**Storage Class:**

In addition to data types each variables has one more attribute known as storage class. The proper use of storage classes make our program efficient and fast. There are four types of storage classes-

1. Automatic
2. External
3. Static
4. Register

A storage class decides about these four aspects of a variable-

1. Life time – time between the creation and destruction of the variable.
2. Scope – location where the variable is available for use.
3. Initial value – default value taken by the uninitialized variable.
4. Place of storage – place in memory where the memory is allocated for the variable.
5. **Automatic:**

**All the variables declared inside a block or function without any class specifier are call automatic variable.**

They are called automatic because storage for them is reserved automatically each time when the control enters the function/block and are released automatically when the function/ block terminates

**The uninitialized auto variable initially contains garbage value**. Scope of these variables are limited to the function/block.

1. **Extern:**

int var;

Here, an integer type variable called var has been declared as well as defined. (remember that definition is the super set of declaration). Here the memory for var is also allocated. Now here comes the surprise, when we declared/defined a C function, we saw that an extern was present by default. While defining a function, we can prepend it with extern without any issues. But it is not the case with C variables. If we put the presence of extern in variable as default then the memory for them will not be allocated ever, they will be declared only. Therefore, we put extern explicitly for C variables when we want to declare them without defining them. Also, as the extern extends the visibility to the whole program, by externing a variable we can use the variables anywhere in the program provided we know the declaration of them and the variable is defined somewhere.

Now let us try to understand extern with examples.

Example 1:

|  |
| --- |
| int var;  int main(void)  {     var = 10;     return 0;  } |
|  |

Analysis: This program is compiled successfully. Here var is defined (and declared implicitly) globally.

Example 2:

|  |
| --- |
|  |
| extern int var;  int main(void)  {  return 0;  } |
|  |
|  |

Analysis: This program is compiled successfully. Here var is declared only. Notice var is never used so no problems.

Example 3:

|  |
| --- |
| extern int var;  int main(void)  {   var = 10;   return 0;  } |

Analysis: This program throws error in compilation. Because var is declared but not defined anywhere. Essentially, the var isn’t allocated any memory. And the program is trying to change the value to 10 of a variable that doesn’t exist at all.

Example 4:

|  |
| --- |
| #include "somefile.h"  extern int var;  int main(void)  {   var = 10;   return 0;  } |

Analysis: Supposing that somefile.h has the definition of var. This program will be compiled successfully.

Example 5:

|  |
| --- |
| extern int var = 0;  int main(void)  {   var = 10;   return 0;  } |

Analysis: Guess this program will work? Well, here comes another surprise from C standards. They say that. if a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated i.e. that variable will be considered as defined. Therefore, as per the C standard, this program will compile successfully and work. So that was a preliminary look at “extern” keyword in C.

I’m sure that you want to have some take away from the reading of this post. And I would not disappoint you.

In short, we can say

1. Declaration can be done any number of times but definition only once.
2. “extern” keyword is used to extend the visibility of variables/functions ().
3. Since functions are visible throughout the program by default. The use of extern is not needed in function declaration/definition. Its use is redundant.
4. When extern is used with a variable, it’s only declared not defined.
5. As an exception, when an extern variable is declared with initialization, it is taken as definition of the variable as well.
6. **STATIC storage class:**

Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope.

**Syntax: static data\_type var\_type = var\_value**

* Static variables are allocated memory in the data section not in the stack section.
* Static variables are initialised with zero value ‘0’ like global variable if they are not initialize explicitly.
* In C, static variables can only be initialized using constant literals. For example, following program fails in compilation. See [this](http://www.geeksforgeeks.org/g-fact-80/)for more details.

|  |
| --- |
| #include<stdio.h>  int initializer(void)  {      return 50;  }  int main()  {      static int i = initializer();      printf(" value of i = %d", i);      getchar();      return 0;  } |
|  |
|  |

If we change the program to following, then it works without any error.

|  |
| --- |
| #include<stdio.h>  int main()  {      static int i = 50;      printf(" value of i = %d", i);      getchar();      return 0;  } |

The reason for this is simple: All objects with static storage duration must be initialized (set to their initial values) before execution of main () starts. So a value which is not known at translation time cannot be used for initialization of static variables.

* If an extern variable is defined as static it cannot be used by the other files of the program.

e.g. first.c

static int i=10;

int main()

{

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

}

Second.c

Int func()

{

\*\*\*\*

\*\*\*\*

}

in the above case the variable ‘**int i**’ cannot be used by other file by extern declaration for it.

1. **Register storage class:**

Register storage class can be applied to only local variable. Its scope, initial value are same as that of automatic variable. Register variables don’t have address so ‘&’ cannot be applied to register variable. Only register storage class specifier can be applied to formal arguments of a function while the other three storage class specifier cannot be used that way.

**POINTERS:**

**1. Array of pointers to store string.**

e.g.

#include <stdio.h>

const int MAX = 4;

int main () {

char \*names[] = {

"Zara Ali",

"Hina Ali",

"Nuha Ali",

"Sara Ali"

};

int i = 0;

for ( i = 0; i < MAX; i++) {

printf("Value of names[%d] = %s\n", i, names[i] );

}

return 0;

}

**2. Return Pointer from function.**

It is not a good idea to return the **address of a local variable** outside the function, so you would have to define the local variable as **static** variable.

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

int \* getRandom( ) {

static int r [10]; **// this variable e is declared as static as we are retuning its address outside the // function**

int i;

for ( i = 0; i < 10; ++i) {

r[i] = rand ();

printf ("%d\n", r[i] );

}

return r;

}

int main () {

int \*p;

int i;

p = getRandom();

for ( i = 0; i < 10; i++ ) {

printf("\*(p + [%d]) : %d\n", i, \*(p + i) );

}

return 0;

}

**3. void pointer—generic pointer:**

**e.g.**

void main(){

int a;

int \*p0;

void \*p1; // generic pointer

char \*p2;

p2 =p0; // will give us compilation error

p2= (char\*) p0; //so we need to type cast p0 as a char

p1=p0; // it will not generate any error as p1 is a void pointer

p1=p2; // it will not generate any error as p1 is a void pointer

**//\*\*Note: but dereferencing is not possible using void pointer. And infect increment of these pointer is also not possible (p1+1).**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**main () {**

Char c[] =”Hello world”; // string get stored in the memory allocated for the array

**Char\* c =”Hello world”; // string gets stored in as a compile time constant in the**

**//text segment**

}

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**4. POINTERS AND MULTIDIMENSIONAL ARRAY:**

**B[i] [j] = \*(B[i] + j) =\*(\* (B + i) +j)**

**Where**

**B[i] is an integer pointer i.e. int\***

**B pointer to 1D array of 3 int**

**400 404 408 412 416 420**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |

B[0][0] B[0][1] B[0][2] B[1][0] B[1][1] B[1][2]

int B[2][3];

print B; //it will print 400 -------pointer to 1D array of 3 integer

print \*B; //it will also print 400 ----pointer to int i.e. int\*

print B [0]; // again 400 will be printed

**5. FUNCTION POINTER:**

Uses of function pointer:

1. When we want to pass a function as a parameter to another function. This decreases the execution time and decrease the complexity.
2. Function pointers are used to call a function at run time instead of compile time.
3. Are used in callback functions.
4. Is used to return more than one value to the main function.

e.g.

**int getSum (int, int); // a user define function;**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*now we need to declare a function pointer, which can be done as follows\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**int c;**

**int (\*p)(int, int);**

**p= &getSum;**

**c = (\*p) (a, b);** //**dereferencing and executing the function**

**// the above two statements can also be written as follows.**

**//p=getSum;**

**//c=p (a, b);**

**Pointer to constant vs constant pointer:**

**int \*** **const ptr** —> **ptr** is constant pointer. You can change the value at the location pointed by pointer **ptr**, but you cannot change **ptr** to point to other location. **int const** \* **ptr** —> **ptr** is a pointer to a constant. You can change **ptr** to point other variable. But you cannot change the value pointed by **ptr**.

**STRING:**

**Some inbuilt functions:**

**1.strchr(s1,ch)**

**char \*strchr(const char\*, int);**

Returns a pointer to the first occurrence of character ch in string s1.

#include<stdio.h>

#include<string.h>

void main(){

char str[] ="i am gonna strive hard for the next 6 months.";

char ch = 'm';

char \*ptr;

ptr = strchr(str, ch);

int k=1;

while(ptr!=NULL){

printf("occurance: %d", k);

printf("\nthe position of the occurance of the word |%c| is %ld", ch, ptr-str+1);

printf("\nthe string after the charecter |%c| is |%s|\n\n\n", ch, ptr);

ptr= strchr(ptr+1, ch);

k++;

}}

**2.strstr(s1,s2).**

Returns a pointer to the first occurrence of string s2 in string s1.

**3. strcpy()**

**4. strcat()**

**5. strcom(**)

**STRUCTURE:**

**struct [structure tag] {**

**member definition;**

**member definition;**

**...**

**member definition;**

**} [one or more structure variables];**

**Nested Structure:**

How nested structure is written shown hereunder-

e.g.

struct address{

char H/no[];

int lane\_no;

};

struct name {

char name[];

int roll;

struct address add;

} shubham;

strcpy (shubham.name, “Shubham Sharma”);

Shubham.roll=10;

**Shubham.add.lane\_no=151; //using dot operator again and again variable in the nested structure can //be accessed**

**STRUCTURE as function argument:**

#include<stdio.h>

#include<string.h>

struct kaka\_pg {

char name [10];

char institute [10];

char course [10];

} Ashok;

void func(struct kaka\_pg uncle); **// declaration of this function must be done after the declaration of // the structure**

void main(){

strcpy(Ashok.name, "vivek");

strcpy(Ashok.institute, "vector");

strcpy(Ashok.course, "embedded system");

func(Ashok);

}

void func(struct kaka\_pg uncle){

printf("name of the student:%s\n", uncle.name);

printf("name of the institute:%s\n", uncle.institute);

printf("name of the course:%s\n", uncle.course);

}

**Pointer to structure:**

**struct [name of structure] \*structure pointer;**

we can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows--

**struct\_pointer = &structure variable;**

#include<stdio.h>

#include<string.h>

struct kaka\_pg {

char name[10];

char institute[10];

char course[16];

} Ashok;

void func(struct kaka\_pg \*uncle);

void main(){

strcpy(Ashok.name, "vivek");

strcpy(Ashok.institute, "vector");

strcpy(Ashok.course, "embedded system");

func(&Ashok);

}

void func(struct kaka\_pg \*uncle){

printf("name of the student:%s\n", uncle->name);

printf("name of the institute:%s\n", uncle->institute);

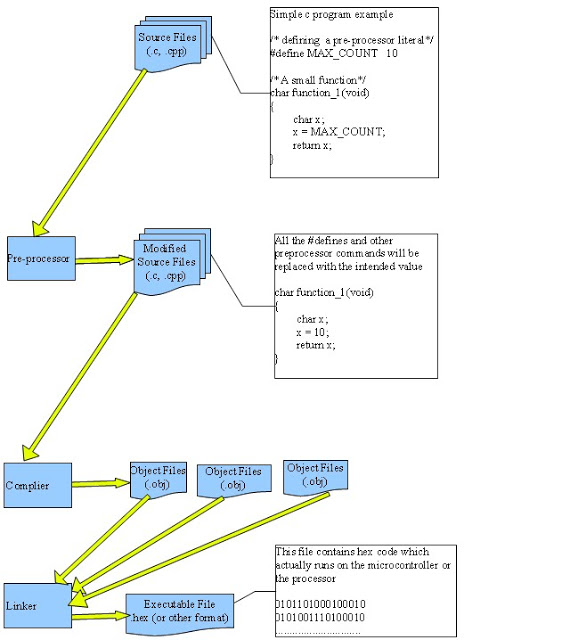
printf("name of the course:%s\n", uncle->course);

}

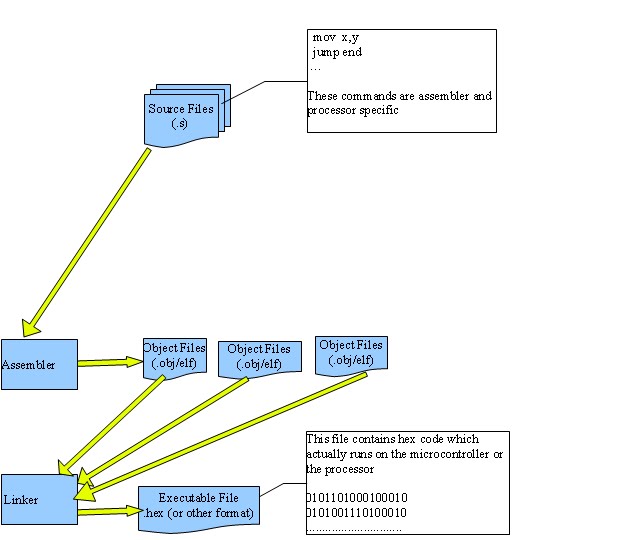
**typedef & #define:**

* typedef is limited to giving symbolic names to types only where as #define can be used to define alias for values as well, q., you can define 1 as ONE etc.
* Typedef interpretation is performed by the **compiler** whereas #define statements are processed by the **pre-processor.**

**Pre-Processing / Compilation / Linking**



**Assembly / Linking**



**Little endian and big endian architecture:**

In Big-Endian, Most-Significant-Byte (MSB) is stored at lowest address while, in Little-Endian, Least-Significant-Byte (LSB) is stored at lowest address.

x four byte integer contains a hex value0x76543210 (0xstands for hex), the least significant byte will contain 0x10 and the most significant byte will store 0x76. Now if you take a pointer c of type char and assign x's address to c by casting x to char pointer, then on little endian architecture you will get 0x10 when \*c is printed and on big endian architecture you will get 0x76 while printing down \*c. Thereby you can find out the endianness of your machine.

#include <stdio.h>

int main ()

{

unsigned int x = 0x76543210;

char \*c = (char\*) &x;

printf ("\*c is: 0x%x\n", \*c);

if (\*c == 0x10)

{

printf ("Underlying architecture is little endian. \n");

}

else

{

printf ("Underlying architecture is big endian. \n");

}

return 0;

}

**FILE:**

**OPENING FILES:**

**fopen()** function is used to open or create a new new file.

**Syntax**: FILE \*fopen(const char \* file name, const char \* mode);

|  |  |
| --- | --- |
| **Mode** | **Description** |
| **R** | Opens an existing text file for reading purpose. |
| **W** | Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file. |
| **A** | Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content. |
| **r+** | Opens a text file for both reading and writing. |
| **w+** | Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist. |
| **a+** | Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended. |

**Closing a file:**

**Syntax: int fclose(FILE \*fp);**

The fclose () function returns zero on success, or EOF if there is any error in closing the file. This function actually flushes any any data still pending in the buffer to the file, close the file, and release any memory used by the file. The EOF is a constant define in the **stdio.h header** file.

**Writing a FILE:**

**Syntax: 1. int** fputc**(int** c**, FILE** \*fp**);**

**2.** **int** fputs**(const char** \*s**, FILE** \*fp**);**

both of the above functions returns a non negative number on success and EOF on failure.

**Reading a FILE:**

**Syntax:**

**1. int fgetc(FILE \*fp);**

**2. char \*fgets(char \*buf, int n, FILE \*fp);**

The **fgetc()** reads a charecter from the file pointed by the file pointer. The return value is the charecter read, in case of failure it returns EOF.

The functions **gets()** reads up to n-1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character.

#include<stdio.h>

void main(){

FILE \*fp;

char buff[255];

fp=fopen("file.txt", "r");

fscanf(fp,"%s", buff); //reads untill it encounters a space

printf("1 :%s\n", buff);

fgets(buff, 255, fp); //reads untill the \o i.e. NULL charecter

printf("2 :%s", buff);

fgets(buff, 255, fp);

printf("3 :%s\n", buff);

fclose(fp);

}

**C-PREPROCESSOR**

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.

All preprocessor command begins with a symbol hask (#)

|  |  |
| --- | --- |
| **Directive** | Description |
| **#define** | Substitutes a preprocessor macro. |
| **#include** | Inserts a particular header from another file. |
| **#undef** | Undefines a preprocessor macro. |
| **#ifdef** | Returns true if this macro is defined. |
| **#ifndef** | Returns true if this macro is not defined. |
| **#if** | Tests if a compile time condition is true. |
| **#else** | The alternative for #if. |
| **#elif** | #else and #if in one statement. |
| **#endif** | Ends preprocessor conditional. |
| **#error** | Prints error message on stderr. |
| **#pragma** | Issues special commands to the compiler, using a standardized method. |

**Predefined macros:**

|  |  |
| --- | --- |
| **Macro** | **Description** |
| **\_\_DATE\_\_** | The current date as a character literal in "MMM DD YYYY" format. |
| **\_\_TIME\_\_** | The current time as a character literal in "HH:MM:SS" format. |
| **\_\_FILE\_\_** | This contains the current filename as a string literal. |
| **\_\_LINE\_\_** | This contains the current line number as a decimal constant. |
| **\_\_STDC\_\_** | Defined as 1 when the compiler complies with the ANSI standard. |

**RECURSION**

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop. Recursive functions are basically used to solve some mathematical problem.

**ERROR HANDLING**

As such, C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and set an error code **errno**. It is set as a **global variable** and indicates an error occurred during any function call.

So a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice, to set errno to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program.

**errno, perror(), strerror()**

C programming language provides two functions **prror(), strerror()** which prints the value returned by the variable **errno.**

* **perror()** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.
* **strerror()** function, which returns a pointer to the textual representation of the current errno value.

Second important point to note is that you should use **stderr** file stream to output all the errors.

# **Command Line Arguments**

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program.

#include <stdio.h>

int main( int argc, char \*argv[] ) {

if( argc == 2 ) {

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 ) {

printf("Too many arguments supplied.\n");

}

else {

printf("One argument expected.\n");

}

}

**DYNAMIC Memory allocation:**

**1. malloc() :** stands for memory allocation.

The function **malloc()** reserves a block of memory of specified size and return a pointer of type **void**, which can be casted into a pointer of any form.

syntex: ptr =(cast\_type \*) malloc (size);

mallocl() does not initialize the memory location with any value i.e. the memory allocate with the help of malloc() contains garbage value.

**2. calloc() :** stands for contiguous allocation.

calloc is also used for dynamic memory allocation. The difference between calloc and malloc is that calloc takes two arguments(number of block and size of each block) while maloc takes only one argument. Calloc initialize the location with zero value which malloc does not.

Syntex: ptr =(cast\_type \*) calloc (number of block, size of each block);

**3. realloc :**

**Dangling, void, NULL and wild pointer**

**Dangling pointer:**

A pointer is consider dangling if it is still pointing to a memory location which has been deleted or freed is called dangling pointer.

**Void pointer:**

Void pointer is a specific pointer type – void \* – a pointer that points to some data location in storage, which doesn’t have any specific type. Void refers to the type.

* Void pointers cannot be dereferenced. It can however be done using typecasting the void pointer.
* Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

e.g.

#include<stdlib.h>

int main()

{

int x = 4;

float y = 5.5;

//A void pointer

void \*ptr;

ptr = &x;

// (int\*)ptr - does type casting of void

// \*((int\*)ptr) dereferences the typecasted

// void pointer variable.

printf("Integer variable is = %d", \*( (int\*) ptr) );

// void pointer is now float

ptr = &y;

printf("\nFloat variable is= %f", \*( (float\*) ptr) );

return 0;

}

**NULL pointer:**

NULL Pointer is a pointer which is pointing to nothing. In case, if we don’t have address to be assigned to a pointer, then we can simply use NULL.

#include <stdio.h>

int main()

{

// Null Pointer

int \*ptr = NULL;

printf("The value of ptr is %u", ptr);

return 0;

}

**Wild pointer:**

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

int main()

{

int \*p; /\* wild pointer \*/

int x = 10;

// p is not a wild pointer now

p = &x;

return 0;

}

To calculate the time taken by a process, we can use clock () function which is available in “time.h” header file. We use clock () function at the start and ending of any portion of a program to calculate the time taken for execution.

e.g.

#include<time.h>

clock\_t start, end;

double time\_taken;

start=clock();

\*

\*

Func();

\*

\*

End=clock();

time\_taken= (double)(end-start)/CLOCK\_PER\_SEC;

printf(“%lf”, time\_taken);

Some compiler does not allow to declare a variable inside loop. To make it possible we need to add c99 mode, which can be added at the time of compilation as shown here under.

e.g. gcc –std=c99 file\_name.c –o[object\_file]