**Srorage class, ...**

A variable has the following attributes.

* Name
* location
* type
  + stands for a set of values and a set of operations
* value
  + may have default value based on storage class
* life – is about existence
  + When does the variable get location?
  + How long does the variable remain?
  + When does the variable die – loose its location?
* Scope – is about visibility
  + where can the variable be referred by its name?
  + Variable can have scope only when it has life
  + Variable may have life, but no scope!!
* storage class
  + In which part of the logical memory, is the allocation done?
  + How does this influence the storage class?
  + How does this influence the default initial value?
* Qualifier
  + How does this prefix change the size of the type?
  + How does this affect the range of values?
  + How does this affect compilation?

We shall discuss all these aspects with suitable examples.

**auto:**

The following example discusses the storage class auto – which is the default for variables declared in a block. The name auto remains to be a keyword for storage class in ‘C’, but the meaning has been changed in C++ 11. We rarely explicitly use the keyword.

These variables are the safest. The memory is managed on the stack. We shall never have garbage and memory leak on the stack. We can still create dangling pointers and corrupt the stack frame. Read the comments written in the file for more information.

Example : ex\_auto.c

// variable has

// a name

// a location

// a type

// a value

// storage class

// qualifier

// life

// scope

// We shall discuss the concept storage class along with other aspects

// storage class:

// global

// static

// auto

// register

// auto :

// stands for automatic

// is the default variables created within a block

// created on entry to the block

// removed on exit from the block

// created on stack

// variables can be of any type

// are not initialized by default

// life time : during the execution of the block

// scope : during the execution of the block

// accessible in the inner blocks as long as there is no clash of names

// there is no scope resolution operator

// holding a pointer to an automatic variable after its life ends causes dangling pointer

// note: auto keyword has totally different meaning in C++ 11

#include <stdio.h>

int main()

{

auto int a = 10; auto int b; // b not initialized

auto int\* p;

b = 20;

{

auto int a = 30; // automatic or local variable of the inner block

// a of the outer block can never be accessed within the inner block

printf("%d %d\n", a, b); // 30 20

p = &a; // ok at this point; p becomes dangling once the block is exited

}

}

Register: This is a request to allocate the variables in the registers of the CPU rather than the memory of the computer. This is a request as the number of registers are limited. Those variables which are accessed often should be in the registers. With the advancement in the compiler technology, the compilers can analyze the source code and make out which variables are accessed more often better than the programmers.

Example: ex\_register.c

// storage class: register

// registers are part of the CPU. If the variables can be stored in the registers,

// instead of main memory, accessing shall become faster. This can make the program execution

// faster. But there are only a few registers. All variables cannot be allocated on the

// registers.

// If the user can make out which variables of his program are likely to be access more often,

// he can give a request to the compiler to allocate the variable on register.

// These days the compiler can analyze the program and make out this aspect better.

// - register

// is a storage class

// is a request or a hit to the compiler to allocate the variable on the register

// treated as auto otherwise

// address operator not allowed on register variables

#include <stdio.h>

int main()

{

register int i;

register int res = 0;

for(i = 0; i < 10000; ++i)

{

res += i;

}

//printf("pointer : %p\n", &res); // NO

printf("res : %d\n", res);

}

global variables:

These variables are declared outside a block. These variables are created at the time of loading and they remain through out the program execution. These can be accessed across translation units. The linker gets to know about these variables. We may indicate that the variable can be used across files by declaring the variable outside a block – with or without initialization. All these variables are initialized to 0 by default. The linker is told that this translation has a public symbol. In another translation unit, we may declare the variable prefixed with the keyword extern and no initialization. This tells the compiler that the variable can be used in this translation unit, but no memory should be allocated. The compiler tells the linker that there is an external reference. The linker tries to find a public symbol in some other translation for all external references. The linker emits two possible errors – multiple definition or unresolved external.

Example : ex1\_global.c

// global:

// all variables declared outside a block with no storage class specified

// available across files

// linker takes care of connecting the usage and the definition

// extern : keyword ; tells the compiler to pass the name to the linker

// by default, all global names are passed to the linker

//

// Variables are allocated memory if the variable is initialized.

// We should not initialize the same variable in more than one file. That would cause linked error.

// If we declare and do not initialize, the C compiler allocates tentatively memory for the variable in each file - linker will keep only one of them

**// global variables should be avoided as far as possible.**

**// 1. functions written to process global variables are not flexible. They cannot work on any**

**// other variable at all. Not at all flexible.**

**// 2. create a type instead of global variable**

**// we can create any number of variables of a particular type**

**// 3. type errors can result in accessing global variables instead of resulting in preferred compile time error**

**// 4. may result in impure functions if global variables are used.**

**// may break the abstraction; may be difficult to comprehend; may be difficult to maintain**

#if 0

int a[10]; int n;

void mysort()

{

// routine for sorting array a having n elements

// cannot sort an array called b if any

}

#endif

#if 0

char name1[20]; int age1;

char name2[20]; int age2;

// instead

struct person { char name[20]; int age; };

// then created automatic variables

{

struct person person1;

struct person person2;

}

#endif

#include <stdio.h>

int a; // same as extern int

int main()

{

printf("a : %d\n", a); // 0

}

Example : ex2\_global.c

int a;

Global variables are not encouraged in programming. They are maintenance nightmare. We do not recommend global variables. So, in this whole course, but for this example, we never used global variables.

Please check out those comments in bold.

Static :

static is a highly misused word in Computer Science. It has number of meaning based on the context and based on the language.

We shall discuss only the meaning of static in ‘c’.

There are two types of static in ‘C’.

a) External static:

The keyword static used outside a block.

This can be associated with a variable or a function declaration and definition.

This indicates that the names can be used only in that translation unit or file. The name is not given to the linker as a public symbol.

We say that this name has internal linkage.

These variables are created at the time of loading and they remain through out the program execution.

b) Internal static:

The keyword static used inside a block.

These are associated with variables only. The C standard does not support nesting of function definitions. These variables are created at the time of loading and they remain through out the program. They are created in data segment. They can be initialized with compile time constants and are initialized to 0 by default. These variables persist unlike auto variables.

Here are some examples.

Example: ex\_static\_external.c

// external static:

// outside a block

// could be a variable or a function

// lifetime : through out the program

// created at load time

// variables initialized by default to 0

// destroyed when the program terminates

// allocated on data segment

// scope :

// only in that file - in that translation

// name is not passed onto the linker

#include <stdio.h>

static void foo()

{

printf("this is foo\n");

}

static char names[][10] = {

"one", "two", "three", "four"

};

int main()

{

foo();

for(int i = 0; i < 4; ++i)

{

printf("%s\n", names[i]);

}

}

example: ex\_static\_internal.c

// static:

// static has multiple meanings depending on the context of usage.

// In terms of storage class, there are two ways of using

// a) external static

// declared outside a block

// b) internal static

// declared inside a block

// Internal static variables:

// - life time : throughout the program

// created when the program starts executing - at the point of loading

// allocated on data segment

// can be initialized by compile time expressions

// uninitialized variables are filled with 0

// destroyed when the program terminates

// - scope : by name, limited to the block

// can be accessed using a pointer outside a block

// safe to return a pointer to a static variable of a block

// As it is associated with the function definition than a function call,

// we would prefer to avoid this feature. Java and most of the languages apart from c++

// which evolved from 'C' have abandoned this feature.

#if 0

#include <stdio.h>

int foo()

{

static int count = 0; // only once; at the point of loading

// Does something here

return ++count;

}

int main()

{

printf("foo called %d times\n", foo());

printf("foo called %d times\n", foo());

printf("foo called %d times\n", foo());

printf("foo called %d times\n", foo());

}

#endif

// note: this feature cannot be used for closure as the property is not associated with the

// callable; but with the definition.

#include <stdio.h>

int\* foo() // perfectly safe

{

static int count = 0; // only once; at the point of loading

// Does something here

count++;

return &count;

}

int main()

{

int \*p = foo();

printf("\*p : %d\n", \*p); // 1

int \*q = foo(); // p points to count of foo; changing count affects \*p !!

printf("\*p : %d\n", \*p); // 2 and not 1 !!

}

qualifiers:

We may change the range of values and/or precision by prefixing the types int and double.

short int or short

long int or long

long double

The sizes of the above types are implementation dependent.

This predicate will definitely hold good.

sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)

The range and precisions for floating point values in increasing order is:

float, double, long double.

We have an interesting qualifier volatile.

Example: ex\_volatile.c

This affects compiler optimization.

#include <stdio.h>

// Observe that a and b are not changing between the evaluation of c and d.

// The compiler can optimize the program by evaluating a + b only once and use it twice

// while evaluating c and d.

// But if a and b could change outside this thread of code, then the compiler cannot make out.

// In such cases, optimizing would be incorrect.

// We declare the variable as volatile to indicate to the compiler that it should not do

// "common subexpression evaluation CSE" optimization and it should get the value of volatile

// variables each time it is used.

int main()

{

// int a; int b;

volatile int a; volatile int b;

scanf("%d %d", &a, &b);

int c; int d;

c = a + b + 10;

printf("after summation one\n");

d = a + b + 20;

printf("after summation two\n");

printf("%d %d\n", c, d);

}

Thats about storage class, life, scope, … .