Programing Methodology in C

Lecture 12 – typedef and storage classes in C

By Elubu Joseph

josebulinga@gmail.com

Agenda

1. typedef and
2. storage classes in C

typedef in C

typedef in C

typedef is a keyword used in C language to assign alternative names to existing datatypes. Its mostly used with user defined datatypes, when names of the datatypes become slightly complicated to use in programs. Following is the general syntax for using typedef,

```
typedef <existing_name> <alias_name>
```

Example

```
typedef int HM;
HM i=0;
```

Example of program that prints the elements of array

```c
#include<stdio.h>
typedef int Arrayer[8];
typedef int HM; //alias of int create
int main(){
Arrayer k = {20,30,40,50,60,34,98,7};// alias of Arrayer created
HM i;
printf("Array elements\n");
for(i=0; i<8; i++){
 printf("%d\n", k[i]);
}
}
```

OUTPUT:

```
Array elements
20
30
40
50
60
34
98
7
```

typedef in C +

Lets take an example and see how **typedef** actually works.

```
typedef unsigned long ulong;
```

The above statement define a term **ulong** for an **unsigned long** datatype. Now this **ulong** identifier can be used to define **unsigned long** type variables.

```
ulong i, j;
```

Application of typedef with structures

typedef can be used to give a name to user defined data type such as structures as well. We will soon apply this concept in structures, but before we do so, lets first remind ourselves about normal structure declaration and definition, then we finally apply it.

Example of Structure Definition without typedef

```
struct Student {
        char name[25];
        int age;
        char branch[10]; // F for female and M for male
        char gender;
       char father[];
};
```

Here struct Student declares a structure to hold the details of a student which consists of 5 **structure elements or members**., namely name, age, branch, gender and father.

Each member can have different datatype, like in this case, name is an array of char type and age is of int type etc. **Student** is the name of the structure and is called as the **structure tag**.

Declaring Structure Variables without typedef

It is possible to declare variables of a **structure**, either along with structure definition or after the structure is defined.

Structure variable declaration is similar to the declaration of any normal variable of any other datatype. Structure variables can be declared in following ways:

1) Declaring Structure variables separately

```
struct Student {
    char name[25];
    int age;
    char branch[10]; //F for female and M for male
    char gender;
};
struct Student S1, S2; //declaring variables of struct
Student
```

2) Declaring Structure variables with structure definition

```
struct Student {
      char name[25];
      int age;
      char branch[10]; //F for female and M for male
      char gender;
}S1, S2;
```

Here S1 and S2 are variables of structure Student. However this approach is not much recommended.

Application of typedef in Structures

typedef can be used to give a name to user defined data type as well. Lets see its use with structures.

```
typedef struct type_name{
        type member1;
        type member2;
        type member3;
} Alias_name;
```

Here **type_name** represents the stucture definition associated with it. Now this **type_name** can be used to declare a variable of this stucture type.

```
Alias_name t1, t2;
```

Example of Structure definition using typedef

```
#include<stdio.h>
#include<string.h>
typedef struct employee {
char name[50];
int salary;
}emp;
void main( ) {
       emp e1;
              printf("\nEnter your name:\t");
              gets(e1.name);
              printf("\nEmployee name is %s", e1.name);

}
```

typedef and Pointers

typedef and Pointers

typedef can be used to give an alias name to pointers also. Here we have a case in which use of typedef is beneficial during pointer declaration.

In Pointers, * binds to the right and not on the left.

```
int* x, y;
```

By this declaration statement, we are actually declaring x as a pointer of type int, whereas y will be declared as a plain int variable.

But if we use typedef like we have used in the example above, we can declare any number of pointers in a single statement.

```
typedef int* IntPtr;
IntPtr x, y, z;
```

NOTE: If you do not have any prior knowledge of pointers, have a look on our 11th Lecture.

Storage classes in C

In C language, each variable has a storage class, the following are the keys that help us differentiate between storage classes:

1. **Scope of a variable**; area where the value of the variable would be available inside/within a program.
2. **default initial value of avariable;** if we do not explicitly initialize that variable, what will be its default initial value.
3. **lifetime of the variable;** for how long will that variable exist.

Storage classes in C+

The following storage classes are most often used in C programming,

i. Automatic variables

ii. External variables

iii. Static variables

iv. Register variables

i. Automatic variables: auto

Scope: Variable defined with auto storage class are local to the function block inside which they are defined.

Default Initial Value: have any random value i.e garbage value.

Lifetime: Till the end of the function/method block where the variable is defined.

A variable declared inside a function without any storage class specification, is by default an **automatic variable**.

i. Automatic variables: auto+

Are created when a function is called and are destroyed **automatically** when the function's execution is completed.

Automatic variables can also be called **local variables** because they are local to a function. By default they are assigned **garbage value** by the compiler.

Example.

```
#include<stdio.h> void main() {
int detail; // or
auto int details; //Both are same
}
```

ii. External or Global variable

- **Scope:** Global i.c everywhere in the program. These variables are not bound by any function, they are available everywhere.
- **Default initial value:** 0(zero).
- **Lifetime:** Till the program doesn't finish its execution, you can access global variables.

A variable that is declared outside any function is a **Global Variable**. Global variables remain available throughout the program execution. By default, initial value of the Global variable is 0(zero). And their values can be changed by any function in the program.

Sample Program handling global variable

```
#include<stdio.h>
int number; // global variable
void main() {
printf(“The value of number before initialization is%d\n",
number);
number = 10;
printf("I am in main function. My value is %d\n", number);
fun1();
fun2();
} /* This is function 1 */
void fun1() {
number = 20;
printf("I am in fun1 function. My value is %d\n", number);
}
/* This is function 1 */
void fun2() {
number = 30;
printf("\nI am in fun2 function. My value is %d", number); }
```

Output

I am in function main. My value is 10

I am in fun1 function. My value is 20

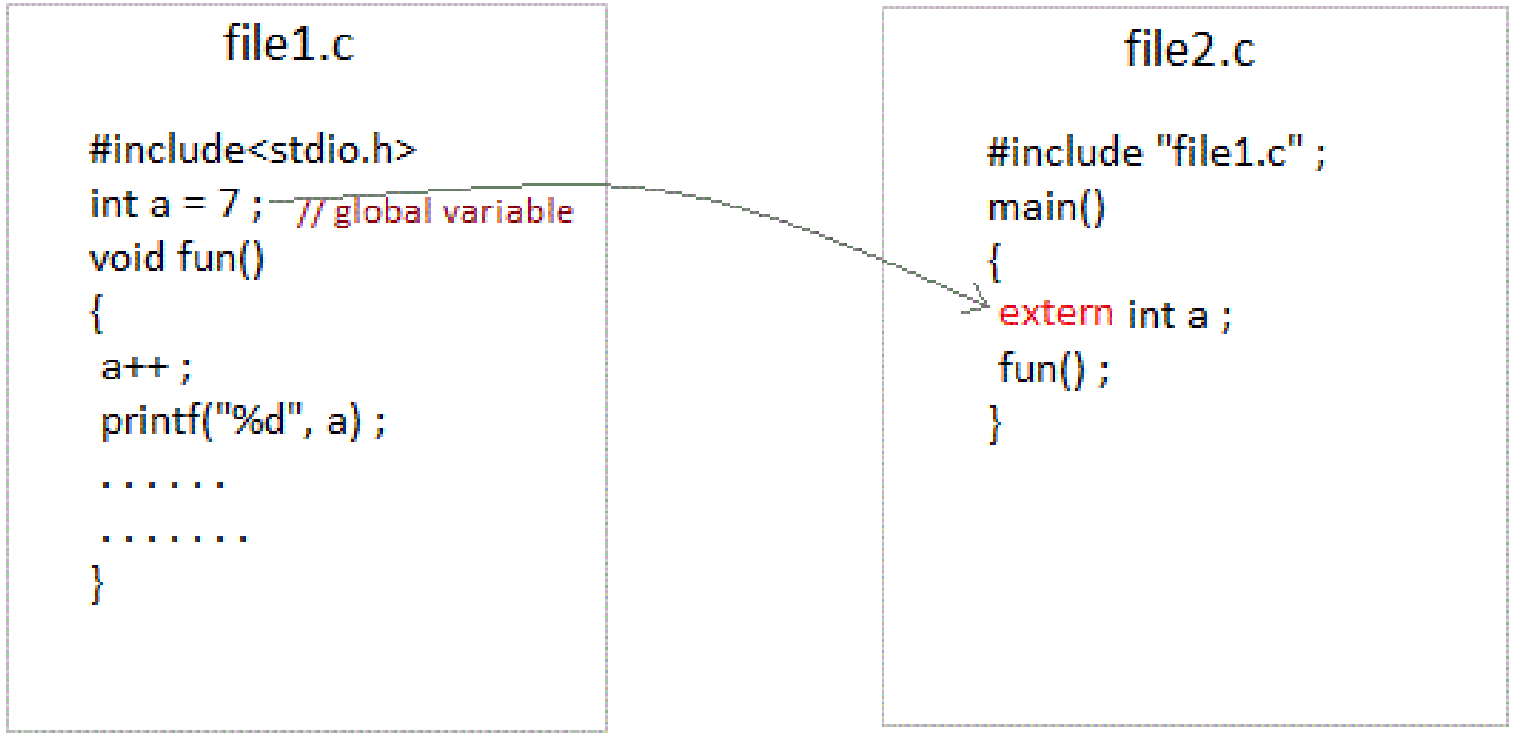
I am in fun2 function. My value is 20

Here the global variable **number** is available to all three functions and thus, if one function changes the value of the variable, it gets changed in every function unless the value is changed by individual functions. See fun2.

Note: Declaring the storage class as global or external for all the variables in a program can waste a lot of memory space because these variables have a lifetime till the end of the program. thus, variables, which are not needed till the end of the program, will still occupy the memory and thus, memory will be wasted.

Accessing global variable using extern keyword

The extern keyword is used with a variable to inform the compiler that this variable is declared somewhere else. The extern declaration does not allocate storage for variables.



global variable from one file can be used in other using **extern** keyword.

Problem when extern is not used

```
int main() {
 a = 10; //Error: cannot find definition of variable 'a'
printf("%d", a);
}
```

Example using extern in same file

```
int main()
{
        extern int x; //informs the compiler that it is defined somewhere else
        x = 10;
        printf("%d", x);
}
int x; //Global variable x
```

Static variables

Scope: Local to the block in which the variable is defined or global depending on the declaration

Default initial value: 0(Zero).

Lifetime: Till the whole program doesn't finish its execution.

A static variable tells the compiler to persist/save the variable until the end of program. Instead of creating and destroying a variable every time when it comes into and goes out of scope, static variable is initialized only once and remains into existence till the end of the program.

Static variables +

A static variable can either be internal or external depending upon the place of declaration. Scope of **internal static** variable remains inside the function in which it is defined. **External static** variables remain restricted to scope of file in which they are declared.

They are assigned **0 (zero)** as default value by the compiler.

```c
#include<stdio.h>
void test(); //Function declaration
(discussed in next topic)
int main() {
      test();
      test();
      test();
}

void test() {
      static int a = 0; //a static variable
      a = a + 1;
      printf("%d\t",a);
}
```

Output

1 2 3

4 Register variable

1. **Scope:** Local to the function in which it is declared.
2. **Default initial value:** Any random value i.e garbage value
3. **Lifetime:** Till the end of function/method block, in which the variable is defined.

Register variable +

Register variables inform the compiler to store the variable in CPU register instead of memory.

Register variables have faster accessibility than a normal variable. Generally, the frequently used variables are kept in registers.

Application of Register variable +

Application of register storage class, can be in using loops, where the variable gets to be used a number of times in the program, in a very short span of time.

NOTE: We can never get the address of such variables.

Syntax :

```
register int number;
```

Note: Even though we have declared the storage class of our variable number as register, we cannot surely say that the value of the variable would be stored in a register. This is because the number of registers in a CPU are limited. Also, CPU registers are meant to do a lot of important work. Thus, sometimes they may not be free. In such scenario, the variable works as if its storage class is auto.

Determining when and Why a storage class should be used

To improve the speed of execution of the program and to carefully use the memory space occupied by the variables, following points should be kept in mind while using storage classes:

1. We should use static storage class only when we want the value of the variable to remain same every time we call it using different function calls.
2. We should use register storage class only for those variables that are used in our program very often. CPU registers are limited and thus should be used carefully.

Determining when and why a storage class should be used and when

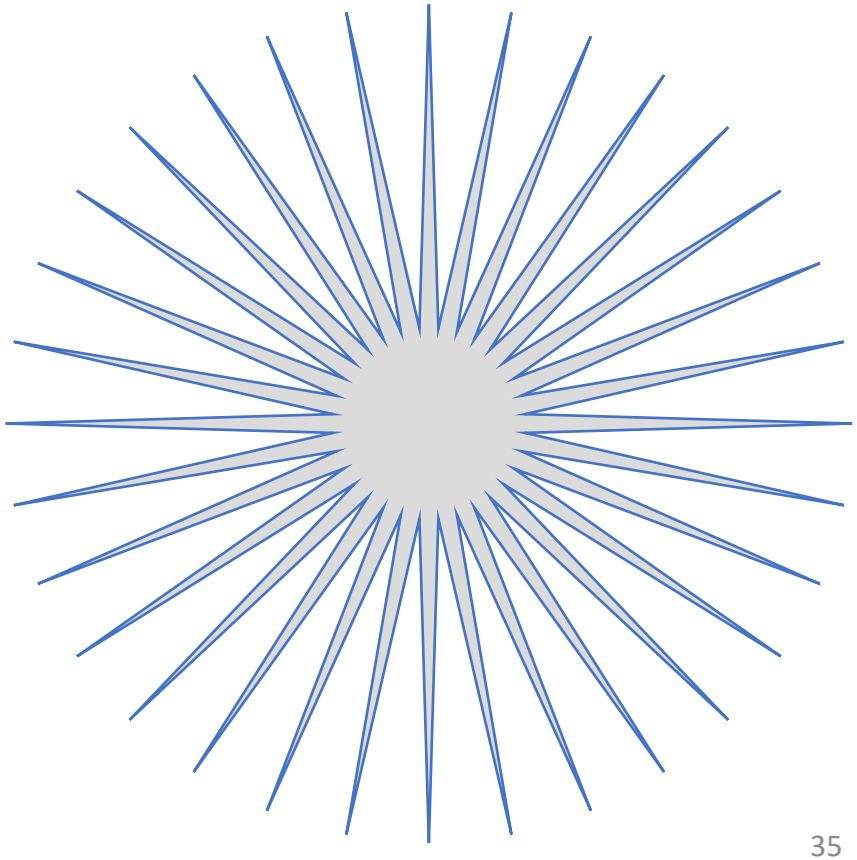
3. We should use external or global storage class only for those variables that are being used by almost all the functions in the program.

Note: If we do not have the purpose of any of the above mentioned storage classes, then we should use the automatic storage class.

Summary

1. typedef
2. storage classes in C (definition, types- global, static, registers and automatic varriables)

Thank you for you attention



Reference

Typedef in C. Studytonight.com. (n.d.). Retrieved November 11, 2021, from https://www.studytonight.com/c/typedef.php.

Storage classes in C. Studytonight.com. (n.d.). Retrieved November 12, 2021, from https://www.studytonight.com/c/storage-classes-in-c.php.