variable does not allocate space on the heap for it
  - simply creates a **local variable** (on the stack memory) that will point to an address
  - use **`malloc()`** to actually allocate memory on the heap

Dynamic Memory Allocation

- Prototype: **`(void*) malloc(size_t size);`**
  - —function searches heap for size contiguous free bytes
  - —function returns the address of the first byte
  - —programmers responsibility to not lose the pointer and not write into area past the last byte allocated
- Example:

```
char *ptr;
// new allocation
ptr = (char*) malloc(sizeof(char));
```



Pointer Usage

- To access a piece of data through a pointer, place an asterisk (*) before the pointer, e.g.:

```
char *ptr = (char*)malloc(sizeof(char));
*ptr = ‘a’;
if(*ptr == ‘a’) { … }
```

- Using the pointer without the asterisk actually accesses the pointer value
  - — not the data the pointer is referencing, this is a very common mistake to make when trying to access data

Common Mistakes in Using Pointers

- Using a pointer before allocating heap space

```
int *ptr;
*ptr = 5;
```

- Changing the pointer, not the value it references

```
int *ptr = malloc(sizeof(int));
ptr = 10; // sets value on stack to 10, not
          value on the heap
```

Learning To Use Pointers

- **DRAW PICTURES**
  - when first using pointers it is much easier to draw pictures to learn what is happening
  - remember that an asterisk (*) follows the pointer
  - no asterisk (*) refers to the actual pointer variable on the stack

Strengthening the Learning Objectives

LO 1.1.1 Write valid C code that utilizes pointers

LO 1.1.4 Allocate memory dynamically

Example:

1. Create an integer pointer **`p`** that points to any existing integer variable.
2. Change the value of the integer pointed by **`p`**.
3. Allocate memory for a floating-point number and let a variable, **`pi`**, reference it. Assign the value **`3.14`**, thereafter.
4. Redirect **`p`** to point to **`pi`** and create another pointer, **`p_ptr`**, that can point to **`p`**. Display the value of **`pi`** through **`p_ptr`**.

LO 1.1.2 Visualize conceptually the configuration of user-defined variables at any point of the code implementation

Example:

Show the configuration (sketch) of all the variables and the output of the code snippet on the right:

```c
#include <stdio.h>
int main(void)
{
    //Local Declarations
    int a;
    int* p;
    int** q;
    //Statements
    a = 14;
    p = &a;
    q = &p;

    printf("%d\n", a);
    printf("%d\n", *p);
    printf("%d\n", **q);
    printf("%p\n", p);
    printf("%p\n", q);
    return 0;
}//main
```

LO 1.1.3 Apply multiple levels of indirections using pointers correctly

Example:

Write a program that creates the configuration shown on the right and then reads (`scanf`) an integer into variable `a` and prints it using each pointer in turn, i.e., the program must read an integer into variable `a` and print it using `p`, `q`, `r`, `s`, `t`, `u`, and `v`.