Storage Classes & Scope Rules

Dr Bhanu

Storage Classes

- We use identifiers for variable names.
- The attributes of variables include name, type, size and value.
- We also use identifiers as names for user-defined functions.
- Actually, each identifier in a program has other attributes, including storage class, storage duration, scope and linkage.
- C provides four storage classes, indicated by the storage class specifiers: `auto`, `register`, `extern` and `static`.
- An identifier's storage class determines its storage duration, scope and linkage.
- An identifier's storage duration is the period during which the identifier exists in memory.

Storage Classes

- The two characteristics that the storage class defines are
  - Lifetime
  - Visibility
- Lifetime of a variable is the length of time it retains a particular value
- Visibility or scope of a variable refers to those parts of a program that will be able to recognize it.
- Need for storage classes….write programs that
  - Use memory more efficiently, run faster
  - Less prone to errors (imp. in large programs)

```
/* This is a sample to demonstrate scope. The techniques
   used in this program should never be used in practice.
*/
#include  <stdio.h>
int fun (int a, int b);                          Global area

 int main (void)
 {
   int   a;                                      main's area
   int   b;
   float y;
   …
      {  // Beginning of nested block
        float a = y / 2;
        float y;                                 Nested block
        float z;                                 area
        …
        z = a * b;
        …
      }  // End of nested block
   …
 }  // End of main

int fun (int i, int j)
{
   int a;                                        fun's area
   int y;
   …
}  // fun
```

- Scope determines the region of the program in which a defined object is visible (i.e. the part where we can use object's name).
  - Variable, function declaration etc
- A block is zero or more statements enclosed in a set of braces.
  - Function's body
- Global area consists of all statements that are outside functions

Scope for Global and Block Areas

Storage Classes

- Some exist briefly, some are repeatedly created and destroyed, and others exist for the entire execution of a program.
- An identifier's scope is where the identifier can be referenced in a program.
- Some can be referenced throughout a program, others from only portions of a program.
- An identifier's linkage determines, for a multiple-source-file program, whether the identifier is known only in the current source file or in any source file with proper declarations.

Storage Classes

- The four storage-class specifiers can be split into two storage durations: automatic storage duration and static storage duration.
- Keywords `auto` and `register` are used to declare variables of automatic storage duration.
- Variables with automatic storage duration are created when the block in which they're defined; they exist while the block is active, and they're destroyed when the block is exited.

Storage Classes

- Only variables can have automatic storage duration.
- A function's local variables (those declared in the parameter list or function body) normally have automatic storage duration.
- Keyword `auto` explicitly declares variables of automatic storage duration.

Storage Classes

- For example, the following declaration indicates that `double` variables `x` and `y` are automatic local variables and they exist only in the body of the function in which the declaration appears:

```
auto double x, y;
```

- Local variables have automatic storage duration by default, so keyword `auto` is rarely used.
- We'll refer to variables with automatic storage duration simply as automatic variables.

Performance Tip 5.1

Automatic storage is a means of conserving memory, because automatic variables exist only when they're needed. They're created when a function is entered and destroyed when the function is exited.

Software Engineering Observation 5.12

Automatic storage is an example of the **principle of least privilege**—*allowing access to data only when it's absolutely needed. Why have variables stored in memory and accessible when in fact they're not needed?*

Storage Classes

- Data in the machine-language version of a program is normally loaded into registers for calculations and other processing.

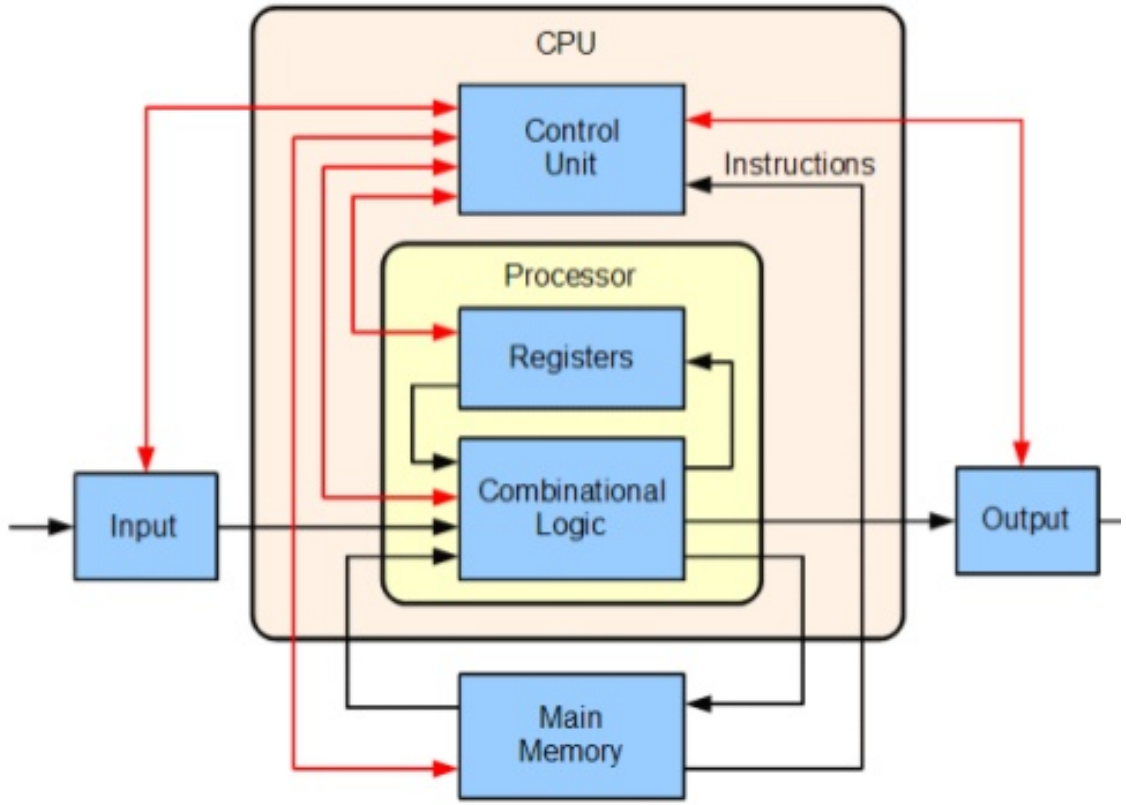

Fig: Building Blog Diagram of Computer

Performance Tip 5.2

The storage-class specifier `register` *can be placed before an automatic variable declaration to suggest that the compiler maintain the variable in one of the computer's high-speed hardware registers. If intensely used variables such as counters or totals can be maintained in hardware registers, the overhead of repeatedly loading the variables from memory into the registers and storing the results back into memory can be eliminated.*

Storage Classes

- The compiler may ignore `register` declarations.
- For example, there may not be a sufficient number of registers available for the compiler to use.
- The following declaration suggests that the integer variable `counter` be placed in one of the computer's registers and initialized to 1:
  - `register int counter = 1;`
- Keyword `register` can be used only with variables of automatic storage duration.

Performance Tip 5.3

Often, `register` *declarations are unnecessary. Today's optimizing compilers are capable of recognizing frequently used variables and can decide to place them in registers without the need for a* `register` *declaration.*

Storage Classes

- Keywords `extern` and `static` are used in the declarations of identifiers for variables of static storage duration.
- Identifiers of static storage duration exist from the time at which the program begins execution.
- Global variables and function names are of storage class `extern` by default.

Storage Classes

- Global variables are created by placing variable declarations outside any function definition, and they retain their values throughout the execution of the program.
- Global variables and functions can be referenced by any function that follows their declarations or definitions in the file.
- This is one reason for using function prototypes—when we include `stdio.h` in a program that calls `printf`, the function prototype is placed at the start of our file to make the name `printf` known to the rest of the file.

Software Engineering Observation 5.13

Defining a variable as global rather than local allows unintended side effects to occur when a function that does not need access to the variable accidentally or maliciously modifies it. In general, use of global variables should be avoided except in certain situations with unique performance requirements (as discussed in Chapter 14).

Software Engineering Observation 5.14

Variables used only in a particular function should be defined as local variables in that function rather than as external variables.

Storage Classes

- Local variables declared with the keyword `static` are still known only in the function in which they're defined, but unlike automatic variables, `static` local variables retain their value when the function is exited.
- The next time the function is called, the `static` local variable contains the value it had when the function last exited.

Storage Classes

- The following statement declares local variable `count` to be `static` and to be initialized to 1.
    - `static int count = 1;`
- All numeric variables of static storage duration are initialized to zero if you do not explicitly initialize them.

Scope Rules in C

Dr Bhanu

Scope Rules in C

- A scope in C is a region of the program where the defined variable can have its existence and beyond that, variable can't be accessed. These are the following three places to declare variables in C:
- Inside the function or in a block (called as local variables)
- In the definition of function parameters (formal parameters).
- Outside of all the functions (global variables).

Scope Rules in C

- Global (File) Scope
  - Any object defined in the global area of a program is visible until the end of the program.
- Local (Block)Scope
  - Variables defined within a block have local scope. They are invisible outside the block.
- Function Scope
  - Variables defined within a function have local scope. They are invisible outside the function.

Scope Rules in C

```
int a = 25;
printf("a  is %d\n", a);
```

```
printf("a  is %d\n", a);
int a = 25;
```

Variables are in scope from declaration until the
end of their block.

Scope Rules in C

```
int main()
{
   int a = 10;
   printf("a = %d\n", a);

   a = a + 5;
   printf("a = %d\n", a);

   return 0;
}
```

```
int main()
{
   {
    int a = 10;
   printf("a = %d\n", a);

   a = a + 5;
   }
   printf("a = %d\n", a);
   return 0;
}
```

Variables are in scope from declaration until the end of their block.

Use of Global Variables

```
#include <stdio.h>

int a = 10;
int b = 20;
void half(void);
void twice(void);

int main()
{
   printf("a is : %d\n", a);
   printf("b is : %d\n", b);

   half();
   twice();

   printf("a in main is : %d\n", a);
   printf("b in main is : %d\n", b);

   return 0;
}
```

```
void half(void)
  {
    int c, d;

    c = a/2;
    d = b/2;
  printf("c in function is : %d\n", c);
  printf("d in function is : %d\n", d);
  }

void twice(void)
  {
    a = a*2;
    b = b*2;
  }
```

- Distinction between local and global variables.
- Passing parameters to the function, using global variables.
- No parameters are passed through the functions and no value is returned to the main program.

Use of Keyword STATIC

```
#include <stdio.h>

int main()
{
    int sumofdigits(int);

   int a, b;

   a = sumofdigits(345);
   b = sumofdigits(345);

   printf("a = %d\n b = %d\n", a, b);

   return 0;
}
```

```
int sumofdigits(int n)
   {
      static int sum = 0;
      while (n > 0)
      {
         sum += (n % 10);
         n /= 10;
      }
      return sum;
   }
```

- Static keyword facilitates the previous value to be retained.

Use of Keyword STATIC

```
#include <stdio.h>

int main()
{

   for (int i = 1; i < 5; i++) {

      static int a = 5;

      int b = 10;
      a++;
      b++;
   printf("a = %d  b = %d\n", a, b);
   }
   return 0;
}
```