Pointers

Rule1 - Pointer is an integer

- Pointer is a variable that can store an address
- The data present in the pointer will be numeric data and is an address.
- Syntax to declare the pointer is:
  datatype * pointer_name;
- Type of the pointer is to store the address of that kind of variable.

Eg: An integer pointer is used to store the address of an integer variable and a character pointer is used to store the address of a character variable etc.

- Size of the pointer depends on the bitness of the system. It is 4 bytes in a 32 bit system and 8 bytes in a 64 bit system. This is because a variable can get the memory allocated anywhere from the memory block. The address varies from 0x00 to 0xFFFFFFFF in a 32 bit system and 0x00 to 0xFFFFFFFFFFFFFFFF in a 64 bit system. The pointer should be able to store any address in it.

Eg:

```
int main()
{
    int *ptr; //integer pointer meant to hold the address of an integer
variable
    int num;
}
```


```
int main()
    {
        int *ptr;
        int num;
        num = 10;
        ptr = 10;
}
```

- Both num and ptr are storing 10. The difference is 10 in num is a normal integer data and 10 in ptr is treated as an address.

Rule 2 - Referencing and dereferencing.

- Pointers have 2 operators namely, referencing and dereferencing operator
- Referencing operator is used to get the address of a variable. It is represented by “&”.
  Eg: &ptr is used to get the address of the ptr variable
- Dereferencing operator is used to get the value from an address. This needs to be applied only on pointer variables. This operator is represented by “*”.
  Eg:

```
int main()
{
        int num = 10;
        int *ptr;
        ptr = &num;
        printf(“%d\n”, *ptr);
        return 0;
}
```

- In the above example, num has 10 at address 1000. &num should give 1000 which is stored in ptr. As ptr is having 1000 stored, doing *ptr means *1000 i.e go to address 1000 and fetch the value.
  Eg2:

```
int main()
{
        int num = 10;
        int *ptr;
        ptr = &num;
        printf(“%d\n”, *ptr);
        *ptr = 20;
        printf(“%d\n”, num);
        return 0;
}
```

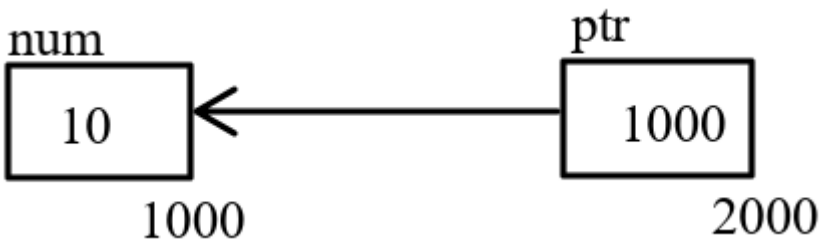
- When *ptr = 20 is done, it means *1000 = 20 i.e go to address 1000 and write 20. This inturn changes the value present in num because it is present in address 1000.

Rule 3 - Pointing means containing

- Whenever there is a pointer variable,it has an address stored inside it. And the pointer will be pointing to that address.

Eg:

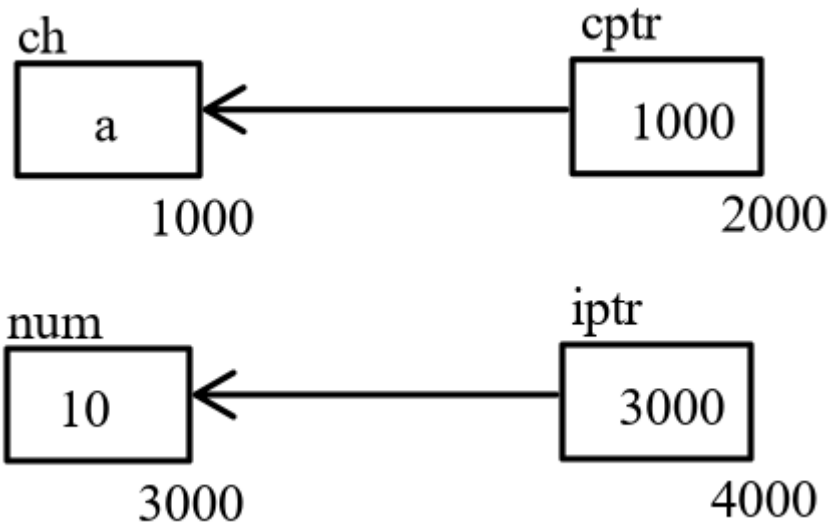
```
int main()
{
        Int num = 10;
        int *ptr;
        ptr = &num;
        return 0;
}
```



Rule 4: Pointer type

- When all the pointers are getting the same bytes of memory allocated, why do we need a type to be specified?
- Let us consider an example. Assume there is a character pointer and an integer pointer.

Eg:
```
int main()
{
        char ch = ‘a’;
        int num = 10;
        char *cptr = &ch;// similar to char *cptr; cptr = &ch;
        int *iptr = &num; // similar to int *iptr; iptr = &num;
        return 0;
}
```



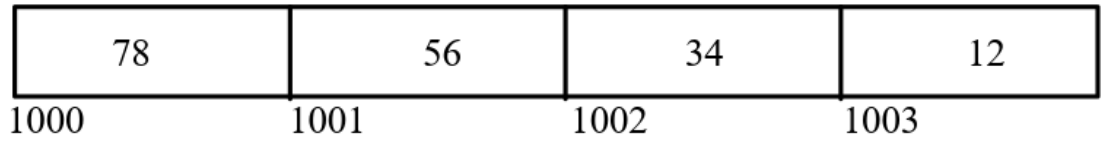
- Generally, can we say that the address of a character variable is 1 byte and the address of integer variable is 4 bytes? No because the size of the address depends on the bitness of the system
- Looking just into the address, can we say what kind of data is present in that address? No.
- Now whenever we dereference the pointer, how many bytes of data does it fetch or write from the given address?
- It all depends on the type of the pointer.

- An integer pointer will fetch sizeof(int) number of bytes from the given address.A character pointer will fetch sizeof(char) number of bytes.
- Hence type of the pointer will say the pointer to fetch sizeof(datatype) number of bytes.

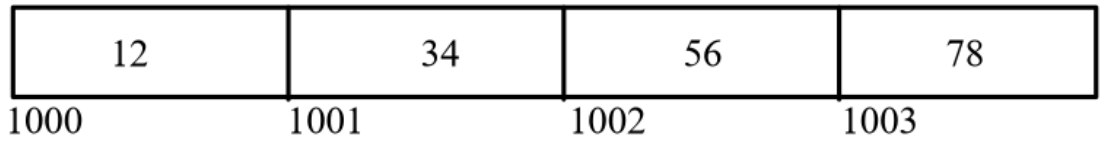
Rule 4 in detail: Endianness of a system

- Byte ordering of the data in the system is called endianness. There are two ways how bytes of data are stored in the system. They are little and big endian systems.
- In the little endian system,the least significant byte is stored in the lowest address and in the big endian system, the most significant byte is stored in the lowest address.
  Eg: int num = 0x12345678;

Little Endian:



Big Endian:



Eg:
```
int main()
{
        int num = 0x12345678;
        int *iptr = &num;
        char *cptr = &num;
        *cptr = 0xAB;
        printf("%x", *iptr);
        return 0;
}
```

- In the above example, when char *cptr = &num is done, it throws a warning as we are trying to store the address of an integer. This can be avoided by typecasting.
- printf gives 123456AB as output in little endian system and AB345678 in big endian system. This is because, when *cptr = 0xAB is done, cptr is pointing to

address 1000. Hence it changes the data in address 1000. When the data is fetched after that, in little endian system as the least significant byte is changed, we get the output as 0x123456AB and in big endian system as the most significant bit is changed, we get output as 0xAB345678.

Rule 5: Pointer arithmetic

- Type of the pointer is also necessary to perform pointer arithmetic.
- Pointer arithmetic always depends on sizeof(pointer_type)
- Assume an integer pointer ptr. When pointer arithmetic is performed, it depends on sizeof(int).

  Eg:
  ```
  int num;
  int *ptr = &num;
  ptr++;
  ```
- If ptr is having address 1000, when ptr++ is done, it gets changed to 1004 as shown below

  ptr++ is ptr = ptr + 1;

  Pointer arithmetic depends on sizeof(type of the pointer). Hence,

  ptr = ptr + 1 * sizeof(int)

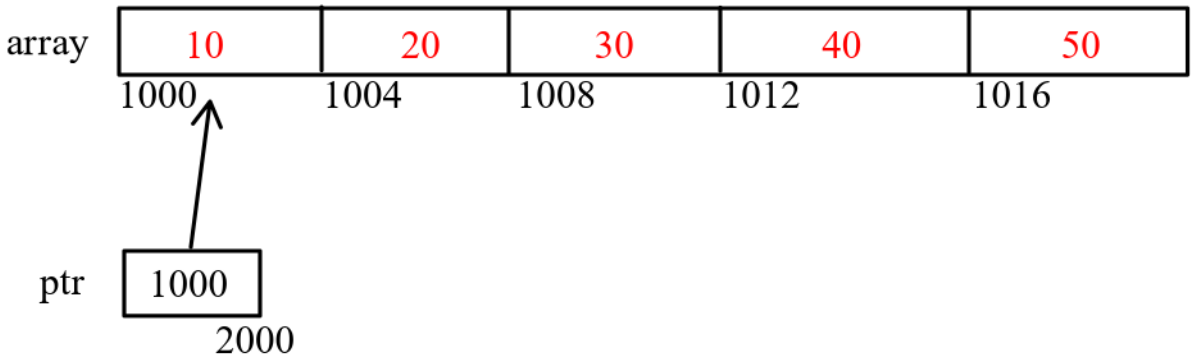
  ptr = 1000 + 1 * 4

  ptr = 1004.

Rule 5 in detail: Passing an array to a function

- Name of the array interprets 2 things.
- First, name of the array is interpreted as the whole address of the array when used in sizeof() operator and when used after & operator.
- Rest all other cases i.e when it is being assigned to a pointer or passed as an argument to another function, it is interpreted as the base address of the array.

  Eg:
  ```
  int main()
  {
      int array[5] = {10, 20, 30, 40, 50};
      int *ptr = array;
      printf("%d\n", *ptr);
      ptr++;
      printf("%d\n", *(ptr + 2));
      return 0;
  }
  ```



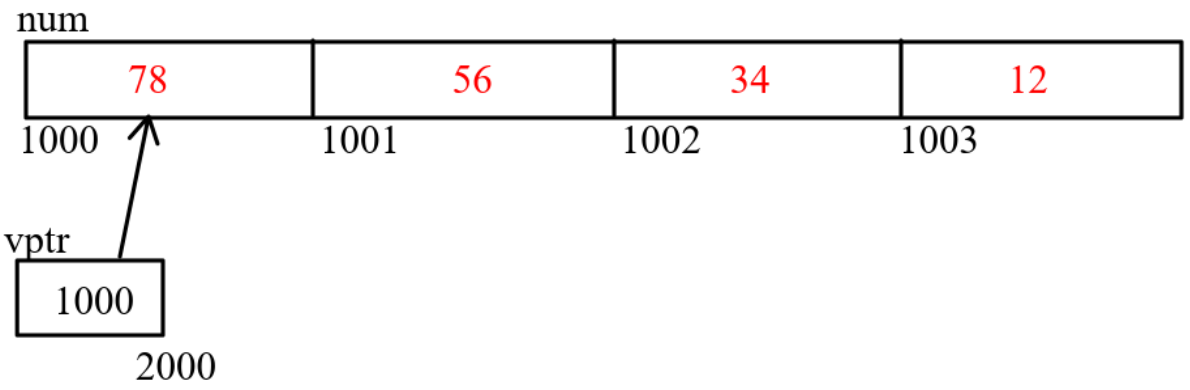
- printf("%d\n", *ptr); *ptr is *1000. This gives 10 as output.
- ptr++ is ptr = ptr + 1 ;
  ptr = ptr + 1 * sizeof(type of the pointer);
  ptr = 1000 + 1 * sizeof(int);
  ptr = 1004
- *(ptr + 2) is *(ptr + 2 * sizeof(type of the pointer))
  *(ptr + 2 * sizeof(int))
  *(1004 + 2 * 4)
  *1012
  Hence printf("%d\n", *(ptr + 2)); gives 40 as it has to go to address 1012 and fetch an integer.
- Different ways of printing array's elements are,
  - ➢ array[i]; This is interpreted by the compiler as *(array +i). Both array and ptr represent the base address. array represents address 1000. Hence array + i is also pointer arithmetic and depends on sizeof(type of the array).
  - ➢ i[array]; // *(array + i) is same as *(i + array)
  - ➢ ptr[i] ; // same as *(ptr + i)
  - ➢ i[ptr];

Rule 6: Pointing to nothing

- If there is an uninitialized pointer variable, it can be pointing to some random address. When we later try dereferencing, it might give some garbage value or it might also lead to segmentation fault.
- Instead of making the pointer fail silently, good practice is to have a defined behaviour.
- Whenever the pointer is declared, it can be initialised with a null pointer constant called NULL.
- NULL might represent address 0 (actually (void *)0) or other address depending on the operating system. The pointers which have NULL stored in them are called Null pointers. These pointers always give a fixed result when dereferenced i.e segmentation fault.
  Eg: int main()

```
{
        int *ptr = NULL;
        printf("%d\n", *ptr); // gives segmentation fault
}
```

- Uses of Null pointer are to indicate the linked list termination, failure of malloc calloc etc.
- There are generic pointers which can hold the address of any variable. THey are called void pointers.
- The syntax to declare void pointer is,
  void *vptr; // vptr is the name of the pointer
- Void pointers cannot be dereferenced. Because when *vptr is said, compiler will not know what kind of data should be fetched. Hence typecasting before dereferencing is necessary.
- *(int *)vptr is a way of typecasting. This means whatever address vptr is holding should be treated as an address of integer and then dereferenced. Now this will dereference 4 bytes of data and gives an integer
- Pointer arithmetic on void pointer is compiler implementation dependent. This is allowed in gcc because sizeof(void) is 1.
  Eg:
```
int main()
{
        int num = 0x12345678;
        void *vptr = &num;
        printf("%x\n", *(int *)vptr);
        printf("%hhx\n", *(char *)(vptr + 1));
        printf("%hx\n", *((short *)vptr + 1));
        return 0;
}
```



➢ *(int *)vptr
  *(int *)1000 // treat address 1000 as address of integer and dereference
  Hence %x prints 12345678

➢ *(char *)(vptr + 1)
  *(char *)(vptr + 1 * sizeof(void)) // because pointer arithmetic is applied on vptr

*(char *)(1000 + 1 * 1)
*(char *)(1001)
Address 1001 is treated as address of the character and 1 byte is fetched. %x is used to print the hexadecimal equivalent of data in 4 bytes. %hx is used to print the hexadecimal equivalent of the data in 2 bytes and %hhx to print the hexadecimal equivalent from 1 byte of data.
Hence this gives output as 56

- *((short *)vptr + 1)
  *((short *)vptr + 1 * sizeof(short)) // because typecasting is done first then pointer arithmetic
  *((short *) 1000 + 1 * 2)
  *((short *)1002)
  Output of third printf is 1234 (because of the little endian system)

Rule 7 : Dynamic memory allocation