To:
Tenouk

MODULE Z
THE C STORAGE CLASSES, SCOPE AND MEMORY ALLOCATION 1

My Training Period: zz hours

Note: **gcc** compilation examples are given at the end of this Module.

C abilities that supposed to be acquired:

- Understand and use the auto, register, extern and static keywords.
- Understand the basic of the **process address space**.
- Understand and appreciate the static, automatic and dynamic memory allocations.
- Understand how the memory is laid out for a running process.
- Understand and use the malloc(), calloc(), realloc() and free() functions.

Z.1 INTRODUCTION

- The storage class determines the part of memory where storage is allocated for an object (particularly variables and functions) and how long the storage allocation continues to exist.
- A scope specifies the part of the program which a variable name is visible, that is the accessibility of the variable by its name. In C program, there are four storage classes: automatic, register, external, and static.
- Keep in mind that in the hardware terms we have primary storage such as registers, cache, memory (Random Access Memory) and secondary storage such as magnetic and optical disk.

Z.1.1 AUTOMATIC VARIABLE - auto

- They are declared at the start of a program's block such as in the curly braces ({ }). Memory is allocated automatically upon entry to a block and freed automatically upon exit from the block.
- The scope of automatic variables is local to the block in which they are declared, including any blocks nested within that block. For these reasons, they are also called **local variables**.
- No block outside the defining block may have direct access to automatic variables (by variable name) but, they may be accessed indirectly by other blocks and/or functions using pointers.
- Automatic variables may be specified upon declaration to be of storage class auto. However, it is not required to use the keyword auto because **by default**, storage class within a block is auto.
- Automatic variables declared with initializers are initialized every time the block in which they are declared is entered or accessed.

Z.1.2 REGISTER VARIABLE - register

- Automatic variables are allocated storage in the main memory of the computer; however, for most computers, accessing data in memory is considerably slower than processing directly in the CPU.
- Registers are memory located within the CPU itself where data can be stored and accessed quickly. Normally, the **compiler determines** what data is to be stored in the registers of the CPU at what times.
- However, the C language provides the storage class register so that the programmer can suggest to the compiler that particular automatic variables should be allocated to CPU registers, if possible and it is not an obligation for the CPU to do this.
- Thus, register variables provide a certain control over efficiency of program execution.
- Variables which are used repeatedly or whose access times are critical may be declared to be of storage class register.
- Variables can be declared as a register as follows:

```
register int var;
```

Z.1.3 EXTERNAL VARIABLE - extern

- All variables we have seen so far have had limited scope (the block in which they are declared) and limited lifetimes (as for automatic variables).
- However, in some applications it may be useful to have data which is accessible from within any block and/or which remains in existence for the entire execution of the program. Such variables are called **global variables**, and the C language provides storage classes which can meet these requirements; namely, the **external** (extern) and **static** (static) classes.
- Declaration for external variable is as follows:

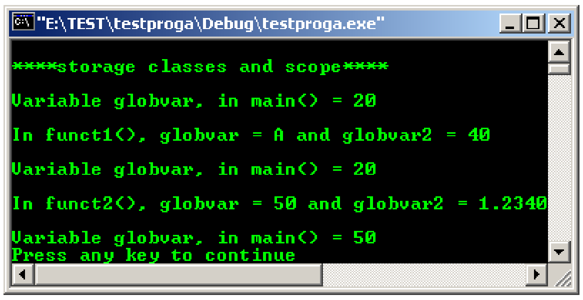
```
extern int var;
```

- External variables may be declared outside any function block in a source code file the same way any other variable is declared; by specifying its type and name (extern keyword may be omitted).
- Typically if declared and defined at the beginning of a source file, the extern keyword can be omitted. If the program is in several source files, and a variable is defined in let say **file1.c** and used in **file2.c** and **file3.c** then the extern keyword must be used in **file2.c** and **file3.c**.
- But, usual practice is to collect extern declarations of variables and functions in a separate header file (**.h** file) then included by using #include directive.
- Memory for such variables is allocated when the program begins execution, and remains allocated until the program terminates. For most C implementations, every byte of memory allocated for an external variable is **initialized to zero**.

- The scope of external variables is **global**, i.e. the entire source code in the file following the declarations. All functions following the declaration may access the external variable by using its name. However, if a local variable having the same name is declared within a function, references to the name will access the local variable cell.
- The following program example demonstrates storage classes and scope.

```
/* storage class and scope */
#include <stdio.h>

void funct1(void);
void funct2(void);

/* external variable, scope is global to main(), funct1()
and funct2(), extern keyword is omitted here, coz just one file */
int globvar = 10;

int main()
{
    printf("\n****storage classes and scope****\n");
    /* external variable */
    globvar = 20;

    printf("\nVariable globvar, in main() = %d\n", globvar);
    funct1();
    printf("\nVariable globvar, in main() = %d\n", globvar);
    funct2();
    printf("\nVariable globvar, in main() = %d\n", globvar);
    return 0;
}

/* external variable, scope is global to funct1() and funct2() */
int globvar2 = 30;

void funct1(void)
{
    /* auto variable, scope local to funct1() and funct1()
    cannot access the external globvar */
    char globvar;

    /* local variable to funct1() */
    globvar = 'A';
    /* external variable */
    globvar2 = 40;

    printf("\nIn funct1(), globvar = %c and globvar2 = %d\n", globvar, globvar2);
}

void funct2(void)
{
    /* auto variable, scope local to funct2(), and funct2()
    cannot access the external globvar2 */
    double globvar2;
    /* external variable */
    globvar =  50;
    /* auto local variable to funct2() */
    globvar2 = 1.234;
    printf("\nIn funct2(), globvar = %d and globvar2 = %.4f\n", globvar, globvar2);
}
```

Output:



```
****storage classes and scope****

Variable globvar, in main() = 20

In funct1(), globvar = A and globvar2 = 40

Variable globvar, in main() = 20

In funct2(), globvar = 50 and globvar2 = 1.2340

Variable globvar, in main() = 50
Press any key to continue
```

- External variables may be initialized in declarations just as automatic variables; however, the initializers must be **constant expressions**. The initialization is done only once at compile time, i. e. when memory is allocated for the variables.
- In general, it is a good programming practice to avoid using external variables as they destroy the concept of a function as a 'black box' or independent module.
- The black box concept is essential to the development of a modular program with modules. With an external variable, any function in the program can access and alter the variable, thus making debugging more difficult as well. This is not to say that external variables should never be used.
- There may be occasions when the use of an external variable significantly simplifies the implementation of an algorithm. Suffice it to say that external variables should be used rarely and with caution.

Z.1.4 STATIC VARIABLE - static

- As we have seen, external variables have global scope across the entire program (provided extern declarations are used in files other than where the variable is defined), and have a lifetime over the entire program run.

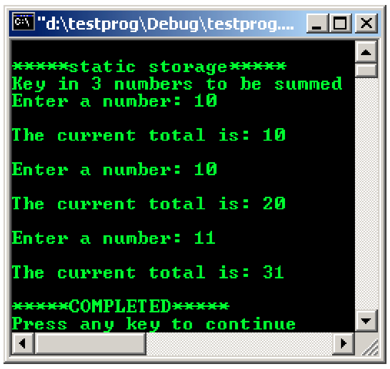
- Similarly, static storage class provides a lifetime over the entire program, however; it provides a way to limit the scope of such variables, and static storage class is declared with the keyword static as the class specifier when the variable is defined.
- These variables are **automatically initialized to zero** upon memory allocation just as external variables are. Static storage class can be specified for automatic as well as external variables such as:

```
static extern varx;
```

- Static automatic variables continue to exist even after the block in which they are defined terminates. Thus, the value of a static variable in a function is retained between repeated function calls to the same function.
- The scope of static automatic variables is identical to that of automatic variables, i.e. it is local to the block in which it is defined; however, the storage allocated becomes permanent for the duration of the program.
- Static variables may be initialized in their declarations; however, the initializers must be **constant expressions**, and initialization is done only once at compile time when memory is allocated for the static variable.

```
/* static storage class program example */
#include <stdio.h>
#define MAXNUM 3

void sum_up(void);

int main()
{
   int count;

   printf("\n*****static storage*****\n");
   printf("Key in 3 numbers to be summed ");
   for(count = 0; count < MAXNUM; count++)
      sum_up();
   printf("\n*****COMPLETED*****\n");
   return 0;
}

void sum_up(void)
{
   /* at compile time, sum is initialized to 0 */
   static int sum = 0;
   int num;

   printf("\nEnter a number: ");
   scanf("%d", &num);
   sum += num;
   printf("\nThe current total is: %d\n", sum);
}
```

Output:



```
*****static storage*****
Key in 3 numbers to be summed
Enter a number: 10

The current total is: 10

Enter a number: 10

The current total is: 20

Enter a number: 11

The current total is: 31

*****COMPLETED*****
Press any key to continue
```

- While the static variable, sum, would be automatically initialized to zero, it is better to do so explicitly.
- In any case, the initialization is performed only once at the time of memory allocation by the compiler. The variable sum retains its value during program execution.
- Each time the sum_up() function is called, sum is incremented by the next integer read. To see the different you can remove the static keyword, re-compile and re-run the program.

Z.2 DYNAMIC MEMORY ALLOCATION

- In the previous section we have described the storage classes which determined how memory for variables is allocated by the compiler.
- When a variable is defined in the source program, the **type of the variable** determines how much memory the compiler allocates.
- When the program executes, the variable consumes this amount of memory regardless of whether the program actually uses the memory allocated. This is particularly true for arrays.
- However, in many situations, it is not clear how much memory the program will actually need. For example, we may have declared arrays to be large enough to hold the maximum number of elements we expect our application to handle.
- If too much memory is allocated and then not used, there is a waste of memory. If not enough memory is allocated, the program is not able to fully handle the input data.
- We can make our program more flexible if, during execution, it could allocate initial and

additional memory when needed and free up the memory when it is no more needed.
- Allocation of memory during execution is called **dynamic memory allocation**. C provides library functions to allocate and free up memory dynamically during program execution. Dynamic memory is **allocated on the heap** by the system.
- It is important to realize that dynamic memory allocation also has limits. If memory is repeatedly allocated, eventually the system will run out of memory.

Z.3 PROCESS MEMORY LAYOUT

- A running program is called a **process** and when a program is run, its executable image is loaded into memory area that normally called a process address space in an organized manner.
- This is a physical memory space and do not confuse yourself with the virtual address space explained in Module W.
- Process address space is organized into three memory areas, called **segments**: the **text** segment, **stack** segment, and **data** segment (bss and data) and can be illustrated below.

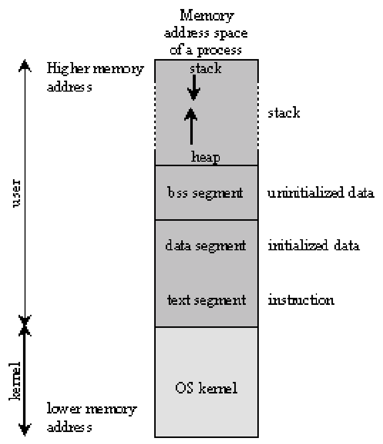


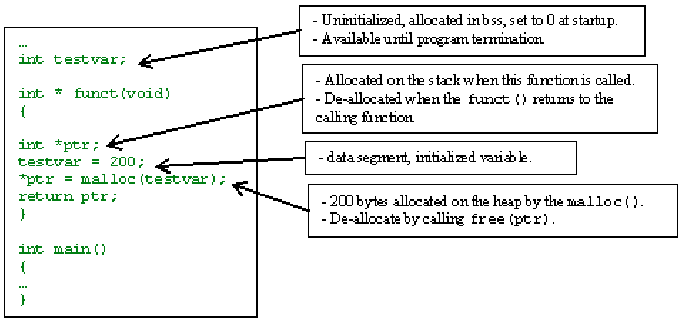
Figure: z.1

- The text segment (also called a code segment) is where the compiled code of the program itself resides.
- The following Table summarizes the segments in the memory address space layout as illustrated in the previous Figure.

Segment	Description
Code - text segment	Often referred to as the **text segment**, this is the area in which the executable or binary image instructions reside. For example, Linux/Unix arranges things so that multiple running instances of the same program share their code if possible. Only one copy of the instructions for the same program resides in memory at any time. The portion of the executable file containing the text segment is the **text section**.
Initialized data – data segment	Statically allocated and global data that are **initialized with nonzero values** live in the **data segment**. Each process running the same program has its own data segment. The portion of the executable file containing the data segment is the **data section**.
Uninitialized data – bss segment	BSS stands for ‘Block Started by Symbol’. Global and statically allocated data that **initialized to zero by default** are kept in what is called the BSS area of the process. Each process running the same program has its own BSS area. When running, the BSS, data are placed in the data segment. In the executable file, they are stored in the **BSS section**. For Linux/Unix the format of an executable, only variables that are initialized to a nonzero value occupy space in the executable’s disk file.
Heap	The heap is where dynamic memory (obtained by malloc(), calloc(), realloc() and new – C++) comes from. Everything on a heap is anonymous, thus you can only access parts of it through a pointer. As memory is allocated on the heap, the process’s address space grows. Although it is possible to give memory back to the system and shrink a process’s address space, this is almost never done because it will be allocated to other process again. Freed memory (free() and delete – C++) goes back to the heap, creating what is called holes. It is typical for the heap to grow upward. This means that successive items that are added to the heap are added at addresses that are numerically greater than previous items. It is also typical for the heap to start immediately after the BSS area of the data segment. The end of the heap is marked by a pointer known as the break. You cannot reference past the break. You can, however, move the break pointer (via brk() and sbrk() system calls) to a new position to increase the amount of heap memory available.
Stack	The stack segment is where local (automatic) variables are allocated. In C program, local variables are all variables declared inside the opening left curly brace of a **function's body** including the main() or other left curly brace that aren’t defined as static. The data is **popped** up or **pushed** into the stack following the **Last In First Out** (LIFO) rule. The stack holds local variables, temporary information, function parameters, return address and the like. When a function is called, a **stack frame** (or a procedure activation record) is created and **PUSH**ed onto the top of the stack. This stack frame contains information such as the address from which the function was called and where to jump back to when the function is finished (return address), parameters, local variables, and any other information needed by the invoked function. The order of the information may vary by system and compiler. When a function returns, the stack frame is **POP**ped from the stack. Typically the stack grows downward, meaning that items deeper in the call chain are at numerically lower addresses and toward the heap.

Table z.1

- In the disk file (object files) the **segments** were called **sections**.
- By using a C program, the segments can be illustrated below.



```
…
int testvar;

int * funct(void)
{

int *ptr;
testvar = 200;
*ptr = malloc(testvar);
return ptr;
}

int main()
{
…
}
```

- Uninitialized, allocated in bss, set to 0 at startup.
- Available until program termination.
- Allocated on the stack when this function is called.
- De-allocated when the funct() returns to the calling function.
- data segment, initialized variable.
- 200 bytes allocated on the heap by the malloc().
- De-allocate by calling free(ptr).

Figure z.2

Z.4 Some Terms

- In C language memory allocation through the variables in C programs is supported by two kinds of memory allocation as listed in the following Table.

Memory Allocation Type	Description
Static allocation	This allocation happens when you declare a **static** or **global** variable. Each static or global variable defines one block of space, of a fixed size. The space is allocated once, when your program is started, and is never freed. In memory address space, for uninitialized variables are stored in bss segment while an initialized variables stored in data segment.
Automatic allocation	This allocation happens when you declare an automatic variable, such as a function argument or a local variable. The space for an automatic variable is allocated when the compound statement containing the declaration is entered, and is freed when that compound statement is exited. As discussed before this allocation done in the stack segment.

Table z.2

- Dynamic allocation is not supported by C variables; there is no storage class called 'dynamic', and there can never be a C variable whose value is stored in dynamically allocated space. All must be done manually using related functions.
- The only way to refer to dynamically allocated space is **through a pointer**. Because it is less convenient, and because the actual process of dynamic allocation requires more computation time, programmers generally use dynamic allocation only when neither static nor automatic allocation will serve.
- The dynamic allocation done by using functions and the memory used is heap area.

Z.5 STACK AND HEAP

- The stack is where memory is allocated for automatic variables within functions. A stack is a Last In First Out (LIFO) storage where new storage is allocated and de-allocated at only one end, called the top of the stack. Every function call will create a stack (normally called stack frame) and when the function exit, the stack frame will be destroyed.
- By referring the following program example and Figure z.3, when a program begins execution in the function main(), stack frame is created, space is allocated on the stack for all variables declared within main().
- Then, when main() calls a function, a(), new stack frame is created for the variables in a() at the top of the main() stack. Any parameters passed by main() to a() are stored on this stack.
- If a() were to call any additional functions such as b() and c(), new stack frames would be allocated at the new top of the stack. Notice that the order of the execution happened in the sequence.
- When c(), b() and a() return, storage for their local variables are de-allocated, the stack frames are destroyed and the top of the stack returns to the previous condition. The order of the execution is in the reverse.
- As can be seen, the memory allocated in the stack area is used and reused during program execution. It should be clear that memory allocated in this area will contain garbage values left over from previous usage.

```
#include <stdio.h>

int a();
int b();
int c();

int a()
{
        b();
        c();
        return 0;
}

int b()
{ return 0; }

int c()
{ return 0; }

int main()
{
        a();
        return 0;
}
```

- By taking just the stack area, the following Figure illustrates what had happened to the stack when the above program is run. At the end there should be equilibrium.

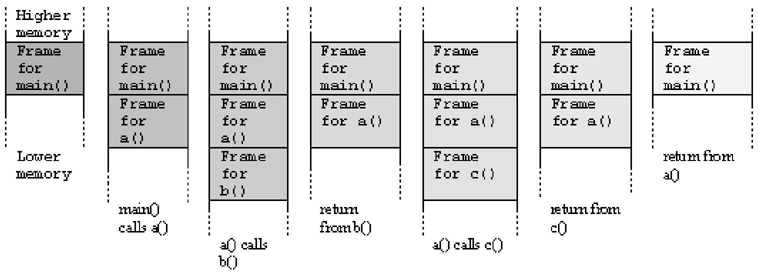


Figure z.3: Stack frame and function call

Z.5.1 FUNCTION CALLING CONVENTION

- It has been said before, for every function call there will be a creation of a stack frame. It is very useful if we can study the operation of the function call and how the stack frame for function is constructed and destroyed.
- For function call, compilers have some convention used for calling them. A convention is a way of doing things that is standardized, but not a documented standard.
- For example, the C/C++ function calling convention tells the compiler things such as:

  1. The order in which function arguments are pushed onto the stack.
  2. Whether the caller function or called function (callee) responsibility to remove the arguments from the stack at the end of the call that is the stack cleanup process.
  3. The name-decorating convention that the compiler uses to identify individual functions.

- Examples for calling conventions are __stdcall, __pascal, __cdecl and __fastcall (for Microsoft Visual C++).
- The calling convention belongs to a function's signature, thus functions with different calling convention are incompatible with each other.
- There is currently no standard for C/C++ naming between compiler vendors or even between different versions of a compiler for function calling scheme.
- That is why if you link object files compiled with other compilers may not produce the same naming scheme and thus causes unresolved externals. For Borland and Microsoft compilers you specify a specific calling convention between the return type and the function's name as shown below.

```
void __cdecl TestFunc(float a, char b, char c);   // Borland and Microsoft
```

- For the GNU GCC you use the __attribute__ keyword by writing the function definition followed by the keyword __attribute__ and then state the calling convention in double parentheses as shown below.

```
void  TestFunc(float a, char b, char c)  __attribute__((cdecl));  // GNU GCC
```

- As an example, Microsoft Visual C++ compiler has three function calling conventions used as listed in the following table.

keyword	Stack cleanup	Parameter passing
__cdecl	caller	Pushes parameters on the stack, in reverse order (right to left). Caller cleans up the stack. This is the default calling convention for C language that supports variadic functions (variable number of argument or type list such as printf()) and also C++ programs. The cdecl calling convention creates larger executables than __stdcall, because it requires each function call (caller) to include stack cleanup code.
__stdcall	callee	Also known as __pascal. Pushes parameters on the stack, in reverse order (right to left). Functions that use this calling convention require a function prototype. Callee cleans up the stack. It is standard convention used in Win32 API functions.
__fastcall	callee	Parameters stored in registers, then pushed on stack. The fastcall calling convention specifies that arguments to functions are to be passed in registers, when possible. Callee cleans up the stack.

Table z.3: Function calling conventions

- Basically, C function calls are made with the caller pushing some parameters onto the stack, calling the function and then popping the stack to clean up those pushed arguments. For __cdecl assembly example:

```
/* example of __cdecl */
push arg1
push arg2
call function
add ebp, 12   ;stack cleanup
```

- And for __stdcall example:

```
/* example of __stdcall */
push arg1
push arg2
call function
/* no stack cleanup, it will be done by callee */
```

- It is a long story if we want to go into the details of the function calls, but this section provides a good introduction :o).

Z.5.2 DYNAMIC ALLOCATION – THE FUNCTIONS

- The heap segment provides more stable storage of data for a program; memory allocated in the heap remains in existence for the duration of a program.
- Therefore, global variables (external storage class), and static variables are allocated on the heap. The memory allocated in the heap area, if initialized to zero at program start, remains zero until the program makes use of it. Thus, the heap area need not contain garbage.
- In ANSI C (ISO/IEC C), there is a family of four functions which allow programs to dynamically allocate memory on the heap.
- In order to use these functions you have to include the **stdlib.h** header file in your program. Table z.4 summarized these functions.
- Keep in mind that there are other functions you will find for dynamic memory allocation, but they are implementation dependant.

Function	Prototype and Description
malloc()	void * malloc (size_t nbytes);

nbytes is the number of bytes that to be assigned to the pointer. The function returns a pointer of type void*. When allocating memory, malloc() returns a pointer which is just a byte address. Thus, it does not point to an object of a specific type. A pointer type that does not point to a specific data type is said to point to void type, that is why we have to type cast the value to the type of the destination pointer, for example:
 	
char * test;
test = (char *) malloc(10);
 	
This assigns test a pointer to a usable block of 10 bytes.	

calloc()	void * calloc (size_t nelements, size_t size);

calloc() is very similar to malloc() in its operation except its prototype have two parameters. These two parameters are multiplied to obtain the total size of the memory block to be assigned. Usually the first parameter (nelements) is the number of elements and the second one (size) serves to specify the size of each element. For example, we could define test with calloc():
 	
int * test;
test = (int *) calloc(5, sizeof(int));
 	
Another difference between malloc() and calloc() is that calloc() initializes all its elements to 0.	

realloc()	void * realloc (void * pointer, size_t elemsize);
It changes the size of a memory block already assigned to a pointer. pointer parameter receives a pointer to the already assigned memory block or a null pointer (if fail), and size specifies the new size that the memory block shall have. The function assigns size bytes of memory to the pointer. The function may need to change the location of the memory block so that the new size can fit; in that case the present content of the block is copied to the new one. The new pointer is returned by the function and if it has not been possible to assign the memory block with the new size it returns a null pointer.	
free()	void free (void * pointer);
It releases a block of dynamic memory previously assigned using malloc(), calloc() or realloc(). This function must only be used to release memory assigned with functions malloc(), calloc() and realloc().	
NULL	NULL is a defined constant used to express null pointers, that is, an unassigned pointer (pointing to the address 0) or a pointer that points to something but not useful.
size_t	Defined type used as arguments for some functions that require sizes or counts specifications. This represents an unsigned value generally defined in header files as unsigned int or by using typedef, typedef unsigned int size_t;

Table z.4

- In practice, one must always verify whether the pointer returned is NULL. If malloc() is successful, objects in dynamically allocated memory can be accessed indirectly by dereferencing the pointer, appropriately cast to the type of required pointer.
- The size of the memory to be allocated must be specified, in bytes, as an argument to malloc(). Since the memory required for different objects is implementation dependent, the best way to specify the size is to use the sizeof operator. Recall that the sizeof operator returns the size, in bytes, of the operand.
- For example, if the program requires memory allocation for an integer, then the size argument to malloc() would be sizeof(int).
- However, in order for the pointer to access an integer object, the pointer returned by malloc() must be cast to an int *.
- The typical code example may be in the following form:

```
int *theptr;
theptr = (int *)malloc(sizeof(int));
```

- Now, if the pointer returned by malloc() is not NULL, we can make use of it to access the memory indirectly. For example:

```
if (theptr != NULL)
*theptr = 23;
```

- Or, simply:

```
if (theptr)
*theptr = 23;
printf("Value stored is %d\n", *theptr);
```

- Later, when the memory allocated above may no longer be needed we have to free up the memory using:

```
free((void *) theptr);
```

- This will de-allocate the previously allocated block of memory pointed to by theptr or simply, we could write:

```
free(theptr);
```

- theptr is first converted to void * in accordance with the function prototype, and then the block of memory pointed to by theptr is freed.
- It is possible to allocate a block of memory for several elements of the same type by giving the appropriate value as an argument. Suppose, we wish to allocate memory for 200 float numbers. If fptr is a:

```
float *
```

- Then the following statement does the job:

```
fptr = (float *) malloc(200 * sizeof(float));
```

- fptr pointer points to the beginning of the memory block allocated, that is the first object of the block of 200 float objects, fptr + 1 points to the next float object, and so on.
- In other words, we have a pointer to an array of float type. The above approach can be used with data of any type including structures.
- In C++ the equivalent construct used are new for memory allocation and delete for de-allocation.

----------------------------------o0o ----------------------------------

Further reading and digging:

1. Check the best selling C and C++ books at Amazon.com.
2. C and buffer overflow tutorial: stack frame activity.
3. C and Assembler, Compiler and Linker.

|< C & C++ Compiler, Assembler, Linker & Loader | Main | C Storage Class & Memory 2 >| C++/OOP | Site Index | Download |

C Storage Class & Memory: Part 1 | Part 2 |