Pointers in C

Basic Concepts

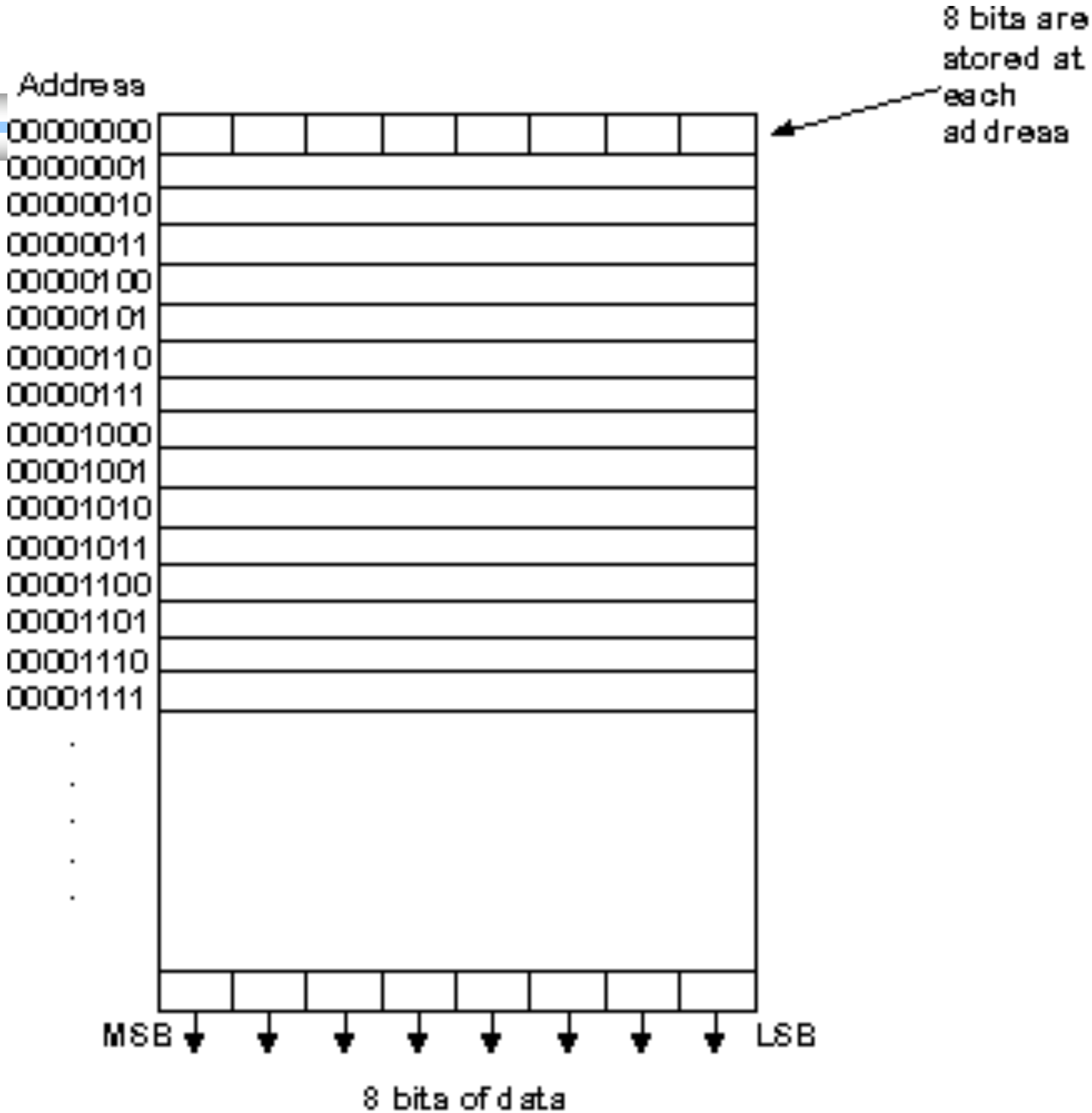
- In memory, every stored data item occupies one or more contiguous memory cells (each cell is of 1 byte).
  - The number of memory cells required to store a data item depends on its type (char, int, double, etc.).

- Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.
  - Since every byte in memory has a unique address, this location will also have its own (unique) address.


1
2
3
4
5
6
7
8
9


3


1
2
3
4
5
6
7
8
9
10


8 bits are stored at each address
Address
00000000
00000001
00000010
00000011
00000100
00000101
00000110
00000111
00001000
00001001
00001010
00001011
00001100
00001101
00001110
00001111
MSB
LSB
8 bits of data

Address	Content	Name	Type	Value
90000000	00	iii	int	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	sss	short	FFFF(-1_{10})
90000005	FF			
90000006	1F	ddd	double	1FFFFFFFFFFFFFFF ($4.450147717014402E\text{-}308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptr	int*	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

Introduction

- Consider the declaration,

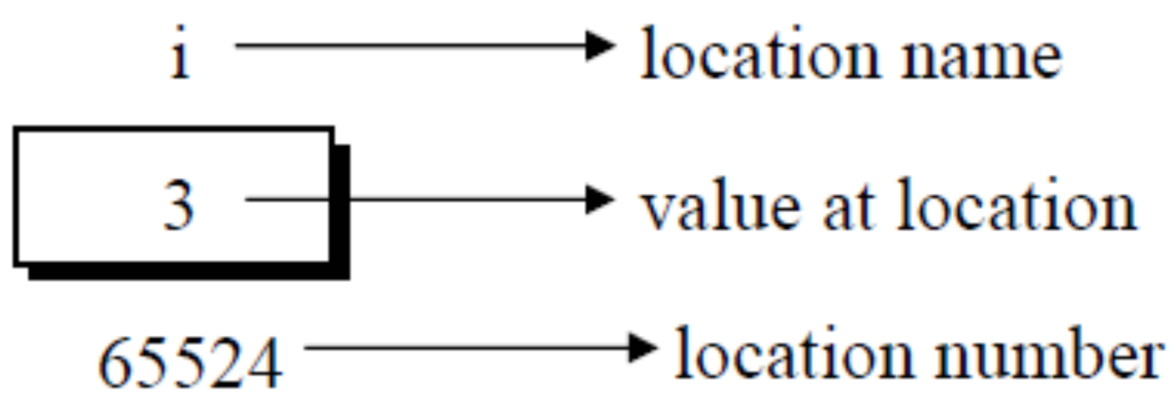
  int i = 3 ;

- This declaration tells the C compiler to:

  (a) Reserve space in memory to hold the integer value.

  (b) Associate the name **i with this memory location.**

  (c) Store the value 3 at this location.



Contd.

- Consider the statement

  int xyz = 50;

  – This statement instructs the compiler to allocate a location for the integer variable xyz, and put the value 50 in that location.

  – Suppose that the address location chosen is 1380.

xyz		Variable
50		Value
1380		address (assumption)

Contd.

- During execution of the program, the system always associates the name xyz with the address 1380.

  – The value 50 can be accessed by using either the name xyz or the address 1380.

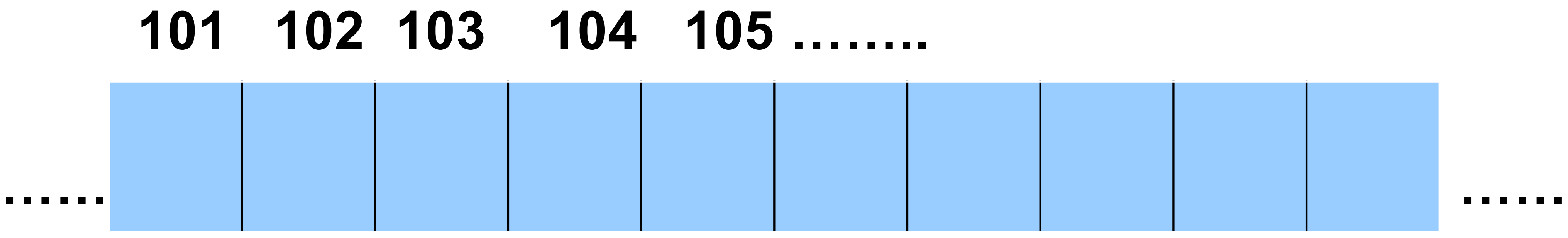
"Madam pl give the blue bag"

OR

Give the token and get the bag

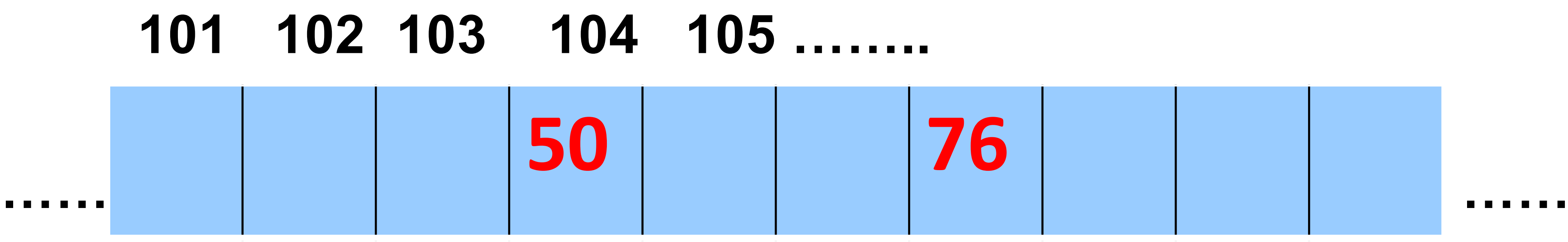
Address vs. Value

- Each memory cell has an address associated with it.



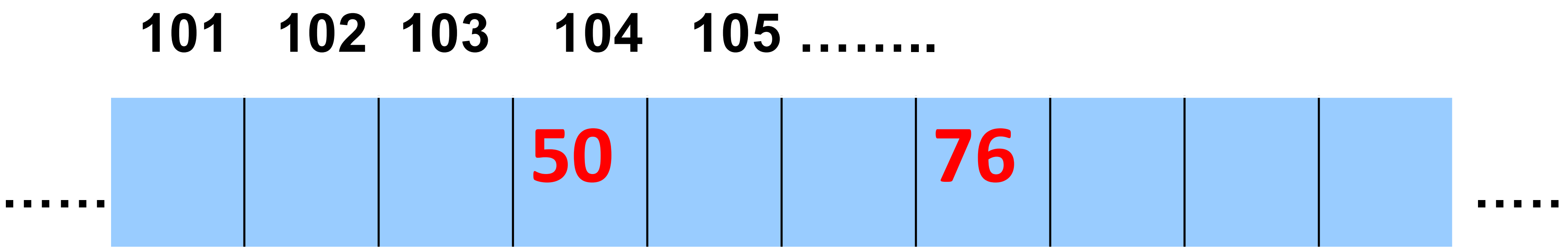
Address vs. Value

- Each memory cell has an address associated with it.
- Each cell also stores some value.



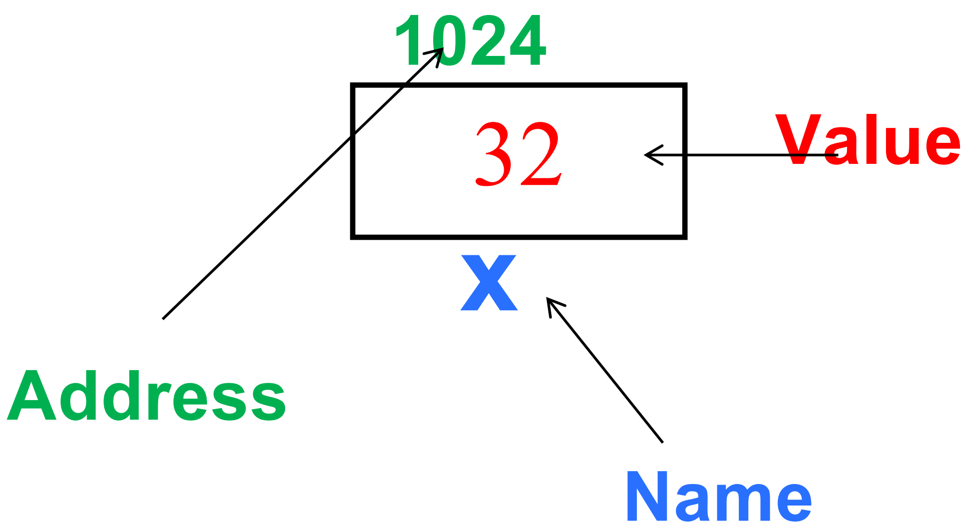
Address vs. Value

- Each memory cell has an address associated with it.
- Each cell also stores some value.
- Don't confuse the address referring to a memory location with the value stored in that location.



Values vs. Locations

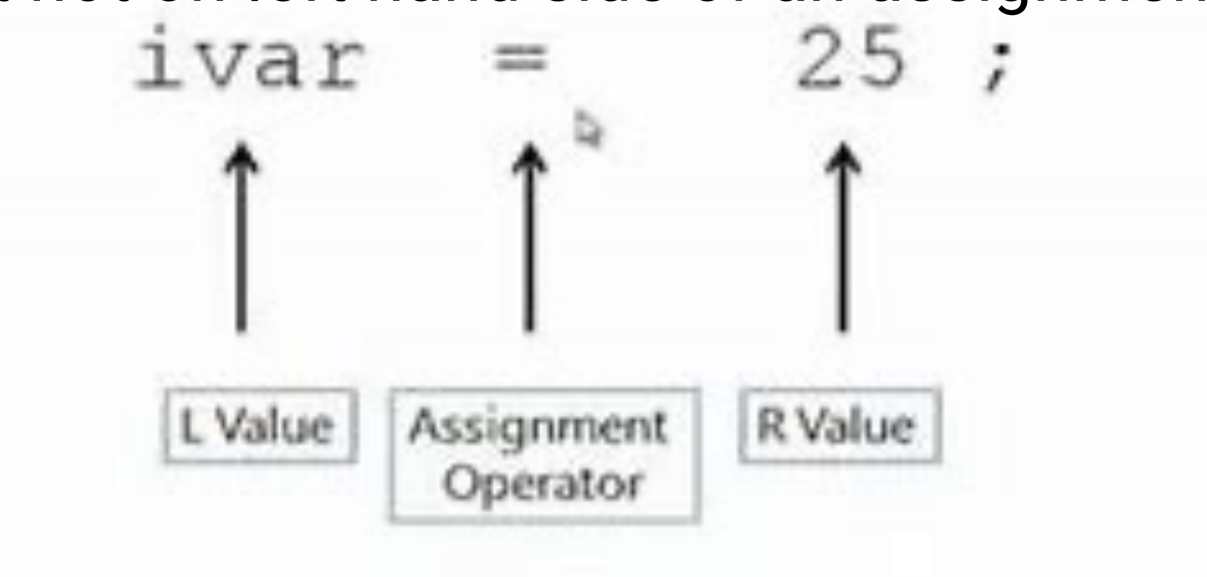
- **Variables name and memory locations, hold values.**



L- Value and R-Value

L-value: “l-value” refers to memory location which identifies an object. l-value may appear as either left hand or right hand side of an assignment operator(=). l-value often represents as identifier.

R-value: r-value” refers to data value that is stored at some address in memory. A r-value is an expression that can’t have a value assigned to it which means r-value can appear on right but not on left hand side of an assignment operator(=).



Introduction to Pointers

- Pointer is variable, just like other variables you studied.
  - so it has
    type,
    storage address,
    value etc.

- A pointer is a variable that holds the memory address of another variable.

- Since pointer is also a kind of variable , thus pointer itself will be stored at different memory location.

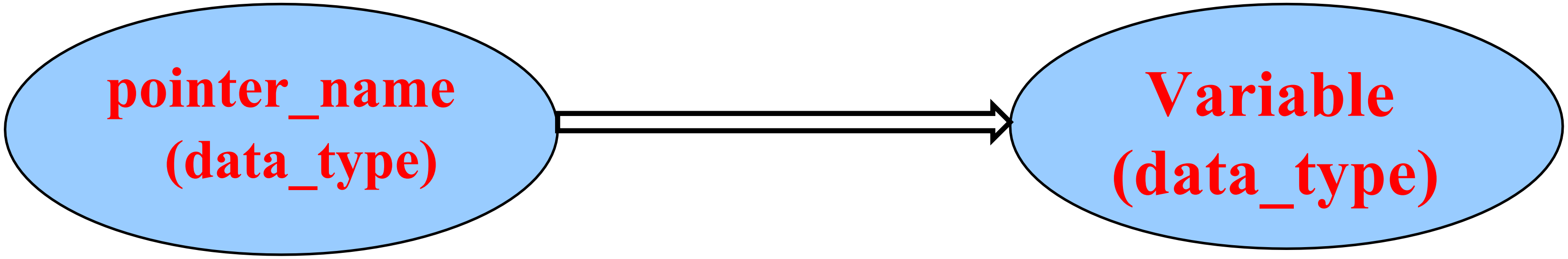
- Difference: it can only store the address (rather than the value) of a data item.

Pointer Declarations

- Like any variable, you must declare a pointer before using it to store any variable address.
- A pointer is just a C variable whose value is the address of another variable!
  - General form:

    data_type *pointer_name;

- Three things are specified in the above declaration:
  - The asterisk (*) tells that the variable pointer_name is a pointer variable.
  - pointer_name needs a memory location.
  - pointer_name points to a variable of type data_type.



Cont..

- Declaration of pointer

```
data_type * variable_name;
int  *ptr;
  int*  ptr;
  int  * ptr;
```

- After declaring a pointer like:

```
int *ptr;
```

ptr doesn't actually point to anything yet. We can either:

– make it point to something that already exists or

– allocate room in memory for something new that it will point to.

Pointer Declarations

Example:

```
int *count;

float *speed;

int * xp ;

double *salary;
```

Cont..

```
int x;
int * xp ;
```

Pointer to int

```
int *ip;          // pointer to an integer
double *dp;       // pointer to a double
float *fp;        // pointer to a float
char *ch;         // pointer to a character
```

Pointer

Recall:

Pointer is a variable which can store addresses

- **Pointer has a “type” (except void pointer)**

e.g.

int *p;

char *cp;

double *dp;

Here p, cp, dp are respectively pointers to integer, character, and double

Size of pointer is decided by compiler

Pointer Initialization

- **Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!**
- **Local variables in C are not initialized by default, they may contain anything.**

- **Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement like:**

```
int *p, xyz;
:
p = &xyz;
```

  – **This is called *pointer initialization.***

Pointer variable can be initialized with NULL or 0 value.

A side note on NULL

- NULL is not the number 0
- NULL is not necessarily the address 0
- NULL is just a special value for pointers told to us by C language.
  - Very often we need special values for a certain type
  - E.g. the value '\0' for a character is universally taken to be a special value indicating end of a character sequence in an array
  - INT_MAX , INT_MIN are values #defined in limits.h for integers
  - These type of values are used in programs to indicate either an *unused variable* or *empty variable* or *error value* on that variable
- Int i = NULL; char c = NULL; works
  - Why do you want to do it? Instead of saying int i = 0; char c = 0;

Operations on (and related to) Pointers

&

*****

=

+int -int

-

[ ]

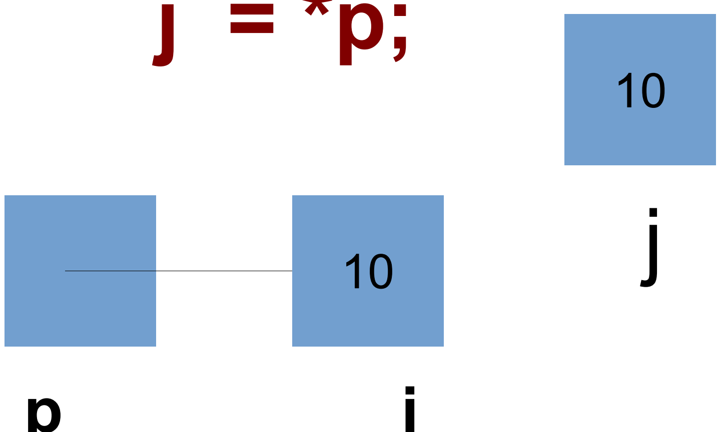
Operations related to pointers: & address / reference operator

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

- & fetches the address of variable
  - Called Referencing operator
  - Here, &i is address of i
  - RHS is address of integer, LHS is variable which can store address of integer
- Diagram of this operation is shown on left side
- No need of assuming some value for address (e.g. address 1028). Just the diagram is sufficient to understand the concept

Operations on pointers: *
value at/ dereferencing operator

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

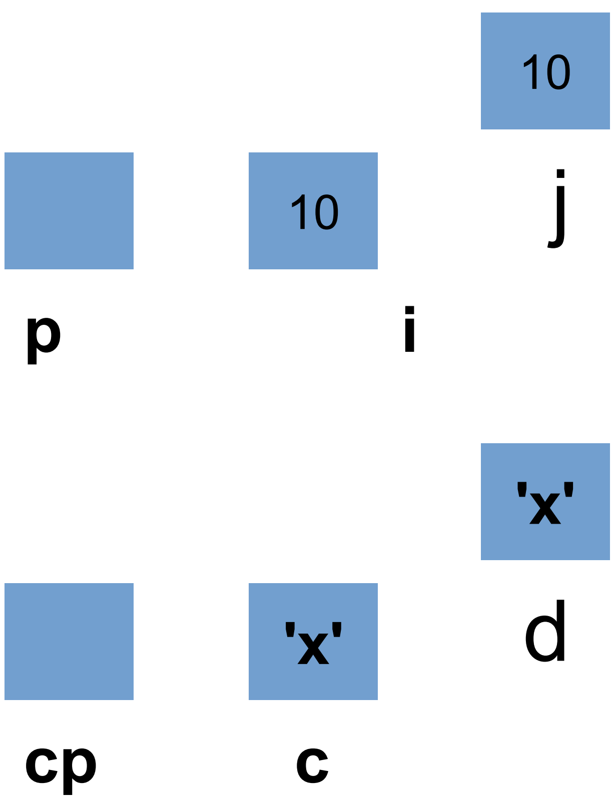


- * fetches the value stored at a given address
  - Called dereferencing operator
- *p : here “value of p” is itself an address *(of i)*, so *p is value stored at “value of p” that is i
- Diagram's make it easy to understand, *p is simply the contents of box p points to

Operations on pointers: *

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

```
char *cp, c, d;
c = 'x';
cp = &c;
d  = *cp;
```



What is the difference between * in the two codes?

The *p fetches the value at given address in sizeof(int) bytes, while *cp fetches the value at given address in sizeof(char)=1 bytes

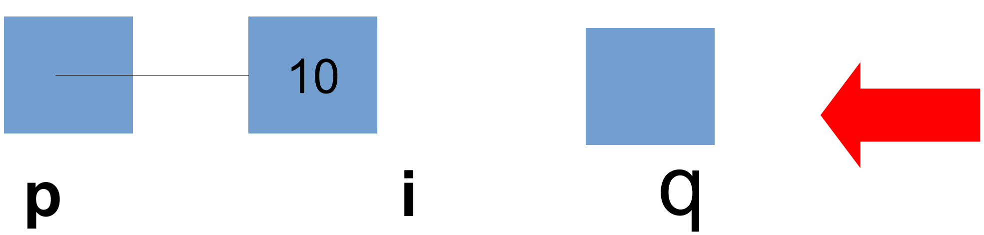
*** works based on the size of the type!**

Operations on pointers: =

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

i = 10;

p = &i;

q  = p;
```

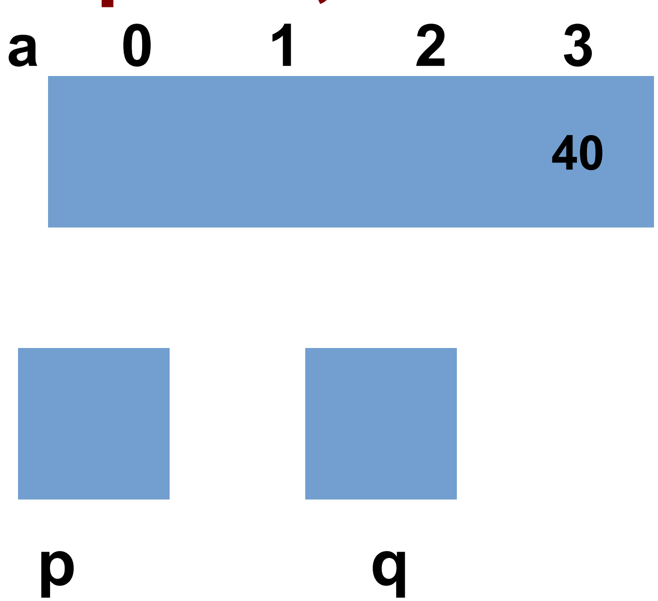
- Pointers can be copied
  - Can arrays be copied?
- Thumb rule:
  - After pointer copy, both pointers point to same location

- Common Mistake
  - One pointer pointing to another



Operations on pointers: +- int

```
int *p, a[4],
*q;
p =  &a[1];
q = p + 2;
*q = 40;
```



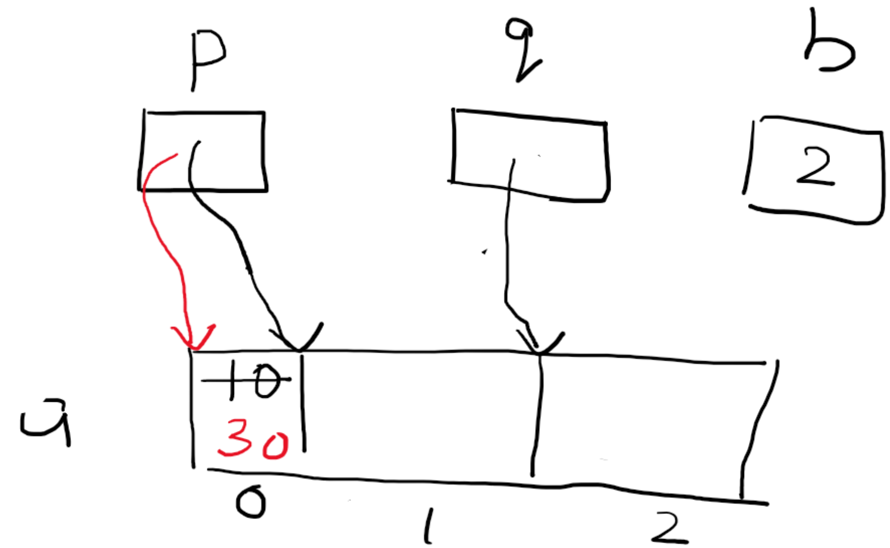
- C allows adding or subtracting an int from a pointer!
  - Weird, but true!
  - e.g. int *p, n; p + n;
- The result is a pointer of the same type
- The resultant pointer points *n type locations ahead(for +) or before (for -)*
- *A type location is equal to sizeof(type).*

Problems: Draw diagrams for the code

```
int main() {
    int *p, *q, a[3],
b;
    a[0] = 10; b = 2;
    p = &a[1];
    q = p + 1;
    p = q - b;
    *p = 30;
}
```

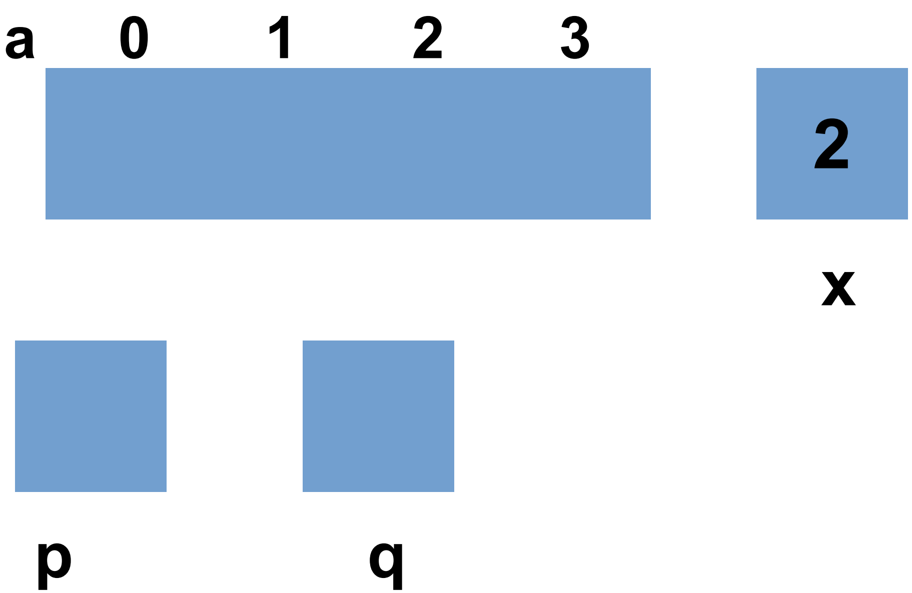
```
int main() {
    int *p, *q, a[3], b;
    a[0] = 10; b = 1;
    p = &a[3];
    q = p - 3;
    p = q + 1;
    *(q + 1) = 30;
    *(p - 1) = 20;
}
```

```
int *p, *q, a[3], b;
a[0] = 10; b = 2
p = &a[1];
q = p+1;
p = q-b;
*p = 30;
```



Operations on pointers: substracting two pointers

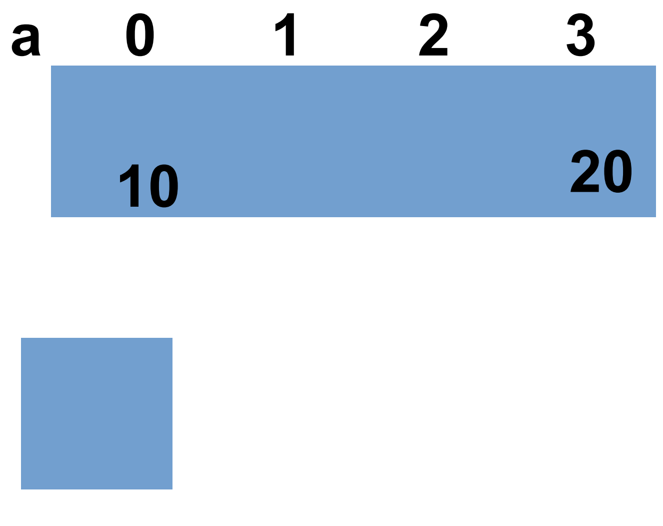
```
int *p, a[4], *q, x;

p =  &a[1];

q = p + 2;

x = q – p;
```



- Two pointers of the *same type* can be substracted
  - Result type is int
  - Result value = no. Of elements of sizeof(type) between two pointers
- Logically derives from adding/substracting int to pointers
  - p = q + 2 ==> p – q = 2

Operations on pointers: [ ]

```
int *p, a[4],
*q;

p =  &a[1];

p[2] = 20;

p[-1] = 10;
```



- Interestingly, C allows [ ] notation to be applied to all pointers!
  - You must be knowing that [ ] is normally used for arrays
- p [i] means * (p + i)
  - P is a pointer and i is an integer (or i is a pointer and p is a pointer is also allowed)

A peculiar thing about arrays

Name of an array is equivalent to the address of (the zeroeth element) the array

int a[3];

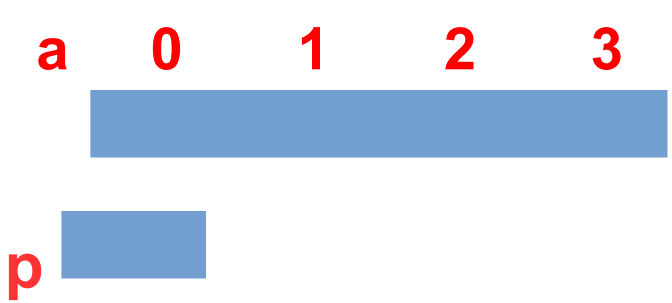
Now

a means *&a[0]*

Because it's an address, it can be stored in a pointer

int a[3], *p;

p = a;



What do the following mean?

int a[3] = {1, 2, 3}, *p;

a + 1;

*(a + 1);

p = (a + 2)

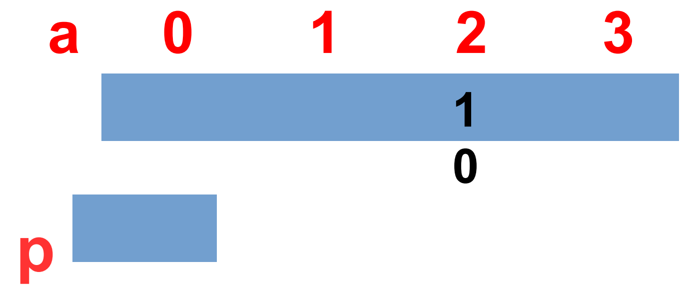
Pointer *as if it was an array*

- Combine the concepts of
  - Pointer arithmetic (+- int)
  - [ ] notation for pointers
  - Array name as address of array

```
int a[3], *p;
p = a;
p[2] = 10;
```

Here we are using p as if it was an array name

Possible only if p was pointing to array base



Pointers != Arrays

Array is a continuous collection of elements of one type.

Array has name, the name also stands for the address of the array

[ ] is allowed operation on arrays

Array name can't be reassigned to another address

Pointer is a variable that can store an address

Pointers can be of various types

[ ], = , +- int, substraction are operations on pointers

Pointers can be reassigned to point to different addresses

Concept of (Binding) “Time”

- **Compile Time**
  - When you are running commands like

    cc program.c -o program

- **Load time**
  - After you type commands like

    ./program

    Before the main() starts running

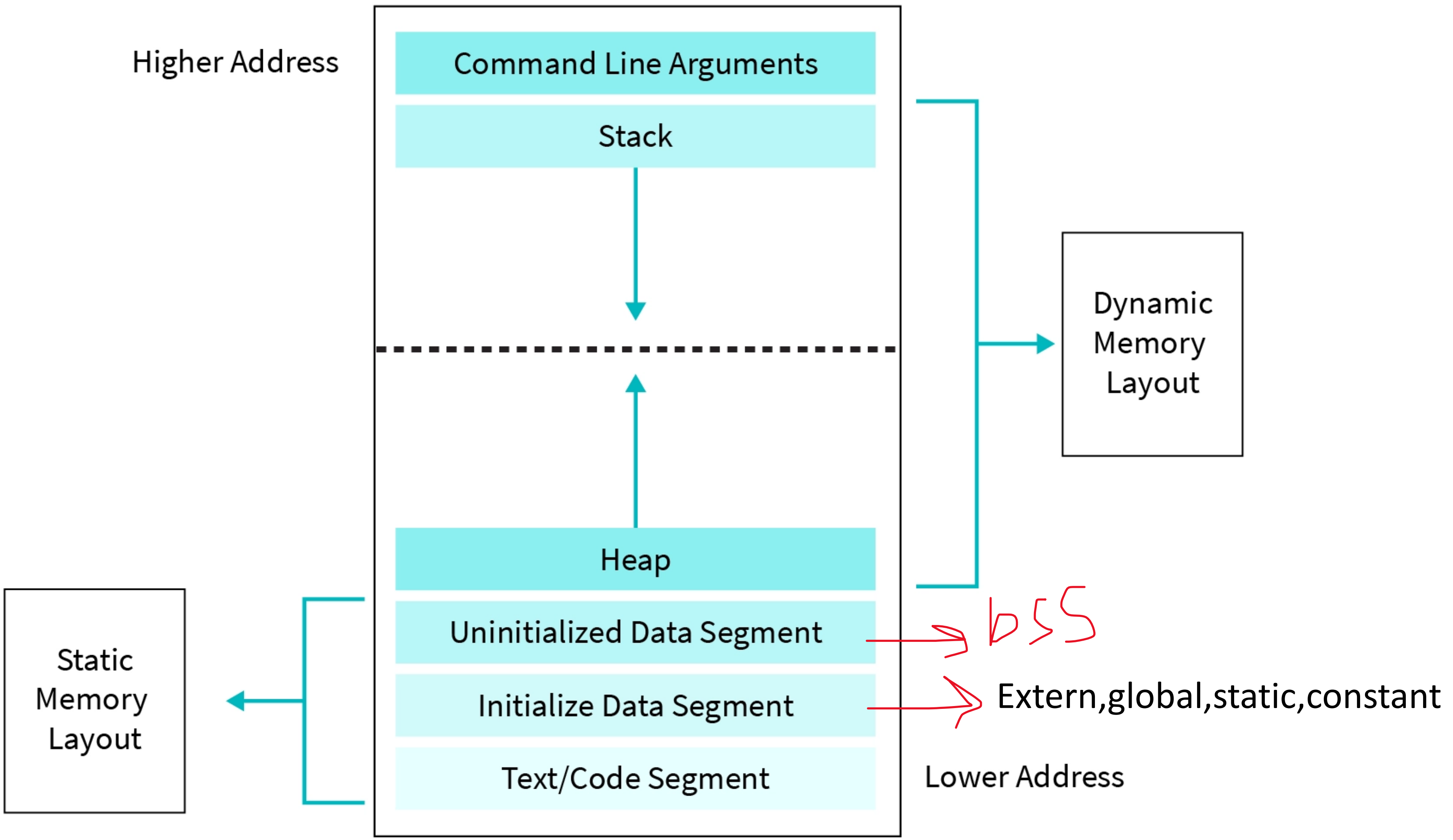
- **Program Run Time**
  - After you type commands like

    ./program

    When the main() of the program starts running, till it exits

- **Function Call Time**
  - Interval between the call of a function and before the called function starts running. Part of “Run Time”.

Memory Representation in C



Memory Layout

- **binary file loads into RAM in an organized manner**
- **memory layout in C Program has six components which are text segment, initialized data segment, uninitialized data segment, command-line arguments, stack, and heap.**
- **If a program tries to access the value stored in any segment differently than it is supposed to, it results in a segmentation fault error.**

Initialized data segment

- contains values of all external, global, static, and constant variables whose values are initialized at the time of variable declaration in the program.
- read-write permission.
- We can further classify the data segment into the read-write and read-only areas.
- const variable comes under the read-only area.
- The remaining types of variables come in the read-write area.

Uninitialized data segment

- bss (block started by symbol).
- The program loaded allocates memory for this segment when it loads.
- Every data in bss is initialized to arithmetic 0 and pointers to null pointer by the kernel before the C program executes.
- BSS also contains all the static and global variables, initialized with arithmetic 0.
- Because values of variables stored in bss can be changed, this data segment has read-write permissions.

Stack

- follows the LIFO (Last In First Out) structure and grows down to the lower address, but it depends on computer architecture.
- Stack grows in the direction opposite to heap.
- Stack segment stores the value of local variables and values of parameters passed to a function along with some additional information like the instruction's return address, which is to be executed after a function call.

Heap

- allocated during the run time (dynamically allocated memory).
- Heap generally begins at the end of bss segment and, they grow and shrink in the opposite direction of the Stack.
- Commands like malloc, calloc, free, realloc, etc are used to manage allocations in the heap segment which internally use sbrk and brk system calls to change memory allocation within the heap segment.
- Heap data segment is shared among modules loading dynamically and all the shared libraries in a process.

Command-line arguments

- **When a program executes with arguments passed from the console like argv and argc and other environment variables, the value of these variables gets stored in this memory layout in C.**

Lifetime of variables and Memory Allocation

- **Global Variables, Static Variables (g, t, and s here)**
  - Allocated Memory at *load time*
  - They are alive (available) till the program exits
- **Local Variables, Formal Arguments (i, j, k in main; a, b, x, y in f)**
  - Allocated Memory on *function call*
  - They are alive (available) as long as function is running
- **Dynamically allocated memory (Run time allocation)**
  - Allocated on explicit call to functions like *malloc()*
  - Alive as long as functions like *free()* are not called on the memory

```
int g;
static int t = 20;
int f(int a, int b) {
    int x, y;
    static int s = 10;
    x = a + b + 5 + g + s;
    return x;
}
int main() {
    int i, j, k, *p;
    g = 10; i = 20; j = 30;
    p = malloc(8);
    k = f(i, j);
}
```

Dynamic Memory Allocation

malloc()

- malloc() is a standard C library function for allocating memory dynamically (at run time)
- #include <stdlib.h> for malloc()
- Function prototype
  - Run “man malloc” to see it
  - void *malloc(size_t size);
  - size_t is a typedef in stdlib.h
    - size_t is unsigned long
  - Reads a number, allocates those many bytes and returns the address of allocated memory (zero'th byte)
- Additional info: malloc() gets the memory from the OS and then gives to your program

void *

A void pointer is a typeless pointer;

- Pure address
- No type --> No “size” information about the type

You can declare a void pointer

- void *p, *q

Void pointer can store *any* address

- void *p;
- int a;
- p = &a; char c;
- p = &c;

void *

Void pointers can be copied

    void *p, *q;

    int a;

    p = &a;

     q = p;

    q also stores address of ‘a’ now

The Dereferencing operator has no meaning when applied to a void pointer

    void *p; int a; p = &a;

    What does *p mean now?

- Needs “size” of the type for its work. Void pointers have no type and so no size information associated with it.

    Note: [ ] is also dereferencing

malloc()

- malloc() returns a “void *”
  - Returns a pure address
  - This address can be stored in a “void *” variable
  - This address can also be stored in any pointer variable
- Suppose we do
  - void *p;
  - p = malloc(8);
  - Now what meaningful operations can we do with the malloced memory? --> only copy!
  - So normally return value of malloc is stored using some typed pointer

malloc()

```
int *p;
p = malloc(8)
```

This code allocates 8 bytes and then pointer p will point to the malloced memory

This code can result in a “warning” because we are converting “void *” to a “int *” with '='

malloc()

```
int *p;
p  = (int *) malloc(8);
```

This code does away with the warning as we are converting the "void *" into "int *" using explicit type casting

Suppose size of integer was 4 bytes, then what does this code mean?

malloc()

int *p;

p = (int *) malloc(8);

p points to 8 byte location

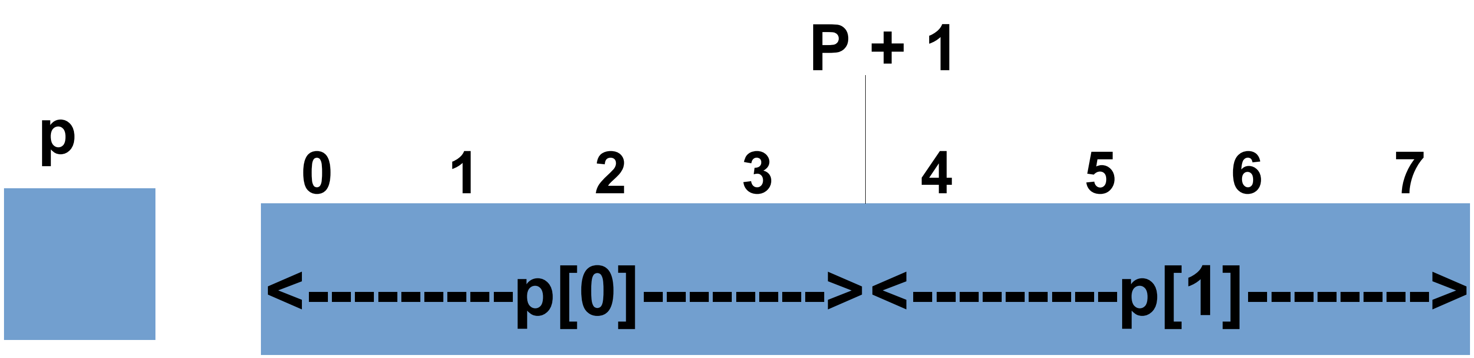
However now, *p means deferencing in “4 bytes”

as size of int is 4 bytes

p[0] means *(p + 0) that is *p

p[1] means *(p + 1) where (p + 1) is a pointer 4 bytes ahead of p

Using this trick we are treating the 8 bytes as if it was a 2 integer array !



malloc()

```
int *p;

p  = (int *) malloc(8);
```

p points to 8 byte location

However now, *p means deferencing in “4 bytes”

as size of int is 4 bytes

p[0] means *(p + 0) that is *p

p[1] means *(p + 1) where (p + 1) is a pointer 4 bytes ahead of p

Using this trick we are treating the 8 bytes as if it was a 2 integer array !

Malloc(): Allocating arrays dynamically

```
int *p;

p  = (int *) malloc(

      sizeof(int) * 4);
```

We can allocate arrays of any type dynamically using malloc()

Use of sizeof(int) here makes sure that the code is *portable,* appropriately sized array will be allocated irrespective of size of integer

- Code on earlier slide assumes 4 byte integer

This code allocates array of 4 integers

- Can be accessed as p[0], p[1], p[2] and p[3]

malloc(): Allocating array of structures

```
typedef struct test {
        int a, b;
        double g;
}test;
test *p;
p  = (test *) malloc(sizeof(test) * 4);
```

- This code allocates an array of 4 structures
- p points to the array of structures
- p[0] is the 0^{th} structure, p[1] is the 1^{st} structure ...
- p[0].a, p[1].g is the way to access the inner elements of structures

free()

free() will give the malloced memory back

malloc() and free() work together to manage what is called as "heap memory" which the memory management library has obtained from the OS

Usage

```
void free(void *p);
```

free() must be given an address which was returned by malloc()

Rule: Every malloc() must have a corresponding free()

DRAW

```
typedef struct slot {
    int value;
    char arr[10];
    char *cp;
    struct slot *sp;
}slot;
int main() {
    slot a, b, *p, *q;
    a.sp = &b;
    p = &b;
    q = &a;
    p->sp = &a;
    p->cp = &(q->arr[5]);
    b.cp = &(b.arr[1]);
    p->value = b.cp - p->cp;
    q->value = 10;
    strcpy(b.arr, "hello");
    return 0;
}
```

Self Referential Pointers in Structures

Self Referential Pointers

- "Self Referential Pointer" is a kind of a misnomer
- C allows structures like this

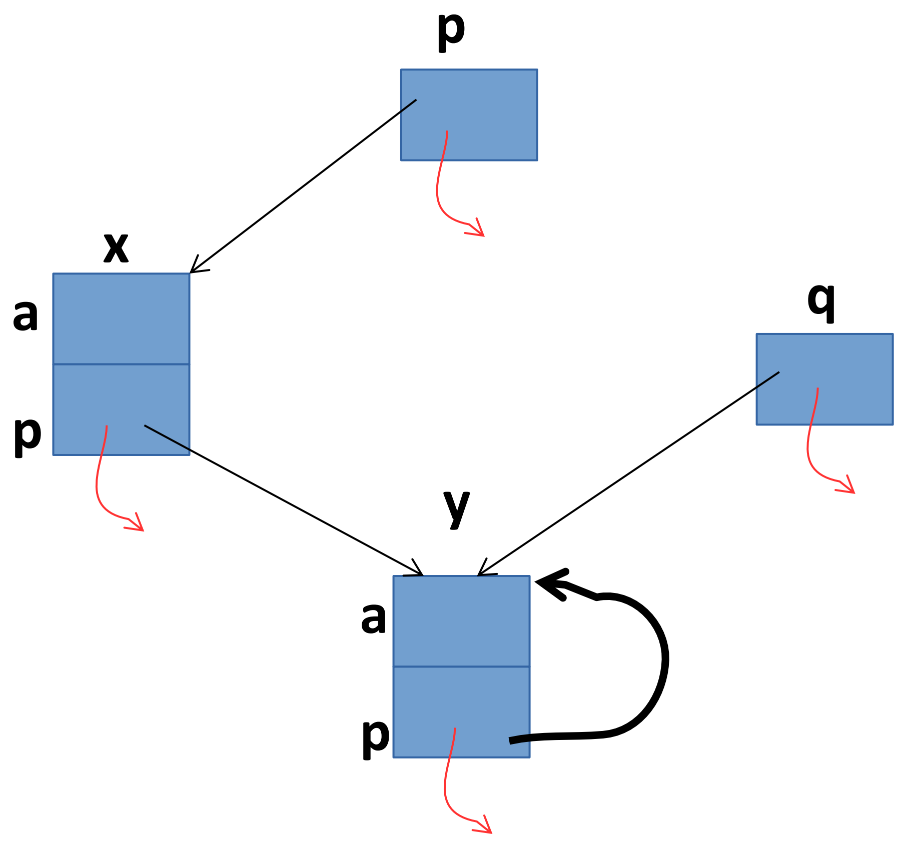
```
struct test {
  int a;
  struct test *p;
};
```

- The pointer p, can point to any variable of the type "struct test" (or be NULL)

Self Referential Pointers

- Consider following code

```
typedef struct test {
    int a;
    struct test *p;
}test;
test  x, y, *p, *q;
p = &x;
x.p = &y;
q = &y;
y.p = &y;
```



Self referential structures allow us to create a variety of "linked" *structures of data*

-> notation

```
typedef struct test {
    int a;
    struct test *p
}test;
test  m, n, *x;
m.a = 20;
x = &n;
(*x).a = 40;
x->a = 50;
x->p = &m;
x->p->p = x;
```

(*x) is the entire structure to which x points

(*x).a is the variable 'a' in that structure

x->a is another notation for (*x).a