**Variables**

**Writer:** Sayak Haldar

**Extern Keyword:**

**The Difference Between Declaration and Definition:**

Declaration of a variable/function simply declares that the variable/function exists somewhere in the program but the memory is not allocated for them. But the declaration of a variable/function serves an important role. And that is the type of the variable/function. Therefore, when a variable is declared, the program knows the data type of that variable. In case of function declaration, the program knows what are the arguments to that functions, their data types, the order of arguments and the return type of the function. So that’s all about declaration. Coming to the definition, when we define a variable/function, apart from the role of declaration, it also allocates memory for that variable/function. Therefore, we can think of definition as a super set of declaration. (or declaration as a subset of definition). From this explanation, it should be obvious that a variable/function can be declared any number of times but it can be defined only once. (Remember the basic principle that you can’t have two locations of the same variable/function). So that’s all about declaration and definition.

Now, let’s clear about the doubts that whenever a function is defined, some memory is allocated:

**#include<stdio.h>**

**int foo()**

**{**

**return 5;**

**}**

**int main()**

**{**

**printf("The starting memory location of function foo %p\n",foo);**

**return 0;**

**}**

It will print a valid memory location.

Now, it that the location of the function foo? I can’t be sure.

**However, there is a way, right?**

Define a function pointer variable and let the function pointer points to that function. And try to print the address pointed by pointer.

**#include<stdio.h>**

**int foo()**

**{**

**return 5;**

**}**

**int main()**

**{**

**int (\*func\_ptr)();**

**//a function pointer func\_ptr is defined (not a type, rather a variable)**

**//which could point to a function which accepts no argument and returns int**

**func\_ptr=foo;**

**printf("The starting memory location of function foo %p\n",foo);**

**printf("The starting memory location of function foo %p\n",func\_ptr);**

**return 0;**

**}**

This will print:

**The starting memory location of function foo 0x4004c4**

**The starting memory location of function foo 0x4004c4**

Hence, my concept was correct.

**Also, memory of all function are allocated globally. (initialized data segment) And all functions are by default have external linkage. Now, to restrict that, (to make the function only available for current translation unit, we use static keyword in c)**

Now, coming back to the main objective. I.e. understanding extern.

Now, let’s start with the easy note. Let’s understand the use of extern with C functions. By default, the declaration and definition of a C function have “extern” prepended with them. It means even though we don’t use extern with the declaration/definition of C functions, it is present there. For example, when we write.

**int foo(int arg1, char arg2);**

There’s an extern present in the beginning which is hidden and the compiler treats it as below.

**extern int foo(int arg1, char arg2);**

Same is the case with the definition of a C function (Definition of a C function means writing the body of the function). Therefore whenever we define a C function, an extern is present there in the beginning of the function definition.

**Since the declaration can be done any number of times and definition can be done only once, we can notice that declaration of a function can be added in several C/H files or in a single C/H file several times. But we notice the actual definition of the function only once (i.e. in one file only).** That is why it is considered to define a function in the header file. **(because, it will cause linking error. Now, since, linking errors are not shown directly, it will be seen in compile time).** Now, that linking error problem could be solved with inline function concept in c.

And as the extern extends the visibility to the whole program, the functions can be used (called) anywhere in any of the files of the whole program provided the declaration of the function is known. (By knowing the declaration of the function, C compiler knows that the definition of the function exists and it goes ahead to compile the program). So that’s all about extern with C functions.

Now let us the take the second and final case i.e. use of extern with C variables. I feel that it more interesting and information than the previous case where extern is present by default with C functions. So let me ask the question, how would you declare a C variable without defining it? Many of you would see it trivial but it’s important question to understand extern with C variables. The answer goes as follows.

**extern int var;**

Here, an integer type variable called var has been declared (remember no definition i.e. no memory allocation for var so far). And we can do this declaration as many times as needed. (remember that declaration can be done any number of times) So far so good.

Now how would you define a variable. Now I agree that it is the most trivial question in programming and the answer is as follows.

**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.

**Static Keyword, Static Variable**

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\_name = var\_value;

Following are some interesting facts about static variables in C.

1. A static int variable remains in memory while the program is running. A normal or auto variable is destroyed when a function call where the variable was declared is over.

For example, we can use static int to count number of times a function is called, but an auto variable can’t be sued for this purpose.

For example below program prints “1 2”

**#include<stdio.h>**

**int fun()**

**{**

**static int count = 0;**

**count++;**

**return count;**

**}**

**int main()**

**{**

**printf("%d ", fun());**

**printf("%d ", fun());**

**return 0;**

**}**

**Output:**

1 2

But below program prints 1 1

**#include<stdio.h>**

**int fun()**

**{**

**int count = 0;**

**count++;**

**return count;**

**}**

**int main()**

**{**

**printf("%d ", fun());**

**printf("%d ", fun());**

**return 0;**

**}**

1. **Static variables are allocated memory in data segment, not stack segment. See [memory layout of C programs](https://www.geeksforgeeks.org/memory-layout-of-c-program/) for details.**
2. Static variables (like global variables) are initialized as 0 if not initialized explicitly. For example in the below program, value of x is printed as 0, while value of y is something garbage. See [this](https://www.geeksforgeeks.org/g-fact-53/)for more details.

|  |
| --- |
| **#include <stdio.h>**  **int main()**  **{**  **static int x;**  **int y;**  **printf("%d \n %d", x, y);**  **}**  (since, static variable’s memory is always allocated from data segment. It it is initialized explicitly, it’s memory is allocated from initialized data segment. In other cases, it will be uninitialized data segment or bss segment) |

1. In C, static variables can only be initialized using constant literals. For example, following program fails in compilation. See [this](https://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);**  **Gretchen();**  **return 0;**  **}**   1. Static global variables and functions are also possible in C/C++. The purpose of these is to limit scope of a variable or function to a file. |

**Register Variables:**Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using register keyword. The keyword register hints to compiler that a given variable can be put in a register. It’s compiler’s choice to put it in a register or not. Generally, compilers themselves do optimizations and put the variables in register.

1) If you use & operator with a register variable then compiler may give an error or warning (depending upon the compiler you are using), because when we say a variable is a register, it may be stored in a register instead of memory and accessing address of a register is invalid. Try below program.

**int main()**

**{**

**register int i = 10;**

**int \*a = &i;**

**printf("%d", \*a);**

**Gretchen();**

**return 0;**

**}**

2) register keyword can be used with pointer variables. Obviously, a register can have address of a memory location. There would not be any problem with the below program.

**int main()**

**{**

**int i = 10;**

**register int \*a = &i;**

**printf("%d", \*a);**

**Gretchen();**

**return 0;**

**}**

3) Register is a storage class, and C doesn’t allow multiple storage class specifiers for a variable. So, register can not be used with static . Try below program.

**int main()**

**{**

**int i = 10;**

**register static int \*a = &i;**

**printf("%d", \*a);**

**Gretchen();**

**return 0;**

**}**

1. There is no limit on number of register variables in a C program, but the point is compiler may put some variables in register and some not.

(i.e. it’s a request to the compiler. It is not guaranteed)

**Auto Keyword:**

auto: This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope).

**1. What is the output of this C code?**

#include <stdio.h>

void main()

{

m();

printf("%d", x);

}

int x;

void m()

{

x = 4;

}

a) 4

b) Compile time error

c) 0

d) Undefined

**Answer: b**

**Because, m() function call cannot find the definition of m.**

**And, during printf("%d", x); we cannot find the declaration of x.**

ctrial59.c: In function ‘main’:

ctrial59.c:5: error: ‘x’ undeclared (first use in this function)

ctrial59.c:5: error: (Each undeclared identifier is reported only once

ctrial59.c:5: error: for each function it appears in.)

ctrial59.c: At top level:

ctrial59.c:8: warning: conflicting types for ‘m’

ctrial59.c:4: note: previous implicit declaration of ‘m’ was here

**2. What is the output of this C code?**

#include <stdio.h>

int x;

void main()

{

printf("%d", x);

}

a) Junk value

b) Run time error

c) 0

d) Undefined

**Answer: c**

Now, int x; this declares and defines the variable. Now, defining means it’s memory is allocated. Now, here, x’s memory is allocated from uninitialized memory segment. Which is called bss segment.

**3. What is the output of this C code?**

#include <stdio.h>

int x = 5;

void main()

{

int x = 3;

printf("%d", x);

{

x = 4;

}

printf("%d", x);

}

a) Run time error

b) 3 3

c) 3 5

d) 3 4

**Answer: d**

**Because, the value is changed in x’s memory location**

However, the following would print 3 3

#include <stdio.h>

int x = 5;

void main()

{

int x = 3;

printf("%d", x);

{

int x = 4;

}

printf("%d", x);

}

**4. What is the output of this C code?**

#include <stdio.h>

int x = 5;

void main()

{

int x = 3;

printf("%d", x);

{

int x = 4;

}

printf("%d", x);

}

a) 3 3

b) 3 4

c) 3 5

d) Run time error

**Answer: a**

**5. Functions in C are ALWAYS:**

a) Internal

b) External

c) Both Internal and External

d) External and Internal are not valid terms for functions

**Answer: b**

By default, all functions have external linkage. Now, that means it is accessible from other translation unit. Now, to restrict it, we have to use static keyword.

**6. Global variables are:**

a) Internal

b) External

c) Both Internal and External

d) None of the mentioned

**Answer) b) External.**

Now, that is confusing, right?

For instance,

**extern int var;**

**int main(void)**

**{**

**var = 10;**

**return 0;**

**}**

This will generate compilation error.

While, the following is not.

**int var;**

**int main(void)**

**{**

**var = 10;**

**return 0;**

**}**

Now, you obviously know the reason. Because, in case of first program, memory was not allocated to var.

However, due to external linkage of a global variable you can do the following:

**Let it be ctrial62.c**

#include<stdio.h>

int x=0;

int main()

{

printf("The x's value in %s is %d\n",\_\_FILE\_\_,x);

foo();

return 0;

}

**And let it be ctrial63.c**

**#include<stdio.h>**

**extern int x;**

**void foo()**

**{**

**x=10;**

**printf("x's value in file %s is %d\n",\_\_FILE\_\_,x);**

**}**

Now, compile them together.

**It will print:**

The x's value in ctrial62.c is 0

x's value in file ctrial63.c is 10

However, the following would not work:

**ctrial62.c**

#include<stdio.h>

int main()

{

int x=0;

printf("The x's value in %s is %d\n",\_\_FILE\_\_,x);

foo();

return 0;

}

And, **ctrial63.c**

#include<stdio.h>

extern int x;

void foo()

{

x=10;

printf("x's value in file %s is %d\n",\_\_FILE\_\_,x);

}

**Compiling them combinedly will generate compilation error.**

**7. Which of the following are an external variable?**

#include <stdio.h>

int func (int a)

{

int b;

return b;

}

int main()

{

int c;

func (c);

}

int d;

a) a

b) b

c) c

d) d

**Answer: d**

**8. What will be the output?**

#include <stdio.h>

int main()

{

printf("%d", d++);

}

int d = 10;

a) 9

b) 10

c) 11

d) Compile time error

**Answer: d**

**9. What will be the output?**

#include <stdio.h>

double var = 8;

int main()

{

int var = 5;

printf("%d", var);

}

a) 5

b) 8

c) Compile time error due to wrong format identifier for double

d) Compile time error due to redeclaration of variable with same name

**Answer: a**

**10. What is the output of this C code?**

#include <stdio.h>

double i;

int main()

{

printf("%g\n",i);

return 0;

}

a) 0

b) 0.000000

c) Garbage value

d) Depends on the compiler

**Answer: a**

%g and %G are simplifiers of the scientific notation floats %e and %E.

%g will take a number that could be represented as %f (a simple float or double) or %e (scientific notation) and return it as the shorter of the two.

The output of your print statement will depend on the value of sum.

This has far been the most apt explanation I found.

Source: Stackoverflow

**Example 1:**

Value to printed :896.956

%.4f - 896.9560

%.2f - 896.96

%f - 896.956

%e - 8.96956e+2

As length of %e is greater than %f

%g - 896.956

**Example 2:**

Value to printed : 56000.0

%.4f - 56000.0000

%.2f - 56000.00

%f - 56000.0

%e - 5.6e+4

As length of %f is greater than %e

%g - 5.6e+4

**11. Which part of the program address space is p stored in the code given below?**

#include <stdio.h>

int \*p = NULL;

int main()

{

int i = 0;

p = &i;

return 0;

}

a) Code/text segment

b) Data segment

c) Bss segment

d) Stack

View Answer

**Answer: b**

Since, it is explicitly initialized to NULL, it will be allocated from **initialized data segment or data segment.**

**(bss segment is uninitialized data segment)**

**12. Which part of the program address space is p stored in the code given below?**

#include <stdio.h>

int \*p;

int main()

{

int i = 0;

p = &i;

return 0;

}

a) Code/text segment

b) Data segment

c) Bss segment

d) Stack

**Answer: c**

**13. Can variable i be accessed by functions in another source file?**

#include <stdio.h>

int i;

int main()

{

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

}

a) 0

b) false

c) Only if static keyword is used

d) Depends on the type of the variable

**Answer: a**

**14. Property of external variable to be accessed by any source file is called by C90 standard as**

a) external linkage

b) external scope

c) global scope

d) global linkage

**Answer: a**

**14. What is the output of this C code?**

#include <stdio.h>

int \*i;

int main()

{

if (i == NULL)

printf("true\n");

return 0;

}

a) true

b) true only if NULL value is 0

c) Compile time error

d) Nothing

**Answer: a**

it’s memory is allocated from uninitialized data segment (bss segment). Now, all variables whose memory are allocated from bss segment are automatically initialized to 0. Now, (void \*) 0 is basically NULL. Even int \*i=0; means i is pointing to NULL (though compiler will probably throw a typecast warning)

**15. What is the output of this C code?**

#include <stdio.h>

int \*i;

int main()

{

if (i == 0)

printf("true\n");

return 0;

}

a) true

b) true only if NULL value is 0

c) Compile time error

d) Nothing

**Answer: b**

**16. What is the output of this C code?**

#include <stdio.h>

static int x = 5;

void main()

{

x = 9;

{

int x = 4;

}

printf("%d", x);

}

a) 9

b) 4

c) 5

d) 0

**Answer: a**

1. **What is the output of this C code?**

#include <stdio.h>

void main()

{

m();

m();

}

void m()

{

static int x = 5;

x++;

printf("%d", x);

}

a) 6 7

b) 6 6

c) 5 5

d) 5 6

**Answer: a**

**18. What is the output of this C code?**

#include <stdio.h>

void main()

{

static int x;

printf("x is %d", x);

}

a) 0

b) 1

c) Junk value

d) Run time error

View Answer

**Answer: a**

(memory allocation from bss segment. Hence, implicitly initialized to 0)

**19. What is the output of this C code?**

#include <stdio.h>

static int x;

void main()

{

int x;

printf("x is %d", x);

}

a) 0

b) Junkvalue

c) Run time error

d) Nothing

**Answer: b**

Because, int x this definition of x shadows the definition of static int x in global scope. Now, the variable x whose scope is local to main, it’s memory is allocated from stack segment. Hence, it is uninitialized. Hence, **junk value is printed.**

**20. What is the output of this C code?**

#include <stdio.h>

void main()

{

static double x;

int x;

printf("x is %d", x);

}

a) Nothing

b) 0

c) Compile time error

d) Junkvalue

**Answer: c**

**21. What is the output of this C code?**

#include <stdio.h>

void main()

{

static int x;

if (x++ < 2)

main();

}

a) Infinite calls to main

b) Run time error

c) Varies

d) main is called twice

**Answer: d**

**22. Which of following is not accepted in C?**

a) static a = 10; //static as

b) static int func (int); //parameter as static

c) static static int a; //a static variable prefixed with static

d) all of the mentioned

**Answer: c**

**23. Which of the following cannot be static in C?**

a) Variables

b) Functions

c) Structures

d) None of the mentioned

Answer: d

**24.What is the output of code given below if these two files are linked and run?**

in file test.c

#include <stdio.h>

#include "test.h"

int main()

{

i = 10;

printf("%d ", i);

foo();

}

in file test1.c

#include <stdio.h>

#include "test.h"

int foo()

{

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

}

in file test.h

#include <stdio.h>

#include <stdlib.h>

static int i;

a) 1 0 0

b) 0 0

c) 10 10

d) None of the mentioned

**Answer: d**

Answer would be 10 0.

Now, why?

Because, a static global variable limits the scope of a variable to the translation unit. Now, test.c is a different translation unit, test1.c is a different translation unit. Hence,

Test.c will print 10 and test1.c will print 0

If you change the programs to the following:

In test.c

**#include <stdio.h>**

**#include "test.h"**

**int main()**

**{**

**i = 10;**

**printf("i\'s value in %s is %d\n",\_\_FILE\_\_,i);**

**foo();**

**}**

In test1.c

**#include <stdio.h>**

**#include "test.h"**

**int foo()**

**{**

**printf("i\'s value in %s in %d\n",\_\_FILE\_\_,i);**

**}**

In test.h

**#include <stdio.h>**

**#include <stdlib.h>**

**static int i;**

**Compiling them together will clearly print:**

i's value in test.c is 10

i's value in test1.c in 0

Now, normal global variables have external linkage. Hence, replacing the **static int i** with **int i** will print:

i's value in test.c is 10

i's value in test1.c in 10

**25. Functions have static qualifier for its declaration by default.**

a) true

b) false

c) Depends on the compiler

d) Depends on the standard

**Answer: b**

**26. Is initialization mandatory for local static variables?**

a) Yes

b) No

c) Depends on the compiler

d) Depends on the standard

**Answer: b (depending upon the initialization, it’s memory is allocated from uninitialized data segment or bss segment and initialized data segment)**

**27. What is the output of this C code?**

#include <stdio.h>

int main()

{

foo();

foo();

}

void foo()

{

int i = 11;

printf("%d ", i);

static int j = 12;

j = j + 1;

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

}

a) 11 12 11 12

b) 11 13 11 14

c) 11 12 11 13

d) Compile time error

**Answer: b**

**28. Assignment statements assigning value to local static variables are executed only once**

a) true

b) false

c) Depends on the code

d) None of the mentioned

**Answer: b**

**25. What is the format identifier for “static a = 20.5;”?**

a) %s

b) %d

c) %f

d) Illegal declaration due to absence of data type

**Answer: b**

**26. Which of the following is true for static variable?**

a) It can be called from another function.

b) It exists even after the function ends.

c) It can be modified in another function by sending it as a parameter.

d) All of the mentioned

View Answer

**Answer: b**

**(it cannot be accessed from another function)**

**27. Comment on the output of this C code?**

#include <stdio.h>

void func();

int main()

{

static int b = 20;

func();

}

void func()

{

static int b;

printf("%d", b);

}

a) Output will be 0

b) Output will be 20

c) Output will be a garbage value

d) Compile time error due to redeclaration of static variable

**Answer: a**

**28. What is the output of this C code?**

#include <stdio.h>

int main()

{

register int i = 10;

int \*p = &i;

\*p = 11;

printf("%d %d\n", i, \*p);

}

a) Depends on whether i is actually stored in machine register

b) 10 10

c) 11 11

d) Compile time error

**Answer: d**

Reference operator cannot be used with register variable. Depending on the compiler standard, it will result warning or compilation error.

**29. register keyword mandates compiler to place it in machine register.**

a) true

b) false

c) Depends on the standard

d) None of the mentioned

**Answer: b**

It is a request to the compiler.

**30. What is the output of this C code?**

#include <stdio.h>

int main()

{

register static int i = 10;

i = 11;

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

}

**a) 10**

**b) Compile time error**

**c) Undefined behaviour**

**d) 11**

Answer: b

Explanation: None.

**31. What is the output of this C code?**

**#include <stdio.h>**

**int main()**

**{**

**register auto int i = 10;**

**i = 11;**

**printf("%d\n", i);**

**}**

a) 10

b) Compile time error

c) Undefined behaviour

d) 11

**Answer: b**

Now, auto and register both are storage class specifiers. Hence, both cannot be applied simultaneously.

**32. What is the output of this C code?**

#include <stdio.h>

int main()

{

register const int i = 10;

i = 11;

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

}

a) 10

b) Compile time error

c) Undefined behaviour

d) 11

**Answer: b**

Now, const is type qualifier whereas register is storage class specifier. Now, both can be applied simultaneously. However, a const variable’s value cannot be changed.

**33. Register storage class can be specified to global variables**

a) true

b) false

c) Depends on the compiler

d) Depends on the standard

**Answer: b**

It will result in compilation error. Since, both are storage class specifiers. And, both cannot be applied simultaneously to a variable

**34. Which among the following is wrong for “register int a;” ?**

a) Compiler generally ignores the request.

b) You cannot take the address of this variable

c) Access time to a is critical

d) None of the mentioned

**Answer: d**

**35. What is the output of this C code?**

#include <stdio.h>

void main()

{

register int x = 5;

m();

printf("x is %d", x);

}

void m()

{

x++;

}

a) 6

b) 5

c) Junk value

d) Compile time error

**Answer: d**

Scoping.

**36.When compiler accepts the request to use the variable as a register?**

a) It is stored in CPU

b) It is stored in cache memory

c) It is stored in main memory

d) It is stored in secondary memory

**Answer: a**

**37. Which data type can be stored in register?**

a) int

b) long

c) float

d) all of the mentioned

**Answer: d**

**38. Which of the following operation is not possible in a register variable?**

a) Reading the value into a register variable

b) Copy the value from a memory variable

c) Global declaration of register variable

d) All of the mentioned

**Answer: d**

**39. Which among the following is the correct syntax to declare a static variable register?**

a) static register a;

b) register static a;

c) Both static register a; and register static a;

d) We cannot use static and register together

**Answer: d**

Because, both are storage class specifiers.

**40. Register variables reside in**

a) stack

b) registers

c) heap

d) main memory

**Answer: b**

CPU registers.

**41. What is the output of this C code?**

#include <stdio.h>

void main()

{

register int x = 0;

if (x < 2)

{

x++;

main();

}

}

a) Segmentation fault

b) main is called twice

c) main is called once

d) main is called thrice

**Answer: a**

Main would be printed twice if int x’s storage class specifier was static.

**42. What is the output of this C code?**

#include <stdio.h>

void main()

{

register int x;

printf("%d", x);

}

a) 0

b) Junk value

c) Compile time error

d) Nothing

**Answer: b**

**43. What is the output of this C code?**

#include <stdio.h>

register int x;

void main()

{

printf("%d", x);

}

a) Varies

b) 0

c) Junk value

d) Compile time error

**Answer: d**

Because, a register variable cannot be global. Both are storage class specifier.

**44. The scope of an automatic variable is:**

a) Within the block it appears

b) Within the blocks of the block it appears

c) Until the end of program

d) Within the block it appears & Within the blocks of the block it appears

**Answer: d**

**45. Automatic variables are allocated space in the form of a:**

a) stack

b) queue

c) priority queue

d) random

**Answer: a**

Memory is allocated from stack segment.

**46. Which of the following is a storage specifier?**

a) enum

b) union

c) auto

d) volatile

**Answer: c**

Enum and union are user defined types. Auto is a storage specifier. Volatile is a type qualifier.

(unsigned, signed, long long etc are type modifiers)

**47. Default storage class if not any is specified for a local variable, is auto**

a) true

b) false

c) Depends on the standard

d) None of the mentioned

**Answer: a**

**48. What is the output of this C code?**

#include <stdio.h>

void foo(auto int i);

int main()

{

foo(10);

}

void foo(auto int i)

{

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

}

a) 10

b) Compile time error

c) Depends on the standard

d) None of the mentioned

**Answer: b**

**49. Automatic variables are stored in**

a) stack

b) data segment

c) register

d) heap

**Answer: a**

**50. What linkage does automatic variables have?**

a) Internal linkage

b) External linkage

c) No linkage

d) None of the mentioned

**Answer: c**

**51. What is the output of this C code?**

#include <stdio.h>

int main()

{

auto i = 10;

const auto int \*p = &i;

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

}

a) 10

b) Compile time error

c) Depends on the standard

d) Depends on the compiler

**Answer: a**

First, const is type qualifier. (const and volatile) and auto is storage specifier.

Second, the value stored at the memory location pointed by p cannot be changed.

**52. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x;

}

here x is

a) automatic variable

b) static variable

c) register variable

d) global variable.

Answer) is a)

**53. Automatic variables are initialised to**

a) Zero

b) Junk value

c) Nothing

d) Both Zero & Junk value

**Answer) b)**

**54. Which of the following storage class supports char data type?**

a) register

b) static

c) auto

d) all of the mentioned

**Answer) d) all of the mentioned**

**55. The variable declaration with no storage class specified is by default:**

a) auto

b) extern

c) static

d) register

**Answer) a) auto**

**56.What is the output of this C code?**

#include <stdio.h>

int i;

int main()

{

extern int i;

if (i == 0)

printf("scope rules\n");

}

a) scope rules

b) Compile time error due to multiple declaration

c) Compile time error due to not defining type in statement extern i

d) Nothing as i value is not zero being automatic variable

**Answer: a**

Because, extern int a declares a variable does not allocate memory for it.

Also, even if, the program is recompiled with the following code:

**#include <stdio.h>**

**int i;**

**int main()**

**{**

**int i;**

**if (i == 0)**

**printf("scope rules\n");**

**}**

It wont generate compilation error. Because, the i, declared and defined with in the local scope, will shadow the I declared and defined in global scope.

**57. What is the output of this C code (without linking the source file in which ary1 is defined)?**

#include <stdio.h>

int main()

{

extern ary1[];

printf("scope rules\n");

}

a) scope rules

b) Linking error due to undefined reference

c) Compile time error because size of array is not provided

d) Compile time error because datatype of array is not provided

**Answer: a**

Since, ary1[] is just declared it can be done without specifying it’s size and datatype.

**58. What is the output of this C code after linking with source file having definition of ary1?**

#include <stdio.h>

int main()

{

extern ary1[];

printf("%d\n", ary1[0]);

}

a) Value of ary1[0]

b) Compile time error due to multiple definition

c) Compile time error because size of array is not provided

d) Compile time error because datatype of array is not provided

**Answer: d**

**59. What is the scope of an external variable?**

a) Whole source file in which it is defined

b) From the point of declaration to the end of the file in which it is defined

c) Any source file in a program

d) From the point of declaration to the end of the file being compiled

View Answer

**Answer: d**

**60. What is the scope of a function?**

a) Whole source file in which it is defined

b) From the point of declaration to the end of the file in which it is defined

c) Any source file in a program

d) From the point of declaration to the end of the file being compiled

**Answer: d**

**61. Comment on the output of this C code?**

#include <stdio.h>

int main()

{

int i;

for (i = 0;i < 5; i++)

int a = i;

printf("%d", a);

}

**a) a is out of scope when printf is called**

**b) Redeclaration of a in same scope throws error**

**c) Syntax error in declaration of a**

**d) No errors, program will show the output 5**

**Answer: c**

**62. Which variable has the longest scope?**

#include <stdio.h>

int b;

int main()

{

int c;

return 0;

}

int a;

a) a

b) b

c) c

d) Both a and b

**Answer: b**

**63.The sequence of allocation and deletion of variables for the following code is.**

#include <stdio.h>

int main()

{

int a;

{

int b;

}

}

a) a->b, a->b

b) a->b, b->a

c) b->a, a->b

d) b->a, b->a

**Answer: b**

**64. Array sizes are optional during array declaration by using \_\_\_\_\_\_ keyword.**

a) auto

b) static

c) extern

d) register

**Answer: c**

**65. What is the output of this C code?**

#include <stdio.h>

void main()

{

int x = 3;

{

x = 4;

printf("%d", x);

}

}

a) 4

b) 3

c) 0

d) Undefined

**Answer: a**x’s value is changed in the memory location.

**66. What is the output of this C code?**

#include <stdio.h>

int x = 5;

void main()

{

int x = 3;

m();

printf("%d", x);

}

void m()

{

x = 8;

n();

}

void n()

{

printf("%d", x);

}

a) 8 3

b) 3 8

c) 8 5

d) 5 3

**Answer: a**

**67. What is the output of this C code?**

#include <stdio.h>

int x;

void main()

{

m();

printf("%d", x);

}

void m()

{

x = 4;

}

a) 0

b) 4

c) Compile time error

d) Undefined

**Answer: b**

**68. What is the output of this C code?**

#include <stdio.h>

static int x = 5;

void main()

{

int x = 9;

{

x = 4;

}

printf("%d", x);

}

a) 9

b) 5

c) 4

d) 0

**Answer: c**

**69. What is the output of this C code?**

#include <stdio.h>

void main()

{

{

int x = 8;

}

printf("%d", x);

}

a) 8

b) 0

c) Undefined

d) Compile time error

**Answer: d**