A pointer is a variable that contains the address of a variable.

Pointer declaration

A pointer is a variable that contains the memory location of another variable.

The asterisk tells the compiler that you are creating a pointer variable.

Finally you give the name of the variable.
datatype `* variable name;`

Example:

```
int *ptr;
float *string;
```

Address operator

Once we declare a pointer variable we must point it to something we can do this by assigning to the pointer the address of the variable you want to point as in the following example:

int num, *ptr;

```
ptr=&num;
```

This places the address where num is stored into the variable ptr.

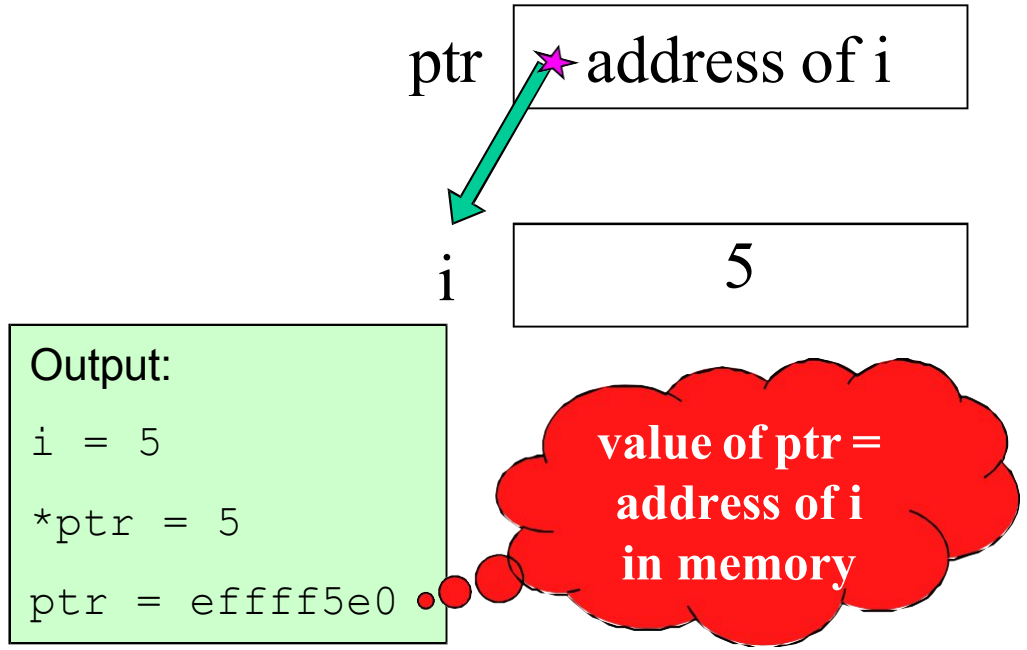
If num is stored in memory 21260 address then

the variable ptr has the value 21260.

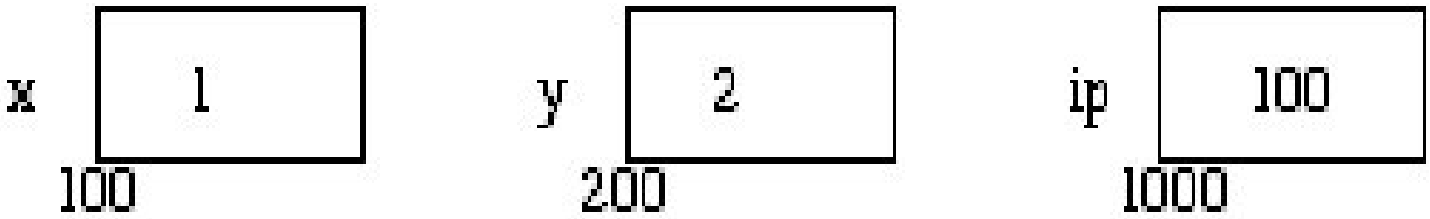
What actually *ptr* is?

- **ptr** is a variable storing **an address**
- ptr is **NOT** storing the actual value of i

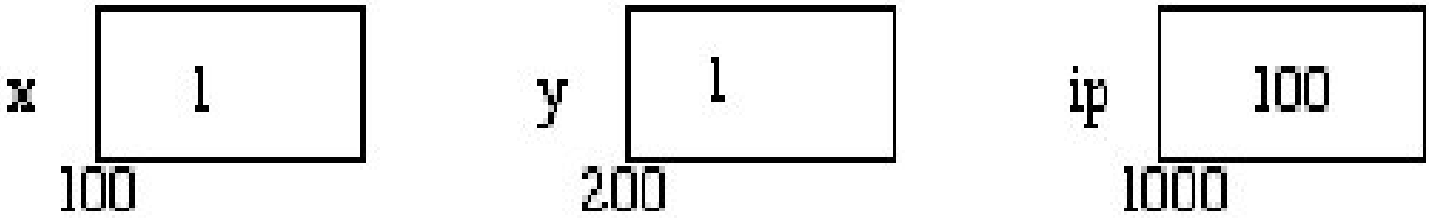
```
int i = 5;
int *ptr;
ptr = &i;
printf("i = %d\n", i);
printf("*ptr = %d\n", *ptr);
printf("ptr = %p\n", ptr);
```



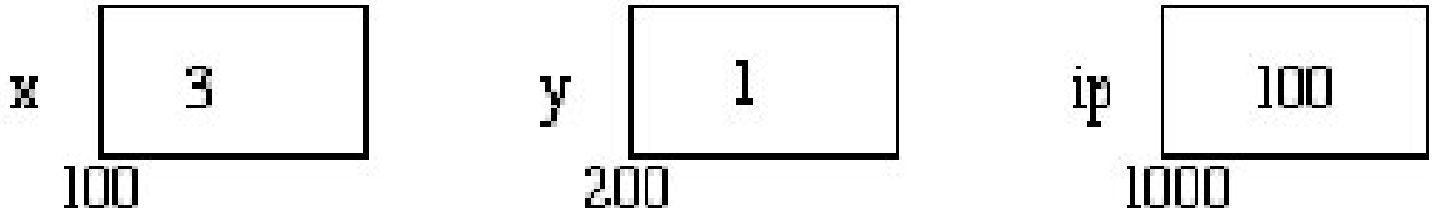
```
int x = 1, y =2;
int *ip;

ip = &x;
```




```
y = *ip;
```

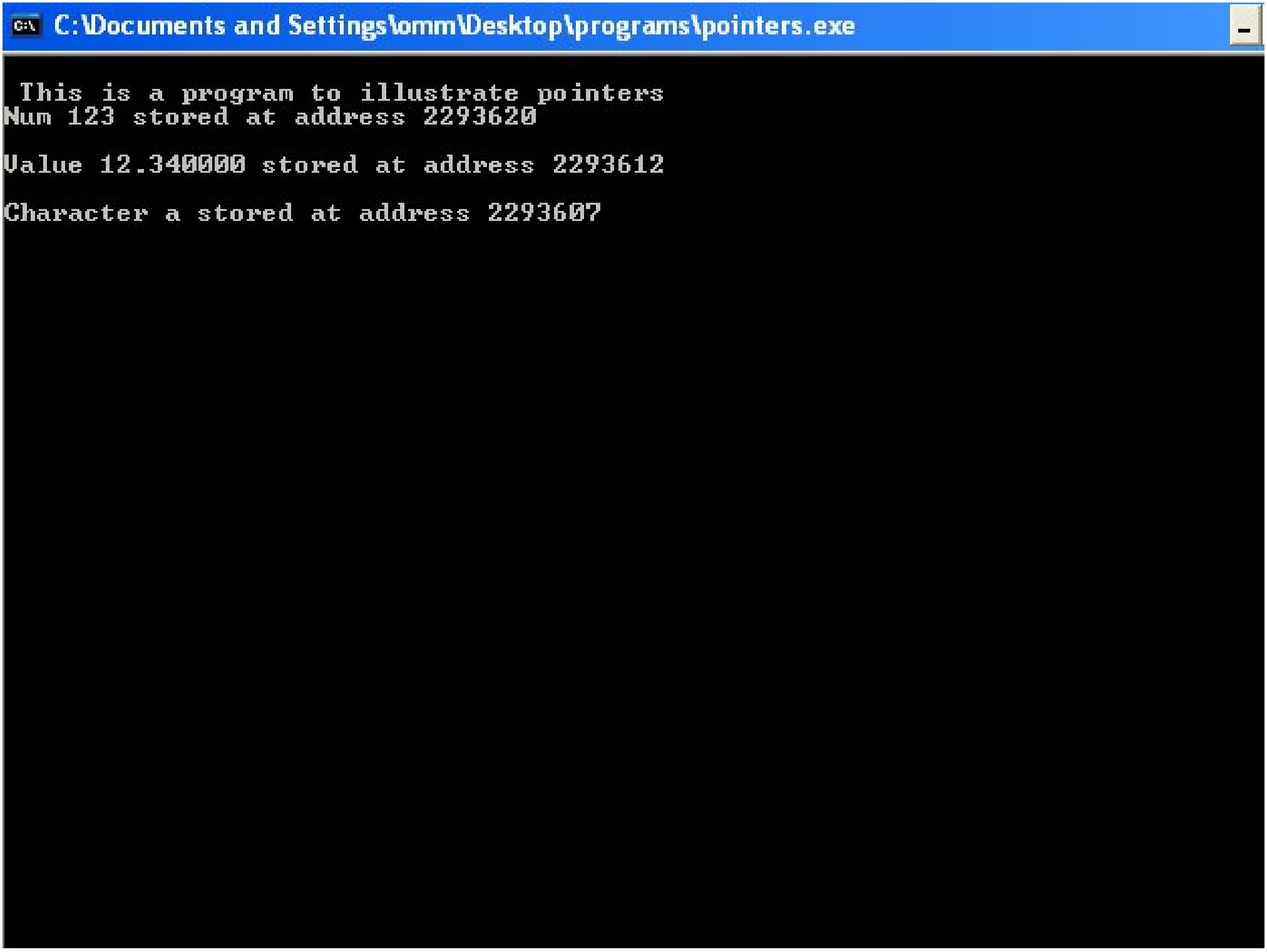



```
*ip = 3
```



```
#include <stdio.h>
int main()
{
printf("\n This is a program to illustrate pointers");
/* Program to display the contents of the variable their address using pointer variable*/
int num, *intptr;
float x, *floptr;
char ch, *cptr;
num=123;
x=12.34;
ch='a';
intptr=&num;
cptr=&ch;
floptr=&x;
printf("\nNum %d stored at address %u\n",*intptr,intptr);
printf("\nValue %f stored at address %u\n",*floptr,floptr);
printf("\nCharacter %c stored at address %u\n",*cptr,cptr);
getchar();
```

```
return 0;
}
```


C:\Documents and Settings\omm\Desktop\programs\pointers.exe
This is a program to illustrate pointers
Num 123 stored at address 2293620
Value 12.340000 stored at address 2293612
Character a stored at address 2293607

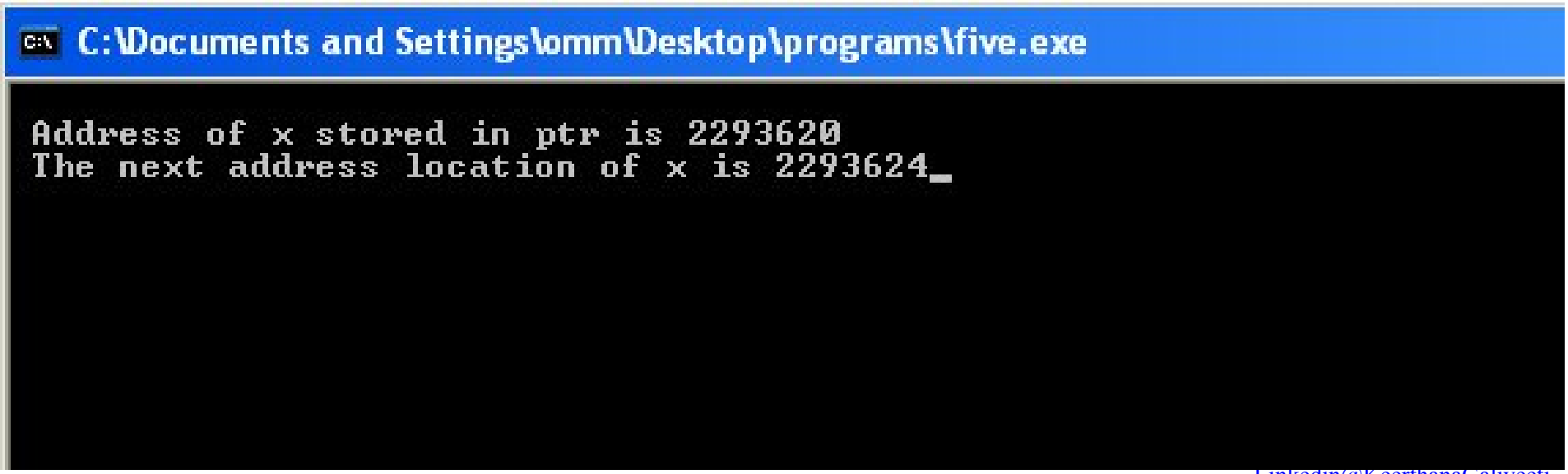
The reason we associate a pointer to a data type is so that it knows how many bytes the data is stored in.

When we increment a pointer we increase the pointer by one ``block'' memory.

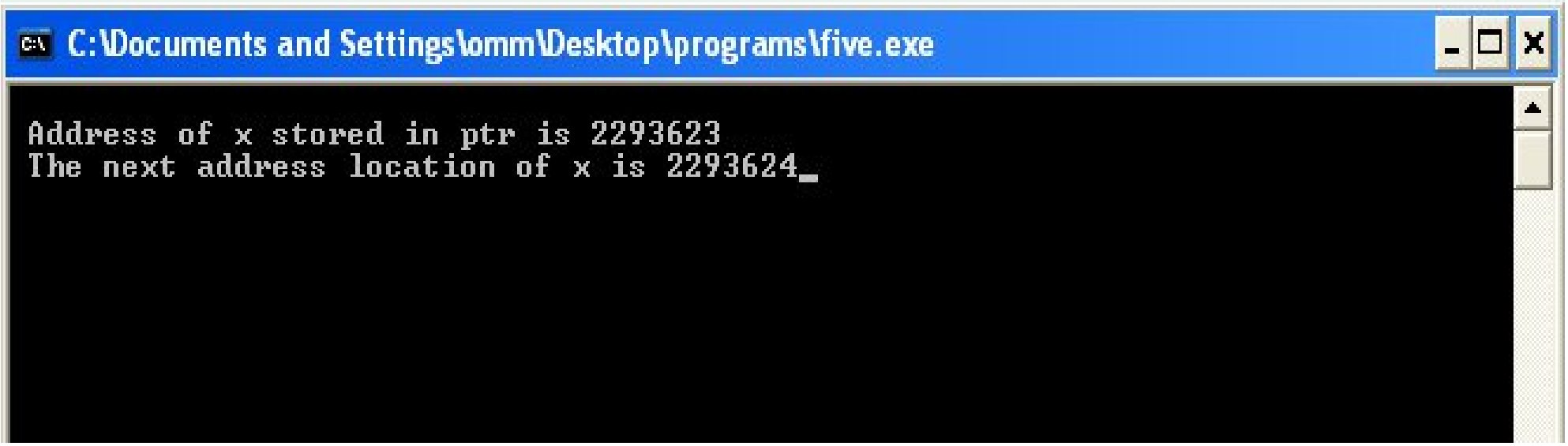
So for a character pointer ++ch_ptr adds 1 byte to the address.

For an integer or float ++ip or ++flp adds 4 bytes to the address.

```c
#include <stdio.h>
main(){
int x;
int *ptr;
ptr=&x;

printf("\n Address of x stored in ptr is %u",ptr);

ptr++;
printf("\n The next address location of x is %u",ptr);

getchar();
return 0;
}
```



```c
#include <stdio.h>
main(){
char x;
char *ptr;
ptr=&x;

printf("\n Address of x stored in ptr is %u",ptr);

ptr++;
printf("\n The next address location of x is %u",ptr);

getchar();
return 0;
}
```



```
#include <stdio.h>
main(){
int x;
int *ptr;
ptr=&x;

printf("\n Address of x stored in ptr is %u",ptr);

ptr++;
printf("\n The next address location of x is %u",ptr);

getchar();
return 0;
}
```

```
#include <stdio.h>
main(){
char x;
char *ptr;
ptr=&x;

printf("\n Address of x stored in ptr is %u",ptr);

ptr++;
printf("\n The next address location of x is %u",ptr);

getchar();
return 0;
}
```

A pointer is only useful if there's some way of getting at the thing that it points to;

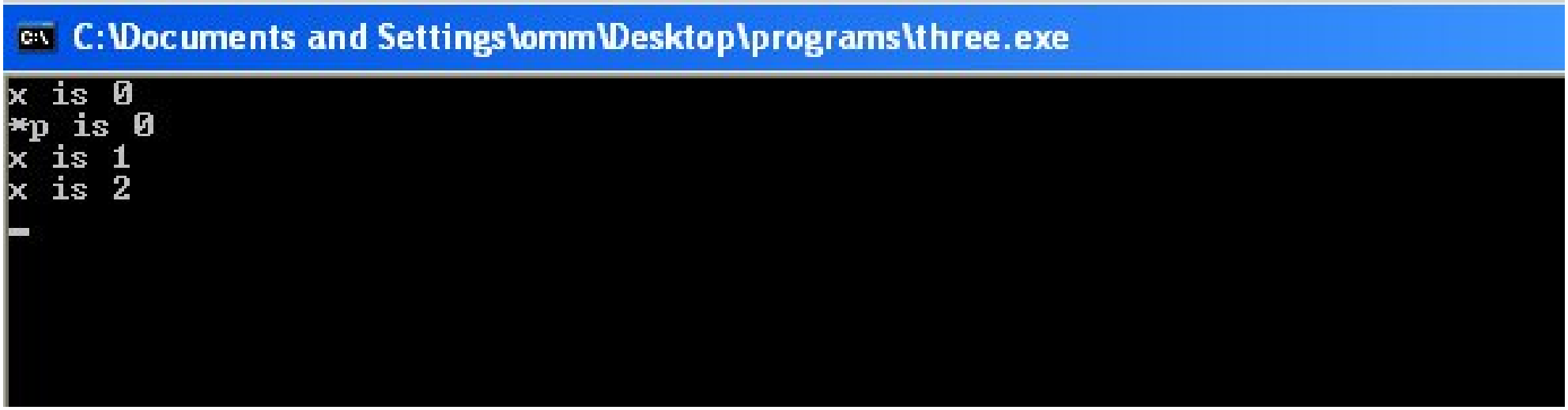
C uses the unary * operator for this job.

If p is of type ‘pointer to something', then *p refers to the thing that is being pointed to.

```
#include <stdio.h>
main(){
    int x, *p;
    p = &x;                          /* initialise pointer */
    *p = 0;                          /* set x to zero */
    printf("x is %d\n", x);
    printf("*p is %d\n", *p);
    *p += 1;                 /* increment what p points to */
    printf("x is %d\n", x);
    (*p)++;                  /* increment what p points to */
    printf("x is %d\n", x);
```

```
getchar(); return 0;  }
```

```c
#include <stdio.h>
main(){
      int x, *p;

      p = &x;          /* initialise pointer */
      *p = 0;          /* set x to zero */
      printf("x is %d\n", x);
      printf("*p is %d\n", *p);

      *p += 1;         /* increment what p points to */
      printf("x is %d\n", x);

      (*p)++;          /* increment what p points to */
      printf("x is %d\n", x);
getchar();
return 0;
}
```



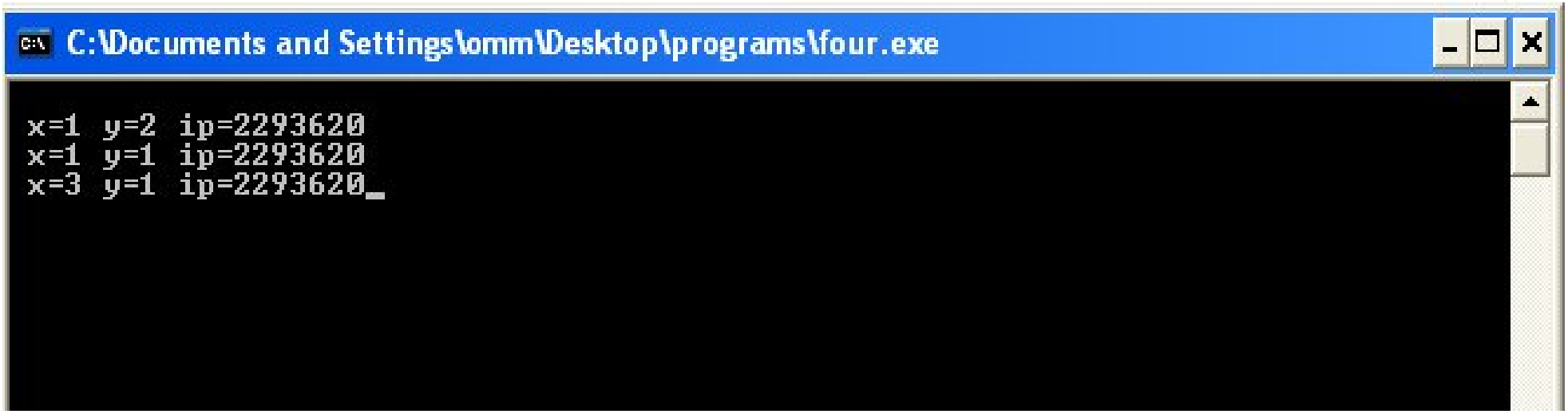
```c
#include <stdio.h>
main(){
int x=1, y=2;
int *ip;

ip=&x;
printf("\n x=%d y=%d ip=%u",x,y,ip);


y=*ip;
printf("\n x=%d y=%d ip=%u",x,y,ip);

*ip=3;
printf("\n x=%d y=%d ip=%u",x,y,ip);

getchar();
return 0;
}
```



```
#include <stdio.h>
main(){
int x=1, y=2;
int *ip;

ip=&x;
printf("\n x=%d y=%d ip=%u",x,y,ip);

y=*ip;
printf("\n x=%d y=%d ip=%u",x,y,ip);

*ip=3;
printf("\n x=%d y=%d ip=%u",x,y,ip);
```

```
getchar();
return 0;    }
```

When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it. So ...

int *ip; *ip = 100; will generate an error (program crash!!).

The correct use is:

int *ip; int x; ip = &x; *ip = 100;

Why pointers?

- Ex: To create a **text editor.**
- Option other than pointers:
  - T**wo-dimensional array**.
- Let's say you set a maximum of 10 open files at once, a maximum line length of 1,000 characters and a maximum file size of 50,000 lines. Your declaration now looks like this:
  - char doc[50000][1000][10];
- Most computers today do not have the RAM, or even the virtual memory space, to support an array that large.

Gont.

Why pointers?

- With arrays:
- Extravagant waste of space.
- Strange to declare a 500 million character array to look at 100 lines of code.
- Have to declare it to have its maximum size in every dimension from the beginning, while there is no way to predict and handle the maximum line length of a text file, because, technically, that number is infinite.

Gont.

Why pointers?

- With pointers:
- You can create **dynamic data structures**.
- **Allocation** of memory from **the heap** while the program is running.
- Ability to use the **exact amount of memory** a document needs, with **no waste**.
- Ability to **return the memory to the heap** when you close a document.
- **Memory can be recycled** while the program is running.

Initialization of Pointers

- Can be initialized four different ways.
  1. Using malloc statement
  2. Using statement such as p = q . If q is pointing at a valid block, p is initialized.
  3. Pointing to a known address, such as a global variable's address. !x. p=&i
  4. Using value zero i.e., p = 0; or: p = NULL;

•One way, as seen already, is to use the malloc statement. This statement allocates a block of memory from the heap and then points the pointer at the block. This initializes the pointer, because it now points to a known location. The pointer is initialized because it has been filled with a valid address -- the address of the new block.

•The second way, as seen just a moment ago, is to use a statement such as **p = q** so that p points to the same place as q. If q is pointing at a valid block, then p is initialized. The pointer p is loaded with the valid address that q contains. However, if q is uninitialized or invalid, p will pick up the same useless address.

•The third way is to point the pointer to a known address, such as a global variable's address. For example, if i is an integer and p is a pointer to an integer, then the statement **p=&i** initializes p by pointing it to i.

•The fourth way to initialize the pointer is to use the value zero. Zero is a special values used with pointers, as shown here:

```
p = 0;
```

or:

```
p = NULL;
```

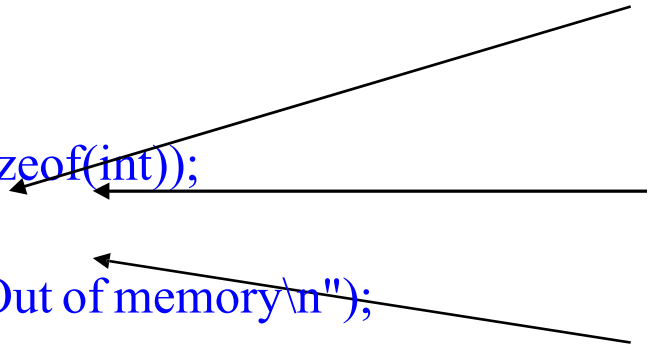
malloc and free

- **malloc( )** is used to allocate a certain amount of memory during the execution of your application.
- OS will reserve a block of memory for your program,
- You can use it in any way you like.
- When you are done with the block, you return it to the OS for recycling by calling the **free( )**.

Gont.

malloc and free

```
int main()
{
  int *p;
  p = (int *)malloc(sizeof(int));
  if (p == 0) {
    printf("ERROR: Out of memory\n");
    return 1;
  }
  *p = 5;
  printf("&d\n", *p);
  free(p);
  return 0;
}
```



±!nough memory available to allocate a block of memory of the size requested?

±Reserves a block from the heap of the size specified, if available..

±System places into the pointer variable, address of the reserved block.

Gont.

malloc( ) does three things:

1. The malloc statement first looks at the amount of memory available on the heap and asks, "Is there enough memory available to allocate a block of memory of the size requested?" The amount of memory needed for the block is known from the parameter passed into malloc -- in this case, **sizeof(int)** is 4 bytes. If there is not enough memory available, the malloc function returns the address zero to indicate the error (another name for zero is NULL and you will see it used throughout C code). Otherwise malloc proceeds.
2. If memory is available on the heap, the system "allocates" or "reserves" a block from the heap of the size specified. The system reserves the block of memory so that it isn't accidentally used by more than one malloc statement.
3. The system then places into the pointer variable (p, in this case) the address of the reserved block. The pointer variable itself contains an address. The allocated block is able to hold a value of the type specified, and the pointer points to it.

if (p == 0) could have also been written as **if (p == NULL)** or even **if (!p)** .

malloc and free

- Let me ask you two Questions:
  - **Is it really important to ckecE tkat tke pointer is zero a ter eack allocation?**
  - **Wkat kappens i I orget to delete a blocE o memory be ore tke program terminates?**

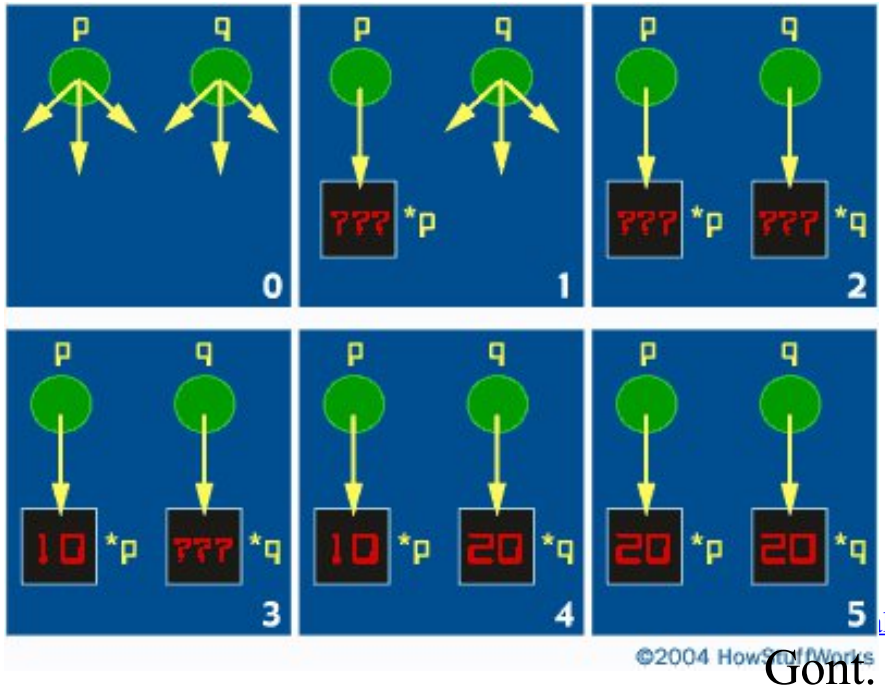
Gont.

•**Is it really important to check that the pointer is zero after each allocation?** Yes. Since the heap varies in size constantly depending on which programs are running, how much memory they have allocated, etc., there is never any guarantee that a call to malloc will succeed. You should check the pointer after any call to malloc to make sure the pointer is valid.

•**What happens if I forget to delete a block of memory before the program terminates?** When a program terminates, the operating system "cleans up after it," releasing its executable code space, stack, global memory space and any heap allocations for recycling. Therefore, there are no long-term consequences to leaving allocations pending at program termination. However, it is considered bad form, and "memory leaks" during the execution of a program are harmful.

Example Program

- ```
  void main() {
      int *p, *q;
      p = (int *)malloc(sizeof(int));
      q = (int *)malloc(sizeof(int));
      *p = 10; *q = 20;
      *p = *q;
      // p = q;
      printf("%d\n", *p);
  }
  ```


  ```

free( ) function

- free( ) is used to deallocate a block of memory when it is no longer needed.
- The free command does two things:
  1. The block of memory pointed to by the pointer is unreserved and given back to the free memory on the heap. which can then be reused later.
  2. It leaves the pointer in an uninitialized state, and must be reinitialized before it can be used again.

Gont.

Example Program

- ```
  #include <stdio.h>
  int main()
  { int *p;
  p = (int *)malloc (sizeof(int));
  *p=10;
  printf("%d\n",*p);
  free(p);
  return 0;
  }
  ```
  ```

Gont.

This code is really useful only for demonstrating the process of allocating, deallocating, and using a block in C. The **malloc** line allocates a block of memory of the size specified -- in this case, **sizeof(int)** bytes (4 bytes). The **sizeof** command in C returns the size, in bytes, of any type. The code could just as easily have said **malloc(4)**, since **sizeof(int)** equals 4 bytes on most machines. Using **sizeof**, however, makes the code much more portable and readable.

The **malloc** function returns a pointer to the allocated block. This pointer is generic. Using the pointer without typecasting generally produces a type warning from the compiler. The **(int *)** typecast converts the generic pointer returned by malloc into a "pointer to an integer," which is what **p** expects. The free statement in C returns a block to the heap for reuse.

Example Program

```
– struct rec
  { int i; float f; char c;
  };
– int main()
  { struct rec *p;
  p=(struct rec *) malloc (sizeof(struct rec));
  (*p).i=10; (*p).f=3.14; (*p).c='a';
  //*p.i=lO;  *p.f=3.l4;  *p.c='a';
  //p->i=lO; p->f=3.l4; p->c='a';
  printf("%d %f %c\n",(*p).i,(*p).f,(*p).c); free(p);
  return 0;
}
```

Note the following line:

```
(*p).i=10;
```

Many wonder why the following doesn't work:

```
*p.i=10;
```

The answer has to do with the precedence of operators in C. The result of the calculation 5+3*4 is 17, not 32, because the * operator has higher precedence than + in most computer languages. In C, the **.** operator has higher precedence than *, so parentheses force the proper precedence.

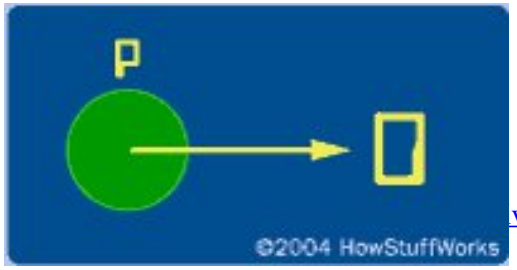
Most people tire of typing **(*p).i** all the time, so C provides a shorthand notation. The following two statements are exactly equivalent, but the second is easier to type:

```
(*p).i=10; p->i=10;
```

You will see the second more often than the first when reading other people's code.

Pointing to zero address

- It physically places a zero into p. The pointer p's address is zero.
- Any pointer can be set to point to zero. When p points to zero, it does not point to a block, but simply contains the address zero, and this value is useful as a tag. It can be used in statements such as:
  - `if (p == 0) { ... }` or: `while (p != 0) { ... }`
- Error messages generate, if you happen to dereference a zero pointer.
- For example, in the following code:
  - `p = 0; *p = 5;`
- The program will normally crash. The pointer p does not point to a block, it points to zero, so a value cannot be assigned to *p.
- zero pointer is used as a flag when using linked lists.



veeti

It physically places a zero into p. The pointer p's address is zero.

Any pointer can be set to point to zero. When p points to zero, however, it does not point to a block. The pointer simply contains the address zero, and this value is useful as a tag. You can use it in statements such as:

```
if (p == 0) { ... }
```

or:

```
while (p != 0) { ... }
```

The system also recognizes the zero value, and will generate error messages if you happen to dereference a zero pointer. For example, in the following code:

```
p = 0; *p = 5;
```

The program will normally crash. The pointer p does not point to a block, it points to zero, so a value cannot be assigned to *p. The zero pointers is used as a flag when using linked lists.

Pointers: Common Bugs

Bug #1 - Uninitialized pointers

- Easiest ways to create a pointer bug is to try to reference the value of a pointer even though the pointer is uninitialized and does not yet point to a valid address. For example:
  - int *p; *p = 12;
- When you say ***p=12;**, the program will simply try to write a 12 to whatever random location **p** points to.
- The program may explode immediately, or may take time to, or it may subtly corrupt data in another part of your program without your notice.
- This can make this error very hard to track down.
- **Make sure you initialize all pointers to a valid address before dereferencing them.**

Gont.

Pointers: Common Bugs

Bug #2 - Invalid Pointer References

- An invalid pointer reference occurs when a pointer's value is referenced even though the pointer doesn't point to a valid block.
- One way to create this error is to say **p=q;** when **q** is uninitialized. The pointer p will then become uninitialized as well, and any reference to *p is an invalid pointer reference.
- The only way to avoid this bug is to draw pictures of each step of the program and make sure that all pointers point somewhere.

- Invalid pointer references cause a program to crash inexplicably for the same reasons given in Bug #1.

Gont.

Pointers: Common Bugs

Bug #3 - Zero Pointer Reference

- A zero pointer reference occurs whenever a pointer pointing to zero is used in a statement that attempts to reference a block.
- For example, if **p** is a pointer to an integer, the following code is invalid:
  - p = 0; *p = 12;
- There is no block pointed to by p.
- Therefore, trying to read or write anything from or to that block is an invalid zero pointer reference.
- There are good, valid reasons to point a pointer to zero.
- Dereferencing such a pointer, however, is invalid.
- All of these bugs are fatal to a program that contains them.
- You must watch your code so that these bugs do not occur & the best way to do that is to draw pictures of the code's execution step by step.

Doubts Please…

Linkedin@KeerthanaGaliveeti

Interview Questions / FAQs

- What's wrong with this code?
  - main() {
    char *p; *p = malloc(10); *p = 'H';
    printf("%c\n", *p); free(p); }
- How about this?
  - char *p = malloc(10);
- In the previous expression, pointer declared is p, not *p.
- When manipulating the pointer itself (for example setting it to make it point somewhere), just name of the pointer is used:
  - p = malloc(10);
- But, when manipulating the pointed-to memory, indirection operator * is used:
  - *p = 'H';
- The 2nd expression is a valid one.

Gont.

Interview Questions / FAQs

- Does *p++ increment p, or what it points to?
- postfix ++ and -- operators have higher precedence than the prefix unary operators. Hence, *p++ is equivalent to *(p++);
- It increments p(i.e., address value) and hence the value at that address is now referred.
- To increment the value pointed to by p, use (*p)++ (or perhaps ++*p, if the evaluation order doesn't matter).

Gont.

Interview Questions / FAQs

- What's wrong with this code? What you expect the last line will print as?
  - …
  - ```
    int array[5], i, *ip;
    for(i = 0; i < 5; i++)
      array[i] = i;
    ip = array;
    printf("%d\n", *(ip + 3 * sizeof(int)));
    ```
  - …
- Pointer arithmetic in C is always automatically scaled by the size of the objects pointed to.
- Inadvertent try to access a nonexistent element past the end of the array is being made (probably array[6] or array[12] depending on int size) .
    ```

Gont.

Interview Questions / FAQs

- Is char a[] is identical to char *a?
- Not at all.
- Arrays are not pointers, though they are closely related and can be used similarly.
- The array declaration char a[6] requests that space for 6 characters be set aside, to be known by the name ``a''. i.e., there is a location named ``a'' at which 6 characters can sit.
- The pointer declaration char *p, on the other hand, requests a place which holds a pointer, to be known by the name ``p''.
- This pointer can point almost anywhere: to any char, or to any contiguous array of chars, or nowhere.
  - char a[] = "hello"; char *p = "world";

Gont.

Interview Questions / FAQs

- If code contains the ``expression'' 5[abcdef] . Is this legal in C?
- Yes, array subscripting is commutative in C.
- From the pointer definition of array subscripting,
  - a[e] is identical to *((a)+(e)),
- For *any* two expressions a and e, as long as one of them is a pointer expression and one is integral, “proof'' looks like this:
  - a[e] = *((a) + (e)) *(by definition)*
    = *((e) + (a)) *(by commutativity of addition)*
    = e[a] *(by definition)*
- Please, do not misinterpret the commutativity as a[i][j] with a[j][i]. Certainly, they are different.

Pointers and Function 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.
- swap (a,b)

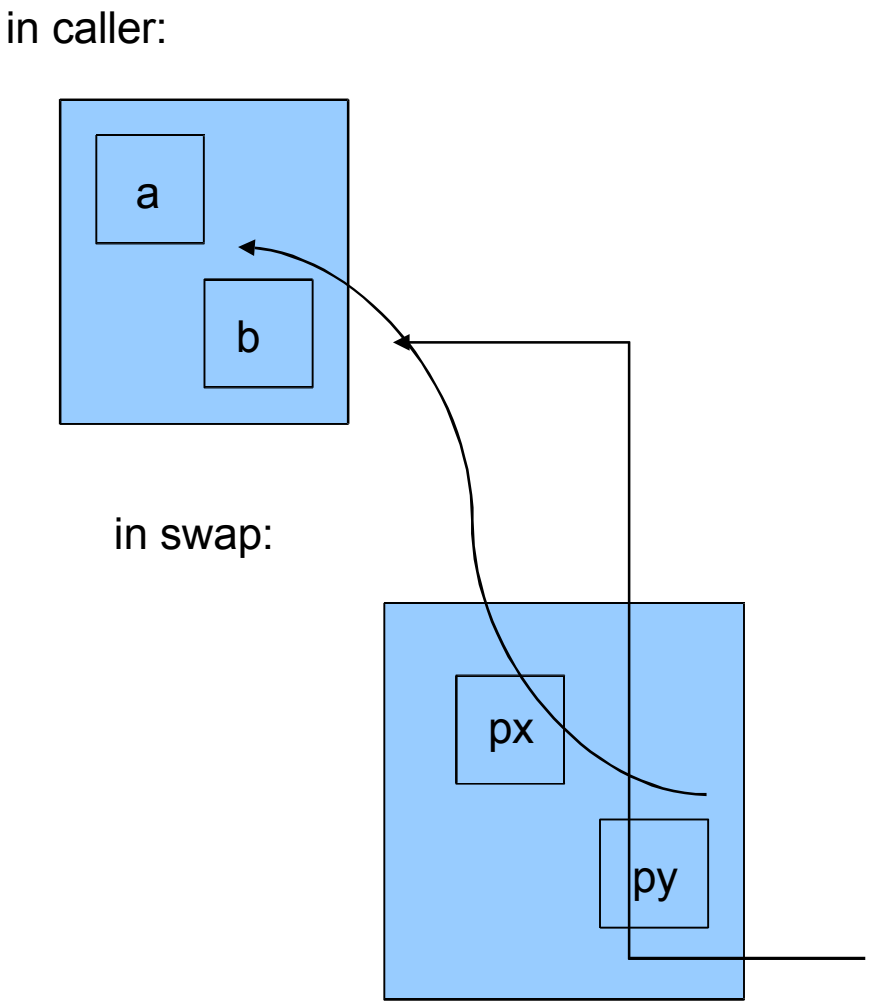
```
void swap ( int x, int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

- THIS IS WRONG. DON'T DO THIS!!

THIS IS CORRECT

- swap(&a,&b);

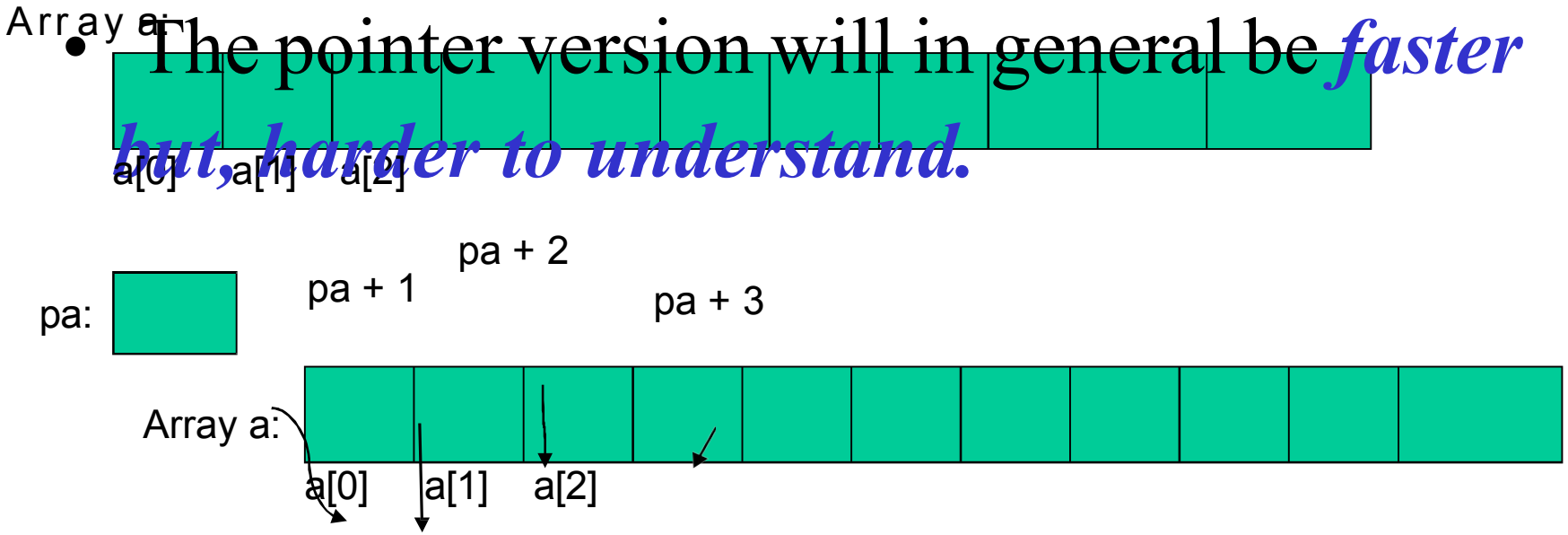
```
void swap(int *px, int
   *py)
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
```



- pointer arguments

Pointers and Arrays

- Any operation that can be achieved by array subscripting can also be done with pointers.
- The pointer version will in general be ***faster but, harder to understand.***



- These remarks are true regardless of type or size of the variables in the array a.
- The meaning of “adding 1 to a pointer” and by extension. all pointer arithmetic, is that pa+1 points to the next object, pa+i points to the i-th object beyond pa.
- a[i] = *(a+i)
- A pointer is a variable, so pa=a and pa++ are legal. But an array name is not a variable; a = pa and a++ are illegal.
- When an array name is passed to a function,what is

passed is the location of the initial element.

Arithmetic

- a pointer can be initialised just as any other variable can, though normally the only meaningful values are zero or an expression involving the addresses of previously defined data of an appropriate type.
- c guarentees that zero is never a valid address for data.
- valid pointer operations are:
  - assignment of pointers of the same type

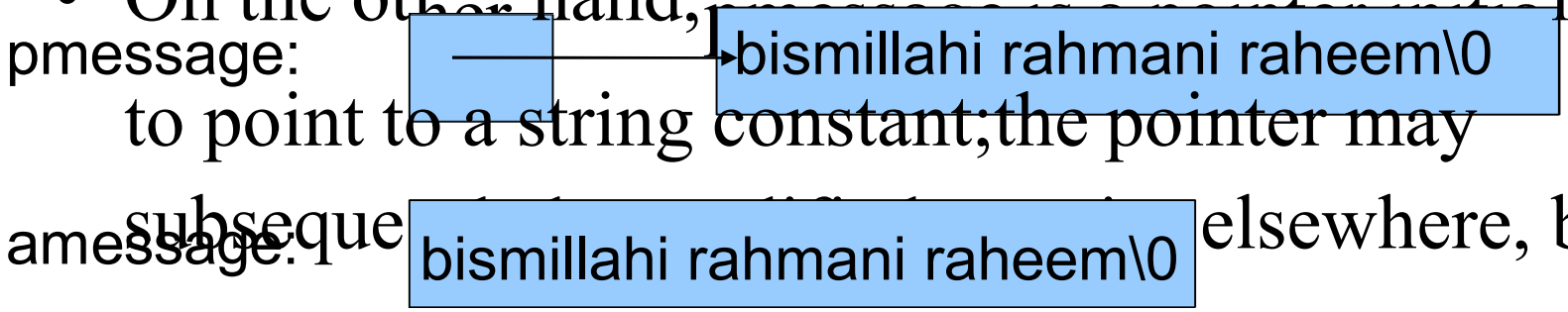
- adding or substracting a pointer and an integer
- substracting or comparing two pointers to members of the same array

String Constant

char amessage[ ] = “bismillahi rahmani raheem”;

char *pmessage = “bismillahi rahmani raheem”;

- Individual characters within the array may be changed but amessage will always refer to the same storage.
- On the other hand, pmessage is a pointer initialised to point to a string constant;the pointer may subsequently be modified to point elsewhere, but



the result is UNDEFINED if you try to modify the string constants.

strcpy

- copy t to s. Array Version

```
void strcpy(char
   *s,char *t)
{
     int i;
     i = 0;
     while((s[i] = t[i])!='\0')
          i++;
```

- copy t to s. Pointer Version

```
}
```

```
void strcpy(char *s,char *t)
{
    i
    n
    t
    i
    ;
    i
    =
    0
    ;
    while((*s = *t)!='\0')
    {
        s++; t++;
    }
}
```

strcpy

- pointer version 2

```
void strcpy(char
   *s,char *t)
{
   while((*s++ =
   *t++)!='\0') ;

}
```

- pointer version 2

```
void strcpy(char
   *s,char *t)
{
   while(*s++ = *t++)
   ;
}
```

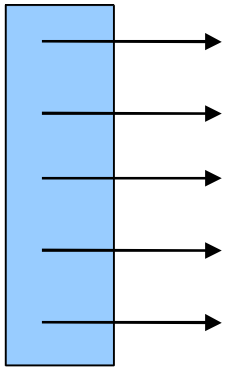
- '\0' has value 0.

Multidimensional Arrays

- int a[10][20];
- int *b[10];
- a is true two dimensional array. 200 int-sized locations have been set aside . 20xrow+col is used to find a[row][col].
- b allocates ten pointers and does not initialize them.
- The important advantage of pointer array is that the rows of the array

- char *name[ ] ={
  “illegal”,”january”,”feb”,”march”,”
  April”};

illega
l\0
janu
ary\0
feb\0
marc
h\0
april\0



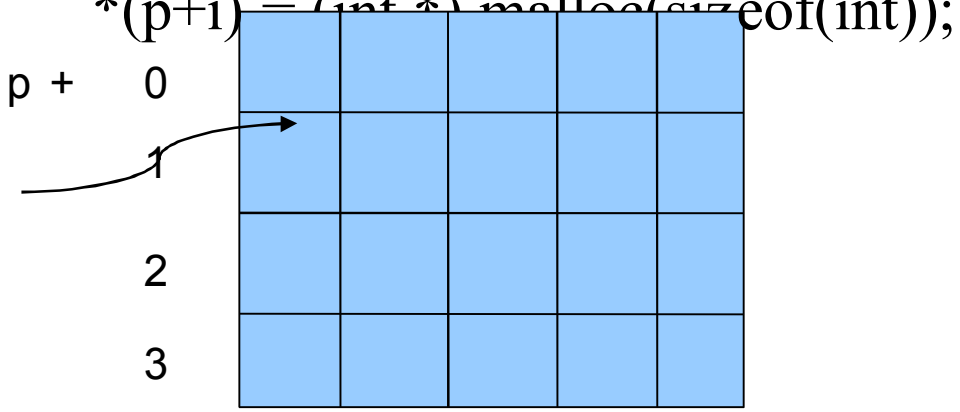
Initializing

- int *p = (int *) malloc(sizeof(int));
- int **p;

p = (int **) malloc(sizeof(int *));

for(i=0; i < what_is_needed; i++)

*(p+i) = (int *) malloc(sizeof(int));



Function Pointers

- function itself is not a variable,but it is possible to define pointers to functions which can be assigned, placed in arrays, passed to functions,returned from functions and so on..
- int (*comp)(void *, void *);

```
comp = add;
(*comp)( 5 , 2 );
```

```
int add (int x,int y)
{
      return x + y;
}

int sub(int x, int y)
{
    return x - y;
}
```

Complicated Declarations

- **char **argv** - argv: pointer to pointer to char
- **int (*daytab)[13]** - daytab: pointer to array 13 of int
- **int *daytab[13]** - daytab: array[13] of pointer to int
- **void *comp()** - comp: function returning void pointer
- **void (*comp)()** - comp : pointer to funciton returning void

- **char (*(*x ( )) [ ]) ( )** - x: function returning

Structures

```
struct person
{
    char name[10];
   int age;
};
struct person ahamed;

ahamed.name = “Ahamed”;
ahamed.age   = 36;
```

- With pointers:

```
struct person *pahamed ;
pahamed = &ahamed;
pahamed->name = “abdullah”;
pahamed->age = 20;
```

```
pahamed = (struct person
     *)malloc(sizeof(struct person);
```

Some C FAQ's

- What's wrong with this declaration? char* p1, p2;
  - char *p1, *p2;
- I'm trying to declare a pointer and allocate some space for it, but it's not working. What's wrong with this code? char *p; *p = malloc(10);
  - The pointer you declared is p, not *p.

- What's the difference between

- int * comp(void *, void *)
- int (*comp)(void*, void *)
- First is a function that takes two void * as arguments and returns a int *
- Second, comp is a function pointer that takes two void * as arguments and return an int.
- USES :
  - in the operating system – kernel structures.
  - to implement call back mechanisms

– to implement state machines.

- Swapping two variables
  - a = a + b; b = a – b; a = a- b;
  - a^=b^=a^=b;
- Does *p++ increment p, or what it points to?
  - The postfix ++ and -- operators essentially have higher precedence than the prefix unary operators. Therefore, *p++ is equivalent to *(p++); it increments p, and returns the value which p pointed to before p was incremented. To increment the value pointed to by p, use (*p)++

- I have a char * pointer that happens to point to some ints, and I want to step it over them. Why doesn't ((int *)p)++; work ?
  - In C, a cast operator does not mean "*pretend these bits have a different type, and treat them accordingly*"; it is a conversion operator, and by definition it yields an rvalue, which cannot be assigned to, or incremented with ++. Say what you mean: use p = (char *)((int *)p + 1); or (since p is a char *) simply p += sizeof(int);
- I have a function which accepts, and is supposed to initialize, a pointer: void f(int *ip) { static int dummy = 5; ip = &dummy; } But when I call it like this: int *ip; f(ip); the pointer in the caller remains unchanged.
  - arguments in C are passed by value. In the code above, the called function alters only the passed copy of the pointer. To make it work as you expect, one fix is to pass the address of the pointer **void f(int **ipp) { static int dummy =**

5; *ipp = &dummy; } ... int *ip; f(&ip); Another solution is to have the function return the pointer: **int *f() { static int dummy = 5; return &dummy;**

What will print out?

```
main()
    {
        char *p1=“name”;
        char *p2;
        p2=(char*)malloc(20);
        memset (p2, 0, 20);
        while(*p2++ = *p1++);
        printf(“%s\n”,p2);
}
```

```
main()
{
      char *p1="name";
      char *p2,*p3;
      p2=(char*)malloc(20);
      p3 = p2;
      memset (p2, 0, 20);

            while(*p2++ = *p1++);
       printf("%s\n",p3);
}
```

OUTPUT ?

- main()
  {
    int x=20,y=35;
    x=y++ + x++;
    y= ++y + ++x;
    printf("%d%d\n",x,y);
- }
- 5794

- What's wrong with the call fopen(”c:\newdir\file.dat”, “r”)?
- print a semicolon using Cprogram without using a semicolon any where in the C code in ur program!!!!
- solution : void main()
  {
  if(printf(”;”))
  { }
  }
- what is the difference between #include and #include”filename”?
- *Give a one-line C expression to test whether a number is a power of 2.*
- Solution 1: int isPower = x>1 && !(x&x-1)

- Solution 2 : printf(((x&1)==0)?”TRUE”:”FALSE”);

In c a pointer is a variable that points to or references a memory location in which data is stored.

Each memory cell in the computer has an address that can be used to access that location so a pointer variable points to a memory location we can access and change the contents of this memory location via the pointer.

A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups.

A pointer is a group of cells that can hold an address.

Recap

- int *ip => says that the expression *ip is of type int.
- void * - is used to hold any type of pointer but can't be dereferenced by itself.
- int x = 10,*ip = &x;

  ++*ip;

  (*ip)++;
- The parentheses is necessary for last one. Without them the expression would increment ip instead of what it points to, because unary operators like * and

++ associate from **right to left**.

Recap

- The unary operator **&** gives the address of an object.
- **p = &c**
- p is now said to "point to" c
- The & operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions,constants or register variables.
- * is indirection or dereferencing operator. when applied to a pointer,it accesses the object the pointer points to.
- int x = 1, y = 2, z[10];

  int *ip;

```
ip = &x;        y = *ip;          *ip = 0;        ip = &z[0];
```

Pointers in C

Linkedin@KeerthanaGaliveeti

Pre-requisite

- Basics of the C programming language
  - Data type
  - Variable
  - Array
  - Function call
  - Standard Input/Output
    - e.g. **`printf(), scanf()`**

Outline

- Computer Memory Structure
- Addressing Concept
- Introduction to Pointer
- Pointer Manipulation
- Summary

Computer Memory Revisited

- Computers store data in memory slots
- Each slot has an *unique address*
- Variables store their values like this:

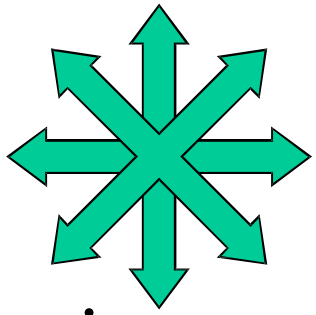
Addr	**Content**	**Addr**	**Content**	**Addr**	**Content**	**Addr**	**Content**
1000	i: 37	1001	j: 46	1002	k: 58	1003	m: 74
1004	a[0]: ‘a'	1005	a[1]: ‘b'	1006	a[2]: ‘c'	1007	a[3]: ‘\0'
1008	ptr: 1001	1009	…	1010		1011	

Computer Memory Revisited

- Altering the value of a variable is indeed changing the content of the memory
  - e.g. i = 40; a[2] = ‘z';

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1001	j: 46	1002	k: 58	1003	m: 74
1004	a[0]: ‘a'	1005	a[1]: ‘b'	1006	a[2]: ‘z'	1007	a[3]: ‘\0'
1008	ptr: 1001	1009	…	1010		1011	

Addressing Concept

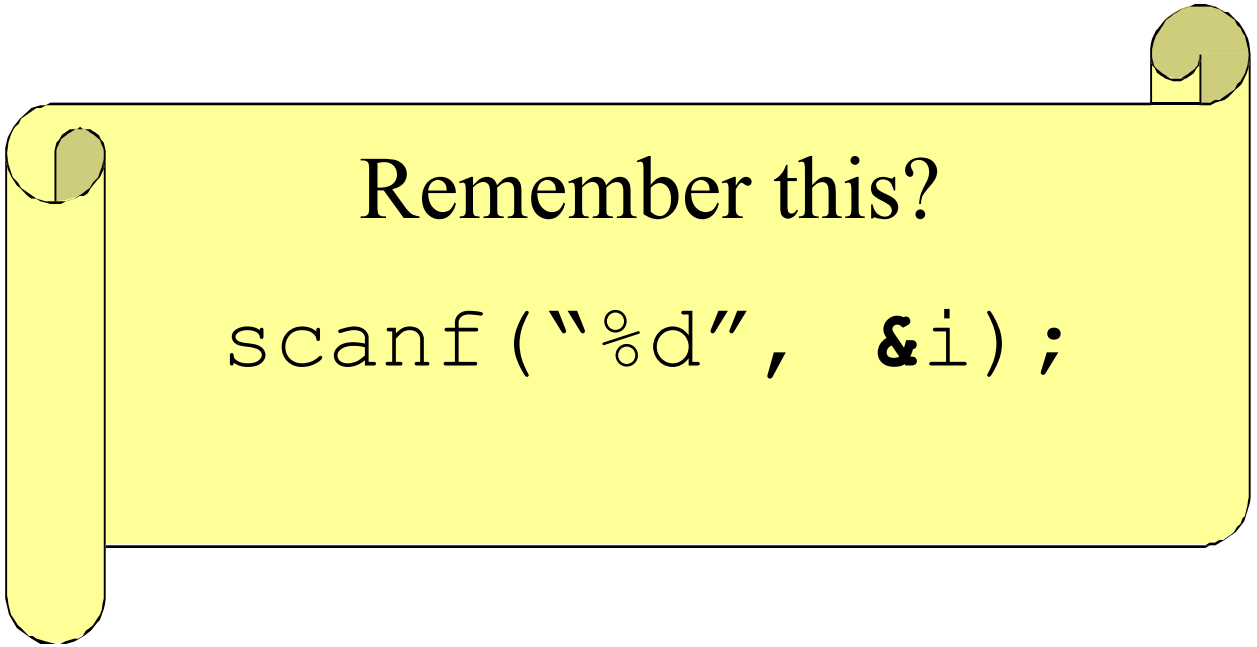


- Pointer stores the **address** of another entity
- It **refers** to a memory location

```
int i = 5;
int *ptr;                          /* declare a pointer variable  */
ptr = &i;                          /* store address-of i to ptr */
printf("*ptr = %d\n", *ptr);       /* refer to referee of ptr */
```

Why do we need Pointer?

- Simply because it's there!
- It is used in some circumstances in C



Twin Operators

- &: Address-of operator
  - Get the *address* of an entity
    - e.g. **`ptr = &j;`**

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1001	j: 33	1002	k: 58	1003	m: 74
1004	ptr: 1001	1005		1006		1007	

Twin Operators

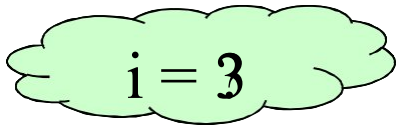
- *: De-reference operator
  - Refer to the *content* of the referee
    - e.g. **`*ptr = 99;`**

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1001	j: 99	1002	k: 58	1003	m: 74
1004	ptr: 1001	1005		1006		1007	

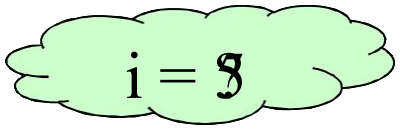
Example: Pass by Reference

- Modify behaviour in argument passing

```
void f(int j)
{
  j = 5;
}
void g()
{
  int i = 3;
  f(i);
}
```




```
void f(int *ptr)
{
  *ptr = 5;
}
void g()
{
  int i = 3;
  f(&i);
}
```



An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10

An Illustration

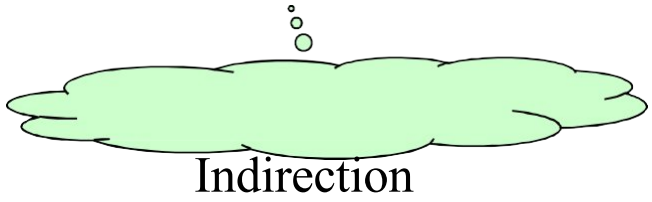
```
int i = 5, j = 10;
int *ptr;     /* declare a pointer-to-integer variable */
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	

An Illustration

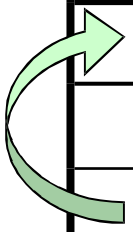
```
int i = 5, j = 10;
int *ptr;
int **pptr;  /* declare a pointer-to-pointer-to-integer variable */
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	
pptr	int **	integer pointer pointer variable	
		Double	


Indirection

An Illustration

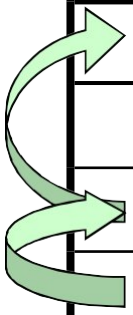
```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;     /* store address-of i to ptr */
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	
*ptr	int	de-reference of ptr	5

An Illustration

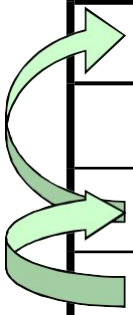
```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr; /* store address-of ptr to pptr */
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*pptr	int *	de-reference of pptr	value of ptr (address of i)

An Illustration

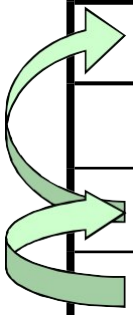
```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	3
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	3

An Illustration

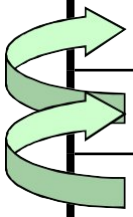
```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
**pptr	int	de-reference of de-reference of pptr	7

An Illustration

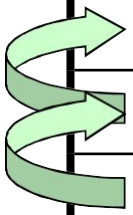
```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	10
ptr	int *	integer pointer variable	address of j
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	10

An Illustration

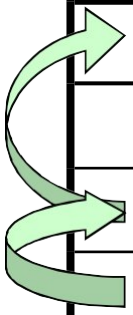
```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	9
ptr	int *	integer pointer variable	address of j
pptr	int **	integer pointer pointer variable	address of ptr
**pptr	int	de-reference of de-reference of pptr	9

An Illustration

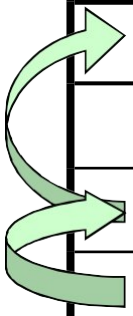
```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	9
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*pptr	int *	de-reference of pptr	value of ptr (address of i)

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	-2
j	int	integer variable	9
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	-2

Pointer Arithmetic

- What's `ptr + 1`?
  - The next memory location!
- What's `ptr - 1`?
  - The previous memory location!
- What's `ptr * 2` and `ptr / 2`?
  - Invalid operations!!!

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	?
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	?
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	3.14

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	address of a[3]
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array

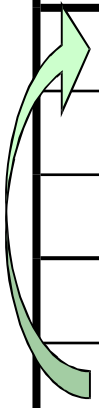
```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[3]
*ptr	float	de-reference of float pointer variable	9.0

Linkedin@KeerthanaGaliveeti

Pointer Arithmetic and Array

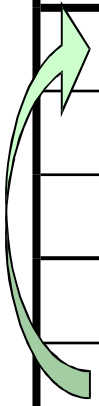
```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```



Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[0]
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```



Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	6.0
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[0]
*ptr	float	de-reference of float pointer variable	6.0

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	6.0
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	3.14

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	6.0
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	7.0
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	7.0

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

- Type of `a` is `float *`
- `a[2]` □□ `*(a + 2)`
  - `ptr = &(a[2])`
  - □ `ptr = &(*(a + 2))`
  - □ `ptr = a + 2`
- `a` is a memory address *constant*
- `ptr` is a pointer *variable*

More Pointer Arithmetic

- What if `a` is a `double` array?
- A `double` *may* occupy more memory slots!
  - Given **`double *ptr = a;`**
  - What's **`ptr + 1`** then?

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	a[0]: 37.9	1001	…	1002	…	1003	…
1004	a[1]: 1.23	1005	…	1006	…	1007	…

1008	a[2]: 3.14	1009	…	1010	…	1011	…

More Pointer Arithmetic

- Arithmetic operators + and – *auto-adjust* the address offset
- According to the *type* of the pointer:
  - 1000 + sizeof(double) = 1000 + 4 = 1004

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	a[0]: 37.9	1001	…	1002	…	1003	…
1004	a[1]: 1.23	1005	…	1006	…	1007	…

1008	a[2]: 3.14	1009	…	1010	…	1011	…

Advice and Precaution

- Pros
  - Efficiency
  - Convenience
- Cons
  - Error-prone
  - Difficult to debug

Summary

- A pointer stores the **address** (memory location) of another entity
- Address-of operator (&) **gets the address** of an entity
- De-reference operator (*) **makes a reference** to the referee of a pointer
- Pointer and array
- Pointer arithmetic