CS 240 Programming in C

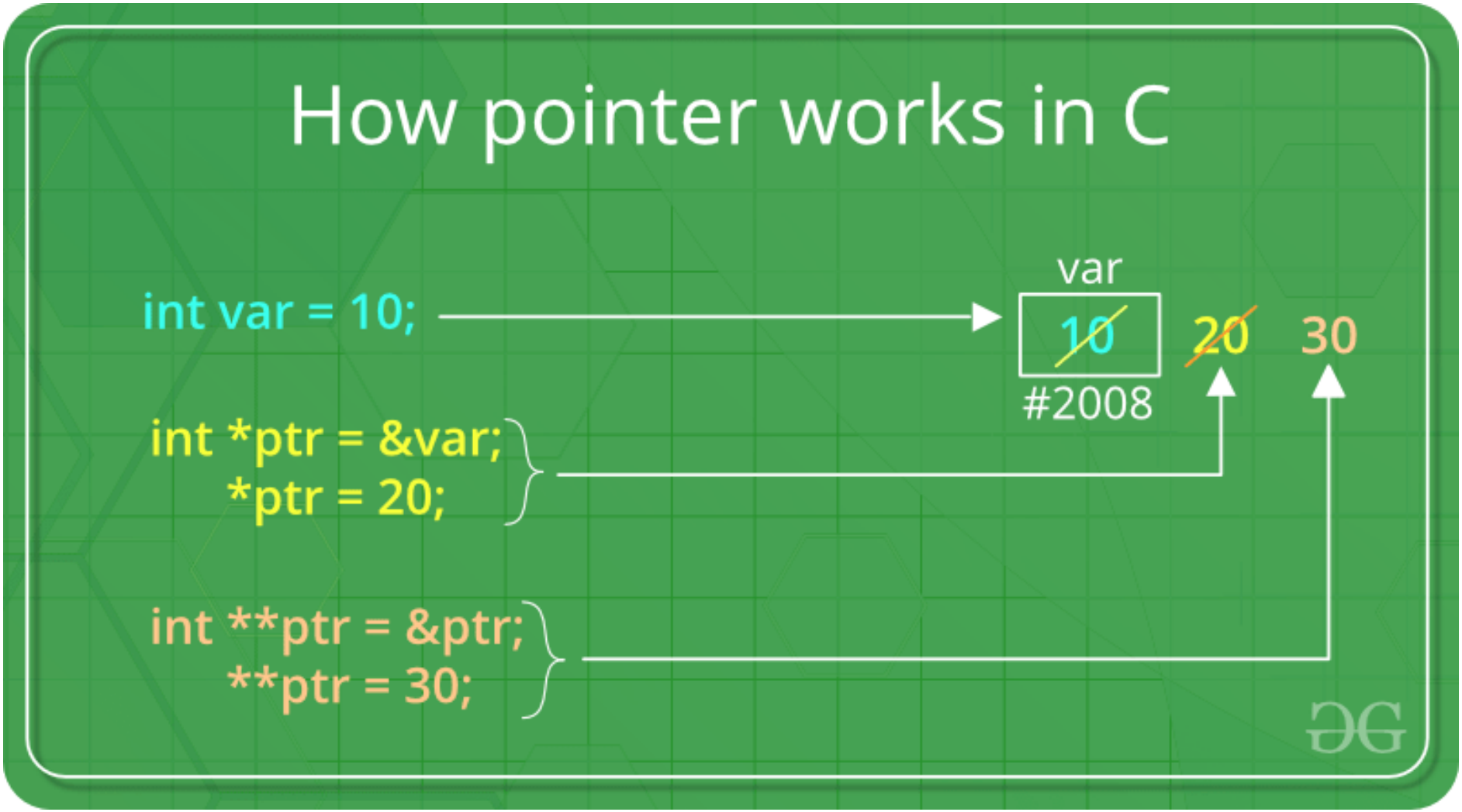
Pointers and Memory Allocation

September 27, 2022

& operator

- A pointer is a variable that contains the address of a variable.
- The unary operator & gives the address of an object.
- The & operator only applies to objects in memory: variables and array elements.
- It cannot be applied to expressions, constants, or register variables.

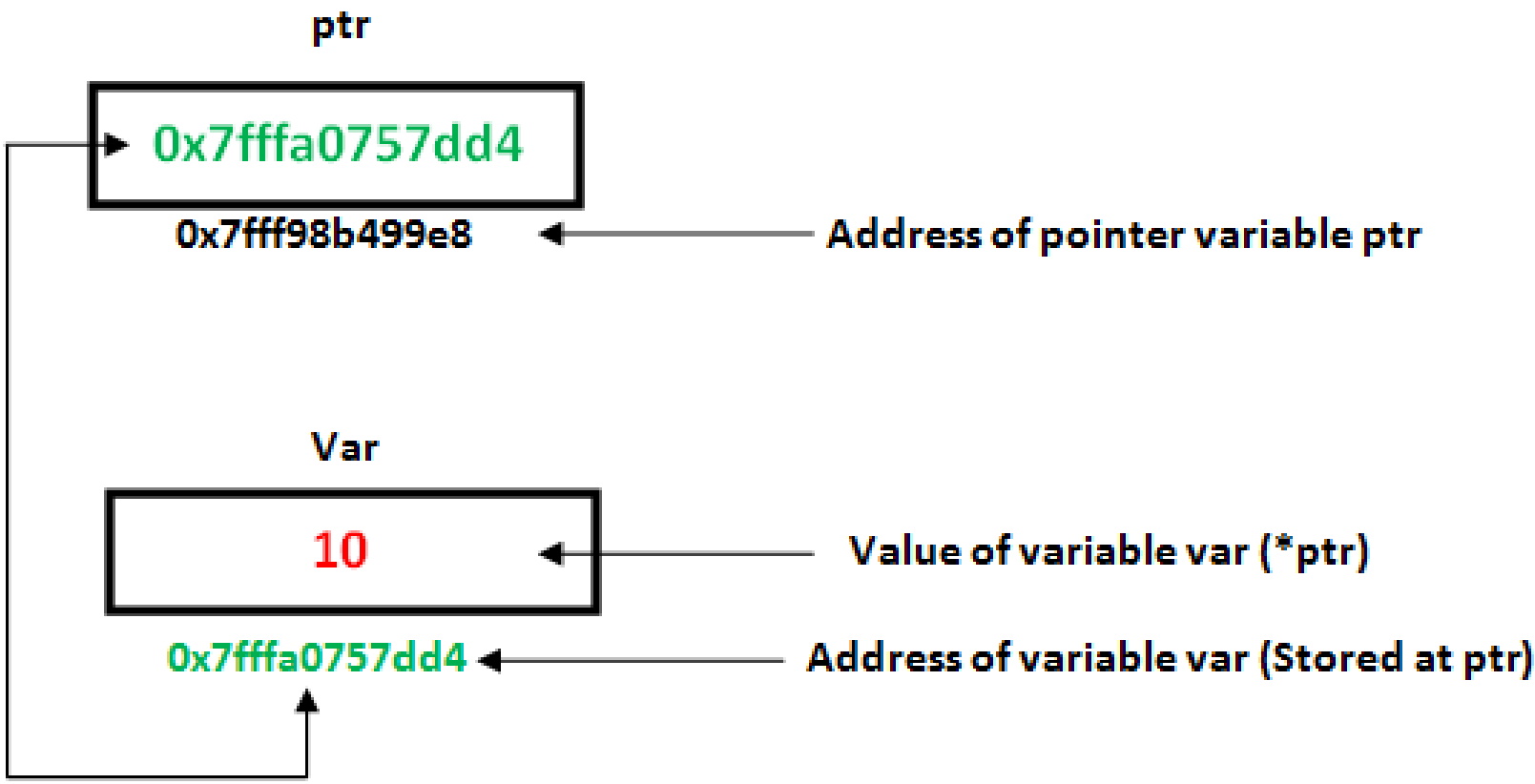
Pointers



1

[1] https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/

Pointers


ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (Stored at ptr)

* operator

- The unary operator * is the indirection or dereferencing operator;
- when applied to a pointer, it accesses the object the pointer points to.
- The declaration of a pointer variable is :

```
            [datatype] *[variable name]
for example:   int *ip;
```

means ip is pointer variable which reference an integer variable. i.e. *ip in an int, and ip is an pointer which stores an address value.

Initialization of a pointer

- There is no legal default value to a pointer variable. You have to initiablize it before using it.
- C guarantees that zero is never a valid address for data, so a pointer of value of zero can be used to signal an abnormal event.
- The symbolic constant NULL is often used in place of zero which is defined in <stdio. h>.
- A pointer has to be initialized to the address of an existing variable before any meaningful using. For example:

```
int i, *ip;
ip = &i;  //     or int i, *ip = &i;
*ip = 3;
```

- This is illegal

```
int *ip;
*ip = 3;
```

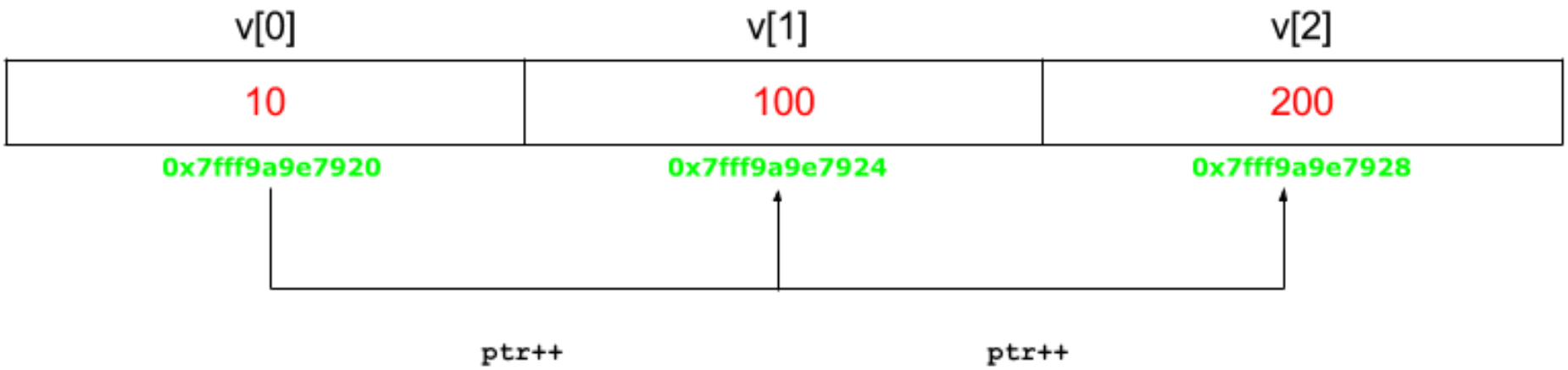
* operator

- The *ip in above case is just an integer variable, so it can be put into the expression where integer can be put in. For example:

```
*ip = * ip + 10;
*ip += 1;
*ip << 2;
*ip < 2;
++*ip;
(*ip)++; // means (*p) = (*p) + 1
*ip++;   // means *(ip = ip + 1)
      because unary operators like *
      and ++ associate right to left.
```

these are all legal expressions.

Pointers


v[0]
v[1]
v[2]
10
100
200
0x7fff9a9e7920
0x7fff9a9e7924
0x7fff9a9e7928
ptr++
ptr++

Pointer as arguments

- Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function.

Pointer and Arrays

- In C, there is a strong relationship between pointers and arrays.
- In fact array variable is just one type of pointer. It can be directly assigned to a pointer variable. For example:

```
int a[10] = {-1, -2}, *p = a;
printf("%d\n", *p);
```

- Besides a is just storing the address of the first element of a.

```
int a[10] = {-1}, *p = a;
printf("%d\n", a == &a[0]);
// what will be print out ?
```

- And p can also be applied array subscripting like:

```
printf("%d\n", p[1]);   // or
printf("%d\n", *(p+1));
```

Pointer and Arrays

- In evaluating a[i], C actually converts it to *(a+i) immediately; the two forms are equivalent.
- &a[i] and a+i are also identical

Pointer and Arrays – One difference

- There is one difference between an array name and a pointer that must be kept in mind.
- A pointer is a variable, so p=a and p++ are legal. But an array name is not a variable; constructions like a=p and a++ are illegal.
- Array name is equivalent to a symbolic constant address value, and it has to be a stack address.
- A pointer can reference to a heap address. We will see how later.

Pointer and Arrays

- As formal parameters in a function definition, char s[] and char *s are equivalent.
- It is preferred of the latter because it says more explicitly that the parameter is a pointer. That's why you see a lot "char *s" in library function headers.
- If one is sure that the elements exist, it is also possible to index backwards in an array; p[-1], p[-2], and so on are syntactically legal,
- But we can not refer to the elements that immediately precede p[0].
- Of course, it is illegal to refer to objects that are not within the array bounds.

Character Pointers and Functions

- String constant.

```
char amessage[] = "now is the time"; /* an array */
char *pmessage = "now is the time"; /* a pointer */
```

  - amessage is an array. Its individual characters within the array may be changed but amessage will always refer to the same storage.
  - pmessage is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere.
  - All in all amessange is left value, while pmessage is a right value.
- All in all amessange is left value, while pmessage is a right value.

Pointer Arrays; Pointers to Pointers

- ```
  char *lineptr[3];
  lineptr[0] = "hello";
  ```

  lineptr is an array of 3 elements, each element of which is a pointer to a char .
  ```

Two-dimensional Arrays

- Declaration and initialization.

```
int arr[2][6] = {
    {1, 2, 3, 4, 5, 6},
    {1, 2, 3, 4, 5, 6}
};
```

Three-dimensional Arrays

- Declaration and initialization.

```
int x[2][3][2] = {
    { {0, 1}, {2, 3}, {4, 5} },
    { {6, 7}, {8, 9}, {10, 11} }
};
```

Dynamic Memory Allocation

- void *malloc(size_t size)

  malloc returns a pointer to space for an object of size size , or NULL if the request cannot be satisfied. The space is uninitialized.

- ptr = (cast-type*) malloc(byte-size)
  Example:
  int *ptr;
  ptr = (int*) malloc(100 * sizeof(int));
  Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

- If space is insufficient, allocation fails and returns a NULL pointer.

Dynamic Memory Allocation

- “calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
- It initializes each block with a default value ‘0’.
- It has two parameters or arguments as compare to malloc().
- void *calloc(size_t nobj, size_t size)
  calloc returns a pointer to space for an array of nobj objects, each of size size , or NULL if the request cannot be satisfied. The space is initialized to zero bytes.

Dynamic Memory Allocation

- void *realloc(void *p, size_t size)

  realloc changes the size of the object pointed to by p to size . The contents will be unchanged up to the minimum of the old and new sizes. If the new size is larger, the new space is uninitialized. realloc returns a pointer to the new space, or NULL if the request cannot be satisfied, in which case *p is unchanged.

Dynamic Memory Allocation

- void free(void *p) free deallocates the space pointed to by p; it does nothing if p is NULL . p must be a pointer to space previously allocated by calloc , malloc , or realloc.