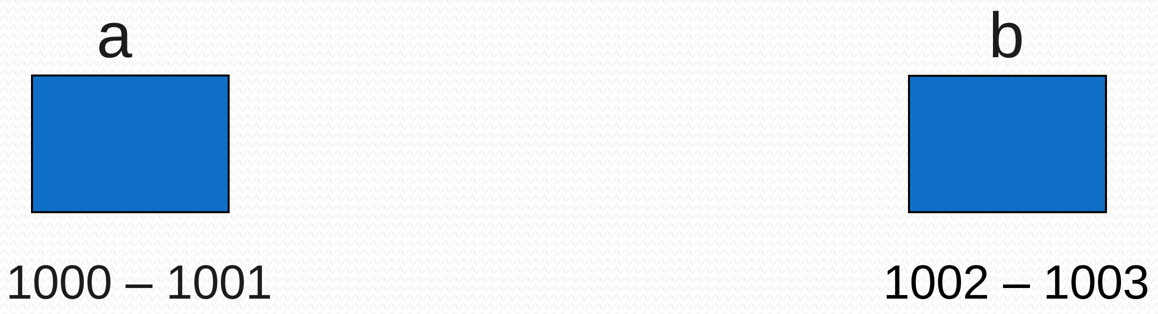
Pointers in ‘C’-Module 3

Pointers

int a=1,b;



b = a; // b=1;

b = &a; X   &a = 1000

→Address can only be stored in pointer variable.

Pointers

Definition:

→A variable which is capable of holding address of another variable is said to be a pointer variable.

→Using pointer variable it is easier for accessing memory location of another variable.

Pointers - Declaration

Syntax:

Data Type * variable;

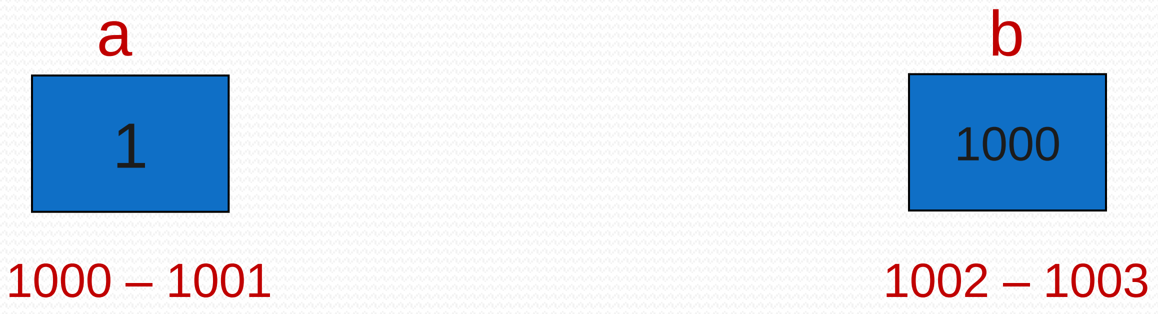
Where

data type → Type of variable whose address is to be stored in pointer variable.

* → Value at address operator or pointer operator.

Pointers

int a=1;
int *b; // b is a pointer variable



b = &a; &a = 1000 ; b=1000
b is pointing to address location 1000
***b = * (1000) = 1**

Using pointers its possible to access value of another variable by storing its address.

Pointers

To access the value stored in the address we use the unary operator (*) that returns the value of the variable located at the address specified by its operand.

```
// C program to demonstrate use of * for
pointers in C
#include <stdio.h>
 int main()
{
   int Var = 10;
       int *ptr = &Var;
       printf("Value of Var = %d\n", *ptr);
       printf("Address of Var = %p\n", ptr);
       *ptr = 20; // Value at address is now 20
       printf("After doing *ptr = 20, *ptr is %d\n",
*ptr);
    return 0;
}
```

Output :

```
Value of Var = 10
Address of Var = 0x7fffa057dd4
After doing *ptr = 20, *ptr is 20
```

Pointer Expressions and Pointer Arithmetic

A limited set of arithmetic operations can be performed on pointers. A pointer may be:

•incremented (++)

•decremented (—)

•an integer may be added to a pointer (+ or +=)

•an integer may be subtracted from a pointer (– or -=)

(Note: Pointer arithmetic is meaningless unless performed on an array.)

```
// C program to illustrate Pointer
Arithmetic
#include <bits/stdc++.h>

int main()
{

   int v[3] = {10, 100, 200};

   int *ptr;

   ptr = v;

   for (int i = 0; i < 3; i++)
   {
      printf("Value of *ptr = %d\n", *ptr);
      printf("Value of ptr = %p\n\n",
ptr);
      ptr++;
```

```
Output:Value of *ptr = 10
Value of ptr = 0x7ffcae30c710
Value of *ptr = 100
Value of ptr = 0x7ffcae30c714
Value of *ptr = 200
Value of ptr = 0x7ffcae30c718
```

Behavior of sizeof operator

```
// 1st program to show that array and pointers are
different
#include <stdio.h>
int main()
{
  int arr[] = {10, 20, 30, 40, 50, 60};
  int *ptr = arr;

  // sizof(int) * (number of element in arr[]) is printed
  printf("Size of arr[] %d\n", sizeof(arr));

  // sizeof a pointer is printed which is same for all type
  // of pointers (char *, void *, etc)
  printf("Size of ptr %d", sizeof(ptr));
  return 0;
}
```

Output:

```
Size of arr[] 24
Size of ptr 4
```

Array members are accessed using pointer arithmetic.

Compiler uses pointer arithmetic to access array element. For example, an expression like “arr[i]” is treated as *(arr + i) by the compiler. That is why the expressions like *(arr + i) work for array arr, and expressions like ptr[i] also work for pointer ptr.

```
#include <stdio.h>
int main()
{   int arr[] = {10, 20, 30, 40, 50, 60};
  int *ptr = arr;
  printf("arr[2] = %d\n", arr[2]);
  printf("*(arr + 2) = %d\n", *(arr + 2));
  printf("ptr[2] = %d\n", ptr[2]);
  printf("*(ptr + 2) = %d\n", *(ptr + 2));
  return 0;
}
```

Output:

```
arr[2] = 30
*(arr + 2) = 30
ptr[2] = 30
*(ptr + 2) = 30
```

Array parameters are always passed as pointers, even when we use square brackets.

```
int fun(int ptr[])
{
   int x = 10;
   // size of a pointer is printed
   printf("sizeof(ptr) = %d\n", sizeof(ptr));
   // This allowed because ptr is a pointer, not array
   ptr = &x;

   printf("*ptr = %d ", *ptr);

   return 0;
}

int main()
{
   int arr[] = {10, 20, 30, 40, 50, 60};
   fun(arr);
   return 0;
}
```

Output:

```
sizeof(ptr) = 4

*ptr = 10
```

Function Pointer in C

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
   printf("Value of a is %d\n", a);
}

int main()
{
   // fun_ptr is a pointer to function fun()
   void (*fun_ptr)(int) = &fun;

   /* The above line is equivalent of following two
     void (*fun_ptr)(int);
     fun_ptr = &fun;
   */

   // Invoking fun() using fun_ptr
   (*fun_ptr)(10);

   return 0;
}
```

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

Output:

```
Value of a is 10
```

Another Example

```
#include <stdio.h>
#include <string.h>

void check(char *a, char *b,
           int (*cmp)(const char *, const char *));

int main(void)
{
  char s1[80], s2[80];
  int (*p)(const char *, const char *);

  p = strcmp;

  gets(s1);
  gets(s2);

  check(s1, s2, p);

  return 0;
}

void check(char *a, char *b,
           int (*cmp)(const char *, const char *))
{
  printf("Testing for equality.\n");
  if(!(*cmp)(a, b)) printf("Equal");
  else printf("Not Equal");
}
```

Function Pointer as switch case in C

- Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.
- Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.

Enter Choice: 0 for add, 1 for subtract and 2 for multiply

Function Pointer as switch case in C

```
#include <stdio.h>
void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}
```

```
int main()
{
// fun_ptr_arr is an array of function
pointers
void (*fun_ptr_arr[])(int, int) = {add,
subtract, multiply};
unsigned int ch, a = 15, b = 10;
printf("Enter Choice: 0 for add, 1 for subtract
and 2 for multiply\n");
scanf("%d", &ch);
if (ch > 2) return 0;
(*fun_ptr_arr[ch])(a, b);
return 0;
```

Dynamic Memory Allocation

Statistics of Rainfall

The weather station of each city has the detail of rainfall in a year. Given the date and cm of rainfall recorded on that day, write a C program to determine the rainfall recorded in each month of the year and average monthly rainfall in the year.

Input	Output	Logic Involved
Details of rainfall recorded in the year (Date and cm of rainfall)	Rainfall recorded for each month and monthly average	Find sum of values for each month and then determine monthly average

Rainfall problem

Input	Output	Logic Involved
Details of rainfall recorded in the year (Date and cm of rainfall)	Rainfall recorded for each month and monthly average	Find sum of values for each month and then determine monthly average

Implementation in C

- For storing each row of the table we need a structure and we may use array of structures for storing details
- Memory allocation for arrays – Statically done
- But number of days varies from city to city…
- Cherrapunji has 170 rainy days on average
- Rajasthan gets least rainfall and receives only rainfall for 30 days in a year.

Static vs Dynamic Memory Allocation

- If we declare an array of size 180 then for a city that has less rainfall wastage of memory
- If the programmer declares an array of size 50 then there will be shortage of memory.
- It is better to get the number of raining days from the user during execution of program and allocate memory
- C supports dynamic memory allocation
- Allocates a chunk of memory and returns the address of first byte

Allocation of Memory

- *Static Allocation*: Allocation of memory space when execution begins.
- *Dynamic Allocation*: Allocation of memory space at run time.

malloc()

- Dynamically allocates memory when required
- This function allocates ‘size’ byte of memory and returns a pointer to the first byte or NULL if there is some kind of error
- Syntax is as follows:

  void * malloc (size_t size);
- return type is of type void *, also receive the address of any type ;

```
  char *p;
  p = malloc(1000); /* get 1000 bytes */
```

calloc()

- Used to allocate storage to a variable while the program is running

Syntax

- void * calloc (size_t n, size_t size);
- For example, an int array of 10 elements can be allocated as follows.
- int * array = (int *) calloc (10, sizeof (int));

Difference between malloc() and calloc()

- Calloc allocates multiple blocks of data whereas malloc allocates as a single block
- Calloc initializes all bytes in the allocation block to zero
- Allocated memory may/may not be contiguous.

realloc()

- Modifies the allocated memory size by malloc () and calloc () functions to new size
- If enough space doesn’t exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees old block
- Syntax

```
void * realloc (void * ptr, size_t size);
```

free()

- frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system
- Stdlib.h is the header file that contains these functions malloc() , Realloc, free()