C LANGUAGE

**Introduction**

1. Developed at Bell Laboratories, used in development of UNIX operating systems.
2. High level language, fast and efficient(because directly connected to memory)
3. Highly portable( because hardware or platform independent, can run on any machine which supports C)
4. Every C statement ends with a semicolon, function name always followed by a ( ) brackets.
5. { } – meant start and end of a block or module.
6. Structure
7. Comment – multi lines --/\* \*/, single line -- //
8. Pre-processor controls (it takes include file or macro – always starts with #) (ex: #include )
9. Global data definitions – variables that can be used in main program and program functions.
10. Function definitions – main( ). This contains both data definitions and instructions to be executed.

**Variables**

1. Declare variables before they are used.
2. There will be different data types while initializing a variable i.e. int(2 bytes), char(1), long int(4) = long, float(4), long float(8) = double. This is according to standard ANSI C.
3. While declaring variables –
4. Name can consists letters,digits, \_. Must start name with a letter. It is case is sensitive. Don’t use reserved words(32) for variable declarations.
5. Declaration of variables can be outside or inside function. If it is outside main( ) function -- global variable. If it is inside function – local variable.

**Constants**

**STORAGE CLASSIFIERS for Variable (defines scope and lifetime of a variable)**

Register – Requests to compiler to store in registers of the microprocessor.

Auto – variable getting automatically allocated and deallocated when function is invoked. Auto variables can be only accessed within the block/function they have been declared and not outside them.

Static - Static variables have the property of preserving their value even after they are out of their scope. They are initialized only once and exist till the termination of the program

Extern – this variable defined somewhere else, so an extern variable is nothing but a global variable initialized.

**COMMAND LINE ARGUMENTS**

main (int argc, char \*argv[ ]) argc - number of arguments , argv - which are the arguments.

While compiling – gcc -Wall -g <filename>

Wall -warning all

g -debugging, when supplying to client, don’t send with -g.

./app 1 2 3 4 5 6 – Number of arguments = 7(including name of the application.)

**FUNCTIONS**

**Local variables**: variables defined inside a function. They are created each time the function is called.

**Lifetime:** Period of time when memory location is allocated.

**Scope:** Region of program text where declaration is visible.

**Arguments passed by value** (it copies the content) – values of arguments given to called function in temporary variables. The modifications to the parameter variables do not effect the variables in calling function.

**call by reference** – (copies the address of variables)

There are two kinds of arguments : **formal arguments**(used inside the function body to refer to its argument) and **actual arguments**(values assigned to the corresponding formal arguments)

**Example 1: Variables as Parameters**

#include <stdio.h>

void main()

{

int a=5,b=6;

swap1(a,b); // It means pass by value. Here after this function a, b values will not swap. Because it swapped when function is called, but while returning it gets vanished. So to solve this. Pass by reference

printf(“%d %d”,a,b); // here it gives 5 6

swap2(&a, &b); // swap using pointers which points to address.

printf(“%d %d”,a,b); // here it gives 6 5

}

|  |  |
| --- | --- |
| **Pass by value**  void swap1(int a1, int a2);  {  int temp = a1;  a1=b1;  b1=temp;  } | **Pass by Reference**  void swap2( int \*a1, int \*b1)  {  Int temp = \*a1; // contents of a1 storing in temp.  \*a1 = \*b1;  \*b1 = temp;  } |

Example 2: **Arrays as Parameters**

void main()

{

int a[5] = {1,2,3,4,5};

int b[]= {-3,-2.-1,0,1,2,3};

printt1(a,5);

printt1(b,7);

}

**All means same passing by reference.**

|  |  |  |
| --- | --- | --- |
| void print1(int tab[], int N)  {  int i;  for(i=0;i<N;i++)  printf(“%d “,tab[i]);  } | void print2(int \*tab, int N)  {  int \*ptr;  for(ptr=tab; ptr<tab+N; ptr++)  printf(“%d “, \*ptr);  } | void print3(int \*tab,int N)  {  int i;  for(i=0;i<N;i++,tab++)  printf(“%d “, \*tab);  } |

**POINTERS**

**Example**

char \*ptr = Null;

s1 = “Amit,Varun,asha”;

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | m | I | t | , | V | A | R | . | . |
| 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | . | . | . | . |

ptr = s1; ----->>>>>>>>>>> it will point to address of s1, ptr = 2000

printf(“%c”,\*ptr) ----->>>>>>>>>>> A (it points to base address of s1, value(2000) -> and its value is A character)

ptr++; (now pointing address will be incremented by 1) ptr = 2001

putchar(\*ptr) ----->>>>>>>>>>> m ---> value(2001).

1. //Accessing string using delimiter

#include <stdio.h>

int main() {

char lines[] = "Amit,Kumar,Ravi";

char Names[10][20];

char \*ptr = NULL;

ptr = lines;

int row =0;

int col =0;

while(\*ptr!='\0')

{

putchar(\*ptr);

ptr++;

}

ptr = lines;

while(\*ptr!='\0')

{

if(\*ptr==',')

{

Names[row][col] == '\0';

break;

}

Names[row][col] = \*ptr;

ptr++;

col++;

}

putchar(\*ptr);

puts(Names[row]);

ptr++;

putchar(\*ptr);

return 0;

}

2.

#include <stdio.h>

int main() {

char lines[] = "Amit te,Kumar,Ravi122ww,dhhd,hds,dhg,hs,sh,hb,bsx,sgdsdh";

char Names[10][20];

char \*ptr = NULL;

ptr = lines;

int row =0;

int col =0;

while(\*ptr!='\0')

{

putchar(\*ptr);

ptr++;

}

ptr = lines;

while(\*ptr!='\0')

{

if(\*ptr==',')

{

Names[row][col] == '\0';

break;

}

Names[row][col] = \*ptr;

ptr++;

col++;

}

putchar(\*ptr);

puts(Names[row]);

while(1)

{

ptr++;

row++;

col=0;

if(\*ptr == '\0')

{

break;

}

while(\*ptr!='\0')

{

if(\*ptr==',')

{

Names[row][col] == '\0';

break;

}

Names[row][col] = \*ptr;

ptr++;

col++;

}

putchar(\*ptr);

puts(Names[row]);

}

ptr++;

putchar(\*ptr);

return 0;

}

Structures

We want to create multiple of its type – so we need user defined datatype --- Structure.

struct tagName{

mem/properties of structure

};

* Structure within a structure can also be defined.

Example:

typdef struct Employee

{

int id;

int sal;

int phno;

char Name[20];

char Gender;

}EMP;

**TO ACCESS:**

In main code : EMP e1;

e1.id

**TO ASSIGN VALUE:**

strcpy(e1.Name,”Bhima”) // for strings

e1.Gender = ‘M’;

***Assigning value from some other array.***

e1.id = atoi(Names[0]); // Here Names[0] is string, but we need integer. So use atoi to change string to int datatype

strcpy(e1.Name,Names[1]);

e1.Gender = Names[2][0]; here we need to get only character , so use col value.

e1.phno = atoi(Names[3]);

e1.sal = atoi(Names[4]);

***To display the employeeDetails.***

Int display(EMP \*e)

{

If(e==NULL)

Return 1;

Prinf(“\nID: %d”,e -> id); // in function argument it is EMP e, then it should be e.id in printf statement

// and all details similarly

Return 0;

}

// prototype this function after struct definition.

// while calling this function ----->>>> it should be display(&e); // here ‘&’ is used, because it is static array. If it is dynamic array, we should not use ‘&’

EMP \*testEmp = NULL;

testEmp = (Emp \*)malloc(sizeof(EMP)) ; // it will allocates space to testEmp.

**DYNAMIC MEMORY ALLOCATION**

: It points to unnamed address

For all data, memory must be allocated.

Compile -time (Static), Run – time (Dynamic)

Ex:

char c; -🡪 it requires one byte

int array[10]; 🡪 10\*sizeof(int) = 40 bytes it requires.

int \*array 🡪 array is a pointer to integer, but we don’t know size of it.

So use malloc() for it. Use #include <stdlib.h> for it. Allocates memory at run time.

**Syntax:** malloc(num\_items \* sizeof(datatype) 🡪 it returns void pointer, type cast to required datatype.

Ex:

//Statically allocates space for two pointers

int \*i;

int \*array;

//Dynamically allocates space for data.

i = (int \*)malloc(sizeof(int));

array = (int \*)malloc(n\*sizeof(int));

\* i =3; 🡪 Contents of pointer i assigned to 3.

Array[3] = 5; It will point to 3rd element of array and assigned to 5.

If variables allocated statically in a function, this variable cant refer outside the function. Because their space is deallocated upon return.

But when we allocated dynamically, it must deallocate explicitly using free(). If forget to free its space, it leads to memory leaks and running out of memory.

For 2D array 🡪

**Allocates space for M\*N matrix**

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

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

p[i] = (int \*) malloc(N\*sizeof(int));

**To deallocate**

For(i=0;i<M;i++)

Free(p[i]);

free(p);

p = NULL;

**Example:**

EMP e1; // e1 is named address

EMP \*e; // e is pointing to unmaned address

1. e = &e1; // Now e1 points to named address i.e.e1

***To scan ‘e1’ data***

scanf(“%d%d%d”, &e1.id,&e1.sal,&e1.phno);

scanf(“%s”,e1.name);

getchar(); to free

scanf(“%c”,&e1.Gender);

***To print ‘e’ data***

Printf(“\nID: %d”,e -> id);

Similarly all

1. e = (EMP \*)malloc(sizeof(EMP)); // unnamed address

***To scan ‘e’ data***

scanf(“%d%d%d”, &e->id,&e->sal,&e->phno);

scanf(“%s”,e->name);

getchar(); to free

scanf(“%c”,&e->Gender);

***To print ‘e’ data***

Printf(“\nID: %d”,e -> id);

Similarly all

free(e); // we have to use free( ) if we allocated dynamically an array to deallocate.

1. **To have n number of employees**

Int noEmp;

Scanf(“%d”,&noEmp);1

e = (EMP \*)malloc(noEmp \* sizeof(EMP));,

temp = e;

**To getEmpDetails**

For(i=0;i<noEmp;i++;e++)

getDetails(e);

e = temp;

**To DisplayDetails**

For(i=0;i<noEmp;i++;e++)

DispDetails(e);

Free(e); // here we will get error as invalid pointer, because it is pointing to null character which is not reserved.

**UNION**

We can access only one member or property at a time.

**ENUM**

Is a set of named integer constants that specify all the legal values a variable of that type can have.

**FILES**

A collection of logically related information.

To display content of file, we need to know starting and ending address. It can be accessed by pointer. We need to have structure for it i.e. FILE. A special data structure.

Functions are available for character-based as well as string-based.

*Declaration:*

FILE \*fp;

FILE \*fopen(char \*name,char \*mode);

fp = fopen(“file name”,”mode”);

// if file is able to open, then only we can have address.

// access mode – append , cursor will blink at end of file. ---->>>>> a (if file is not present, it will create a file. Stream is positioned at the end of the file.)

a+ --open reading and appending . and writing at end of the file.

//access mode – read, we can only read ---->>>>> r

//access mode – write, we can only write ---->>>>> w (if file not present, it will create a file, if file is present, it will clear off)

//access mode – read & write, we can read & write ---->>>>> r+ , w+(if file not present, it will create a file, if file is present, it will truncate(clear off))

fclose() – to close file, after opening a file. Otherwise it will error Memory leakage.

FILE --->> Data Structure

fd ----->> file pointer to FILE Data Structure

fopen(“NameofFILE”,”Mode”) ----->> opening a file

Mode ----->> r,w,a,r+,w+,a+

fclose(fd) ----->> close the opened file.

fprintf, fscanf ----->> formatted writing and reading resp.

fputs, fgets ----->> unformatted i/o operation.

fwrite/fread ----->> reading/writing binary objects.

fseek( ) ---- >>> sets the file position indicator for the stream pointed to by stream

fseek(FILE \*stream,long offset, int whence);

ftell( ) ---- >>> obtains the current value of the file position indicator.

ftell(FILE \*stream);

rewind( ) – sets the file position indicator for the stream pointed to by stream to the beginning of the file. ----- >>>>> it is equivalent to void fseek(stream,OL,SEEK\_SET)

rewind(FILE \*stream );q

SEEK\_SET – seek from beginning of file

SEEK\_CUR – seek from current position

SEEK\_END – seek from end of file

fread( )

fread(void \*ptr, size\_t size,size\_t nmemb,FILE \*stream);

reads nmemb items, each size bytes, from the stream, storing them at location given by ptr.

fwrite(const void \*ptr, size\_t size,size\_t nmemb,FILE \*stream);

writes nmemb items, each size bytes, to the stream, obtaining them from location given by ptr.