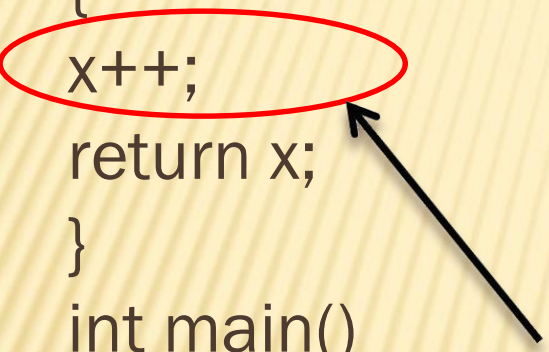
MISCELLANEOUS

QUICK EXERCISE

```
#include <stdio.h>
int add()
{
x++;
return x;
}
int main()
{
int x=5;
x = add();
printf("%d",x);
return 0;
}
```



Undefined variable “x”

Output:
Compiler error.

SCOPE OF A VARIABLE

- The scope of a variable is that region of the program where the variable can be accessed.
- The scope can be of two types:
  - Internal (Local):
    - Can only be accessed within a specific region i.e within { }.
    - Variable “x” in the previous program is a local variable.
    - Therefore, it cannot be accessed outside of main() function.
  - External (Global)
    - Can be accessed anywhere throughout the program.
    - Are defined outside of all blocks/functions
    - i.e outside any{ }.

QUICK EXERCISE

```
#include <stdio.h>
int main()
{
int x=5,y=10;
if(x<6)
{
printf("Inside if block y=%d\n",y);
}
printf("Inside function main() y=%d\n",y);
return 0;
}
```

Output:

Inside if block y=10

Inside function main() y=10

QUICK EXERCISE

```
#include <stdio.h>
int main()
{
int x=5,y=10;
if(x<6)
{
int y=12;
printf("Inside if block y=%d\n",y);
}
printf("Inside function main() y=%d\n",y);
return 0;
}
```

Output:

Inside if block y=12

Inside function main() y=10

QUICK EXERCISE

```
#include <stdio.h>
int z = 13;
int main()
{
int x=5,y=10;
if(x<6)
{
int y=12;
printf("Inside if block y = %d\n",y);
}
printf("Inside function main() y = %d\n",y);
printf("Inside function main() z = %d",z);
return 0;
}
```

Output:

Inside if block y = 12

Inside function main() y = 10

Inside function main() z = 13

QUICK EXERCISE

```
#include <stdio.h>
int y = 13;
int main()
{
int x=5,y=10;
if(x<6)
{
int y=12;
printf("Inside if block y = %d\n",y);
}
printf("Inside function main() y = %d\n",y);
return 0;
}
```

Output:
Inside if block y = 12
Inside function main() y = 10

When the local and the global variables have the same name, preference will always be given to the local variable.

STORAGE CLASSES

- Storage classes in C are special keywords.
- These keywords are placed BEFORE a variable declaration.
- Syntax:

  storage_class datatype variable_name;

- Storage classes mainly define the “extent” of a variable.
- The “extent” or the duration up to which a variable lives in the memory.
- There are four storage classes in C:
  - auto
  - static
  - extern
  - register

auto

- When no storage class is explicitly defined, the default storage class of a variable is *auto*.
- *auto* variables are basically normal local variables used in a program.
- They live in the memory ONLY until the block/function is in execution.
- Once the execution is over, they are destroyed from the memory.
- Their default value is Garbage.

static

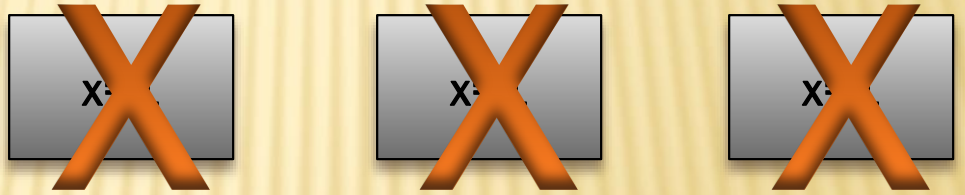
- The **static** storage class instructs the compiler to keep a local variable in existence throughout the life-time of the program.
- Instead of creating and destroying it each time it comes into and goes out of scope, they maintain their values between function calls.
- The static modifier may also be applied to global variables.
- These are called as external static variables.
- Internal static variables can be accessed only within the block/function.
- External static variables can be accessed throughout the program.
- The default initial value of static variables are 0.

QUICK EXERCISE

```
#include <stdio.h>
void func()
{
int x = 0;
x++;
printf("%d ",x);
}
int main()
{
func();
func();
func();
return 0;
}
```

Output:

1 1 1



QUICK EXERCISE

```
#include <stdio.h>
void func()
{
static int x = 0;
x++;
printf("%d ",x);
}
int main()
{
func();
func();
func();
return 0;
}
```

Output:

1 2 3

x = ~~1~~ ~~2~~ 3

QUICK EXERCISE

```
#include <stdio.h>
void func()
{
static int x = 0;
x++;
printf("%d ",x);
}
int main()
{
func();
func();
func();
printf("%d", x);
return 0;
}
```

Output:
Error.

As it is an internal static variable, the “scope” is still within the function only.

EXAMPLE

```
#include <stdio.h>
static int x = 4;
void func()
{
static int x = 0;
x++;
printf("%d ",x);
}
int main()
{
func();
func();
func();
printf("%d", x);
return 0;
}
```

External static variable

Internal static variable

Output:

1 2 3 4

extern

- The extern storage class is used to give a reference of a global variable that is visible to ALL the program files.
- When you use 'extern', the variable cannot be re-initialized.
- It can only refer the variable name at a storage location that has been previously defined.
- extern is used to declare a global variable or function in another file.
- The extern modifier is most commonly used when there are two or more files sharing the same global variables.

EXAMPLE

file1.c

```
#include <stdio.h>
#include “file2.h"
int count = 2;
int main()
{
increment();
printf("%d",count);
}
```

file2.h

```
extern int count;
void increment()
{
count++;
}
```

register

- Register variables tell the compiler to store the variable in CPU register instead of memory.
- Frequently used variables are kept in registers and they have faster accessibility.
- Can you guess what kind of variables?
  - Loop variables – for example
- We can never get the addresses of these variables using an “&”.
- “register” keyword is used to declare the register variables.
- They are always local to the block/function.
- Default initialized value is the garbage value.
- Lifetime is till the end of the execution of the block in which it is defined.

PREPROCESSOR DIRECTIVES

- Some lines of code in a program are “pre-processed” before they are compiled.
- These lines of code begin with a # symbol
  - Example: #include
- Any statement that begins with a # is called as a preprocessor directive.
- There are 4 main types of preprocessor directives:
  - Macros
  - File Inclusion
  - Conditional Compilation
  - Other directives

PREPROCESSOR DIRECTIVES-MACROS

- The '#define' directive is used to define a macro.
- Macros are a value/piece of code in a program which is given some name.
- Whenever this name is encountered by the compiler the compiler replaces the name with the actual value/piece of code.
- Macro names cannot be the same as variable names
- Syntax:

```
#define PI 3.14
#define MAX(a,b)        ((a < b) ?  (b) : (a))
```

PREPROCESSOR DIRECTIVES-MACROS

```
#include <stdio.h>
#define STRING1                 "A macro definition\n"
#define VALUE                   32
#define EXPRESSION1             1 + 2 + 3 + VALUE
#define EXPRESSION2             EXPRESSION1 + 10
#define ABS(x)                  (((x) < 0) ? -(x) : (x))
#define MAX(a,b)                ((a < b) ?  (b) : (a))
int main ()
{
  printf (STRING1);
  printf ("%d\n", EXPRESSION1);
  printf ("%d\n", EXPRESSION2);
  printf ("%d\n", MAX(ABS(-5),ABS(-8));
  return 0;
}
```

Output:

A macro definition

38

48

8

HOMEWORK

- Find out the advantages of using macros as symbolic constants and functions.
- What can be the disadvantages of using macro functions?

PREPROCESSOR DIRECTIVES-CONDITIONAL

- As the name implies, it is the process of selecting the source code conditionally from the program and sending to the compiler.
- They delimit blocks of program text that are compiled only if a specified condition is true.
- The program text within the blocks may consist of preprocessor directives, C statements, and so on.
- The beginning of the block of program text is marked by one of three directives:
  - #if
  - #ifdef
  - #ifndef
- Optionally, an alternative block of text can be set aside with one of two directives:
  - #else
  - #elif
- The end of the block or alternative block is marked by the #endif directive.

CONDITIONAL DIRECTIVES - EXAMPLE

```
#include <stdio.h>
#define MAX   8
int main()
{
  #if MAX < 12
  printf("Hello");
  #elif MAX < 7
  printf("Welcome")
  #endif
  #ifdef MAX
  printf("Hi");
  #endif
  #ifndef MIN
  printf("Bye");
  #endif
  return 0;
}
```

Output:
HelloHiBye

PREPROCESSOR DIRECTIVES

Preprocessor	Syntax/Description
Macro	**Syntax:** #define This macro defines constant value and can be any of the basic data types.
Header file inclusion	**Syntax:** #include <file_name> The source code of the file “file_name” is included in the main program at the specified place.
Conditional compilation	**Syntax:** #ifdef, #endif, #if, #else, #ifndef Set of commands are included or excluded in source program before compilation with respect to the condition.
Other directives	**Syntax:** #undef, #pragma #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program.

ENUMERATION IN C (SYMBOLIC CONSTANTS)

- Enumeration (or enum) is a user defined data type in C.
- It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

```
#include<stdio.h>
enum week {Mon, Tue, Wed, Thur, Fri, Sat, Sun}; <---- Declaration
int main()
{
    enum week day; <------ Instantiation
    day = Wed;
    printf("%d",day); ------>  output: 2
    return 0;
}
```

- If we do not assign values to enum constants, the default value starts with a 0
- We can manually assign values to the constants.

```
enum week{Mon, Tue=2, Wed=5, Thur, Fri, Sat, Sun};
```

QUICK EXERCISE

- Create a data type *boolean* which can either be a true or a false using enumerations. Create a variable of *boolean* called flag. Input a double variable d. Check if d is less than 0.00000005467 using scientific/exponential notation. If yes, make flag as *boolean* true else, false.

QUICK EXERCISE

- SOLUTION:

```
#include <stdio.h>
enum boolean {false, true};
int main()
{
enum boolean flag;
double d;
scanf("%ld", &d);
flag = d<5.467e-8? true : false;
printf("%d", flag);
return 0;
}
```

HOMEWORK

- Find and list down all similarities and differences between an enumeration and structure.
- Difference between a macro and enumeration.

COMMANDLINE ARGUMENTS

- Command line argument is a parameter supplied to the program when it is invoked.
- It is mostly used when you need to control your program from outside.
- Command line arguments are passed to the main() method.
- Syntax:

```
int main(int argc, char *argv[])
```

- Here
  - argc counts the number of arguments on the command line
  - argv[ ] is a pointer array which holds pointers of type char which points to the arguments passed to the program.

EXAMPLE

```
#include<stdio.h>
int main(int argc,char* argv[])
{
    int counter;
    printf("Program Name Is: %s",argv[0]);
    if(argc==1)
    printf("\nNo Command Line Argument Passed Other Than Program Name");
    if(argc>=2)
    {
        printf("\nNumber Of Arguments Passed: %d",argc);
        printf("\n----Following Are The Command Line Arguments Passed----");
        for(counter=0;counter<argc;counter++)
        printf("\nargv[%d]: %s",counter,argv[counter]);
    }
    return 0;
}
```

OUTPUT OF THE PREVIOUS PROGRAM

1. Without argument:

   When the above code is compiled and executed without passing any argument, it produces following output.

   $ ./a.out

   Program Name Is: ./a.out

   No Command Line Argument Passed Other Than Program Name

2. Three arguments :

   When the above code is compiled and executed with three arguments, it produces the following output.

   $ ./a.out First Second Third

   Program Name Is: ./a.out

   Number Of Arguments Passed: 4

   ----Following Are The Command Line Arguments Passed----

   argv[0]: ./a.out

   argv[1]: First

   argv[2]: Second

   argv[3]: Third