Chapter 9

Storage Classes of Variables

Dr. Niroj Kumar Pani

nirojpani@gmail.com

Department of Computer Science Engineering & Applications

Indira Gandhi Institute of Technology

Sarang, Odisha

Chapter Outline…

- Introduction
- The Storage Class ‘auto’
- The Storage Class ‘register’
- The Storage Class ‘static’
- The Storage Class ‘extern’
- Memory Layout & Allocation for Program Components

Introduction

- **In the Chapter "Tokens & Data types" we have learnt that every variable is associated with a data type** (which indicates the type of data stored in the variable and the size of the variable).
- **However, the above statement is not entirely correct. In addition to a 'data type', every variable is also associated with a 'storage class' which tells four things about the variable.**
  - **Storage location: Where the variable is stored, whether within the primary memory (RAM) or within the CPU registers?**
  - **Default initial value: What is the default initial value of the variable?**
  - **Scope / Visibility: The region of code within which the variable is visible.**
  - **Extent / Lifetime: The time for which a memory is associated with the variable (i.e., the variable is alive).**

- C defines four type of storage classes.
    1. Auto (stands for automatic)
    2. Register
    3. Static
    4. Extern (stands for external)
- [NOTE]: The storage class of a variable *must be* specified when the variable is declared.

Let us examine each storage class one by one in detail.

The Storage Class `auto`

- **The storage class of a variable is said to be** `auto` **(automatic) if, it is declared within a function or a block** (a block is a set of statements enclosed within a pair of curly braces) **with the keyword** `auto` **preceding its data type.**

```
void main()
{
   auto int i = 3;
   ...
}
```

- **The default storage class of a variable is** `'auto'`**. i.e.,**

```
void main()
{
   auto int i = 3;
   ...
}
```

is same as:

```
void main()
{
   int i = 3;
   ...
}
```

This is the reason why we were able to declare variables so far without specifying their storage classes.

- **Properties:**

  1. **Storage location:** **Primary memory i.e., RAM (within the stack segment)**

     [To know about ‘stack segment’ refer the section “Memory Layout & Allocation for Program Components”].

  2. **Default initial value:** **A garbage value.**

  3. **Scope / Visibility:** **Within the function / block in which the variable is defined.**

  4. **Extent / Lifetime:** **Till the function / block in which the variable is defined, ends.**

     This is the reason why these variables are called *automatic* or *local* variables. Their lifetime is *local* to the function / block in which they are defined and they are *automatically* destroyed when the current function / block ends.

Few programming examples given next, will help in clarifying the concept related to scope and extent.

Programming Example 1:

```
/* PR9_1.c: Illustration of 'auto' variables */

# include <stdio.h>

void main()
{
    auto int i=1;
    {
        auto int i=2;
        {
            int i=3;
            printf("\ni=%d", i); /* Output: i = 3 */
        }
        printf("\ni=%d", i); /* Output: i = 2 */
    }
    printf("\ni=%d", i); /* Output: i = 1 */
}
```

Output

```
i=3
i=2
i=1
```

How does the code in "Programming Example 1" work?:

The compiler treats the three i's as totally different variables, since they are defined in different blocks. Each 'i' is visible and alive within its own block.

- **When the control is in the `printf()` statement of the inner most block, three i's are alive and visible to this block** (i=1, i=2, i=3). **In such a case, the precedence is always given to the local variable. So,** i=3 **is printed.**
- **When the control is in the `printf()` statement of the middle block, two i's are alive and visible to this block** (i=1, i=2). **Since, the precedence is given to the local variable,** i=2 **is printed.**
- **When the control is in the `printf()` statement of the outer most block, only one i is alive and visible to this block** (i=1). **So,** i=1 **is printed.**

Conclusion: The following conclusion can be drawn from the above discussion. Whenever an automatic variable is searched, the current block is searched 1st. If the variable is found, its value is accessed. Otherwise, its parent block is searched, and so on..

Programming Example 2:

```
/* PR9_2.c: Illustration of 'auto' variables */

# include <stdio.h>

void main()
{
    auto int i=1;
    {
        auto int i=2, j=3;
        {
            int j=4;
            printf("\ni=%d", i); /* Output: i = 2 */
            printf("\nj=%d", j); /* Output: j = 4 */
        }
        printf("\ni=%d", i); /* Output: i = 2 */
    }
    printf("\ni=%d", i); /* Output: i = 1 */
}
```

Output

```
i=2
j=4
i=2
i=1
```

■ Programming Example 3:

```
/* PR9_3.c: Illustration of 'auto' variables */

# include <stdio.h>

void main()
{
     {
          auto int i=2;
          {
               printf("\ni=%d", i); /* Output: i = 2 */
          }
          printf("\ni=%d", i); /* Output: i = 2 */
     }
     printf("\ni=%d", i); /* Error: "i undeclared" */
}
```

Output

```
Compilation error.
The error message is: "i undeclared"
```

The Storage Class `'register'`

- The storage class of a variable is said to be `register` if, it is declared within a function or a block with the keyword `register` preceding its data type.

```
void main()
{
   register int i = 3;
   ...
}
```

- Properties: All the properties of register variables are same as that of the automatic variables except just one.
  - The storage location: While the automatic variables are stored within the primary memory (RAM), the register variables are stored within the CPU registers.

- **NOTES:**
  - **Register variables have one advantage over the automatic variables:** Since the register variables are stored within the CPU registers, which are closer to the processor in comparison to the RAM, accessing time (hence execution time) of register variables is less in comparison to the automatic variables.
  - **Then why not declare all the variables as register?:** The problem is that the number of CPU registers are limited and therefore, they can't accommodate quite a number of variables. If we do so, the compiler automatically converts the excess variables to 'auto'.
  - **The bottom line:** It is advisable to declare the most frequently variables , such as the loop counters, as register

The Storage Class `static`

- **The storage class of a variable is said to be** `static` **if, it is declared with the keyword** `static` **preceding its data type.**

```
static int i = 3;
```

- **Types: Depending upon the place of declaration there can be two types of static variables**
  - **Block static variables**
  - **File static variables**

Block Static Variables

- **It is a static variable (a variable declared with the keyword `static` preceding its data type) which is declared within a function or within a block.**

```
void main()
{
     static int i;
}
```

- **Properties:**
  1. **Storage location: Primary memory i.e., RAM (within the data or BSS segment)** [To know about 'Data segment' and 'BSS segment' refer the section "Memory Layout & Allocation for Program Components"].
  2. **Default initial value: Zero (0).**
  3. **Scope / Visibility: Within the function / block in which the variable is defined.**
  4. **Extent / Lifetime: Till the program ends.** (Because of this property, a block static variable is initialized just once for the 1st time when the program starts.)

- **Example: The following programs demonstrates the difference between the block static and automatic variables.**

```
/* PR9_4_1.c: Program illustrating 'block
static' variable */

# include <stdio.h>

void Increment()
{
     static int i=1;
     printf("\ni=%d", i);
     i++;
}
void main()
{
     Increment();
     Increment();
     Increment();
}
```

Output

```
i=1
i=2
i=3
```

```
/* PR9_4_2.c : The same program using
'automatic' variable */

# include <stdio.h>

void Increment()
{
     int i=1;
     printf("\ni=%d", i);
     i++;
}
void main()
{
     Increment();
     Increment();
     Increment();
}
```

Output

```
i=1
i=1
i=1
```

File Static Variables

- It is a static variable (a variable declared with the keyword `static` preceding its data type) which is declared outside of all functions.

```
static int i;
void Increment()
{
    ...
}
void main()
{
    ...
}
```

- Properties: All the properties of file static variables are same as that of the block static variables except just one.
  - Scope / visibility: While scope of the block static variables is limited to the function / block in which they are defined, the file static variables are visible within the total file (i.e., to all functions / blocks within the file).

The Storage Class `extern`

- The storage class `extern` is used to deal with external or global variables.
- **Let us understand the concept:**
  - A global variable is one, which is **declared outside of all functions.**

```
int i=5;  /* Global variable.
             Notice the difference between a global variable and a file static
             variable. A file static variable would have been declared like
             static int i=5; */

void Increment()
{
   ...
}

void main()
{
   ...
}
```

- **Now, if a global variable is declared after a function in which it is used, or it is declared in another file** (in multi file program a global variable can be declared in one file and used in another file)**, then to tell the current function (or, block) that a global variable exists, we declare the *same* global variable within the block by using the keyword ‘extern’ preceding its data type.**

```
void Increment()
{
   printf(“\ni++=%d”, i++);  /* Error: “i undeclared” */
}

void main()
{
   extern i;
   printf(“\ni=%d”, i);  /* Output: i=5 */
   Increment();
}

int i=5;  /*  Global variable. *.
```

- **Properties of Global / Extern Variables:**

  1. **Storage location: Primary memory i.e., RAM (within the data or BSS segment)** [To know about 'Data segment' and 'BSS segment' refer the section "Memory Layout & Allocation for Program Components"].

  2. **Default initial value: Zero (0).**

  3. **Scope / Visibility: Global** (i.e., across files). Recall that the scope of the file static variables are limited within the file.

  4. **Extent / Lifetime: Till the program ends.**

■ Programming Example:

```
/* PR9_5.c: Illustration of 'global' (extern) variables */
# include <stdio.h>
int x=10;
void main()
{
     extern y;
     int x=20;
     printf("\nWithin main x=%d", x);
     printf("\nWithin main y=%d", y);
     Display();
     getch();
}
void Display()
{
     extern y;
     printf("\n\nWithin Display x=%d", x);
     printf("\nWithin Display y=%d", y);
}
int y = 30;
```

Output

```
Within main x=20
Within main y=30

Within Display x=10
Within Display y=30
```

■ [NOTE]: C doesn't have a scope resolution operator (::), as we have in C++.

Summery:

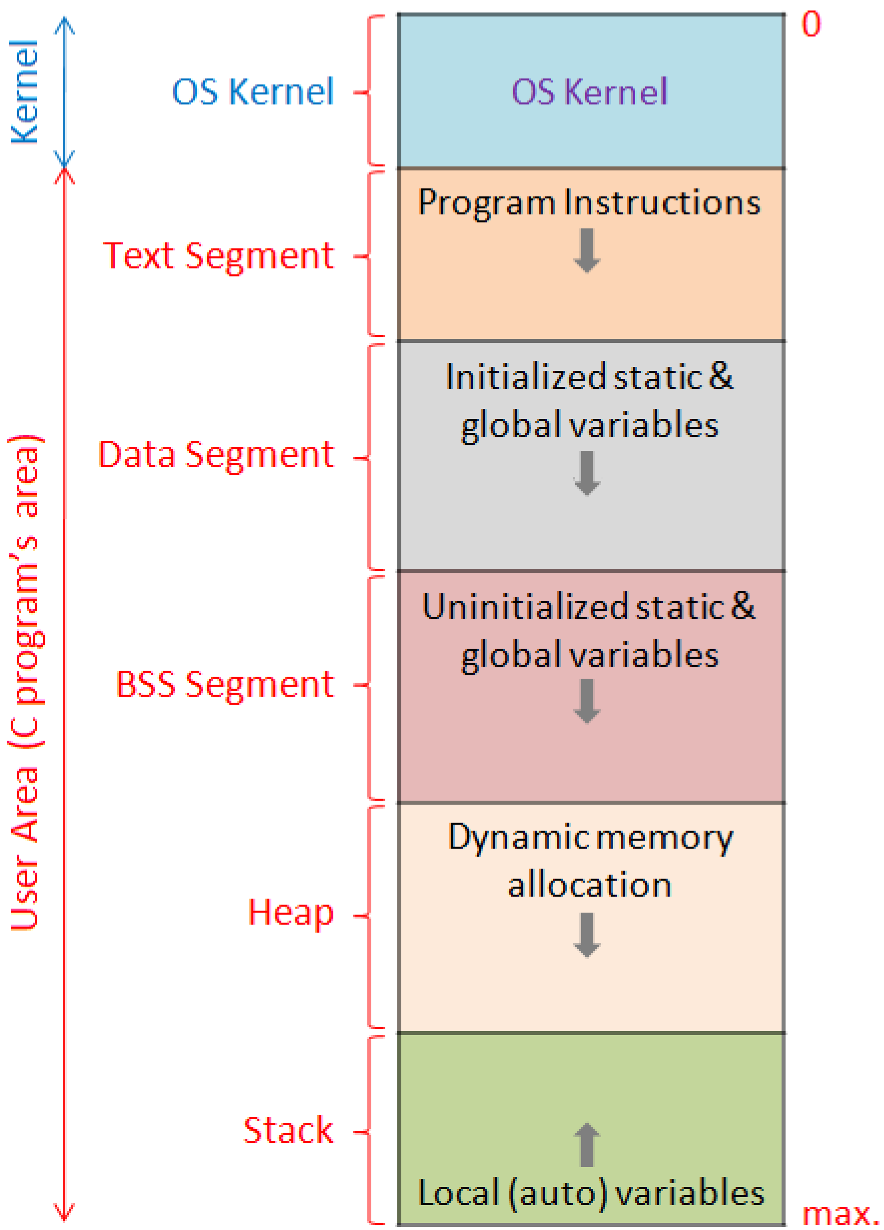
Storage Classes	Storage Location	Default Initial Value	Scope	Extent
Auto	RAM (stack segment)	Garbage	Within the function / block in which the variable is defined.	Till the function / block in which the variable is defined, ends.
Register	CPU Register	Garbage	Within the function / block in which the variable is defined.	Till the function / block in which the variable is defined, ends.
Block Static	RAM (Data / BSS segment)	0	Within the function / block in which the variable is defined.	Till the program ends.
File Static	RAM (Data / BSS segment)	0	Within the total file (i.e., to all functions / blocks within the file).	Till the program ends.
Extern (Global)	RAM (Data / BSS segment)	0	Global (i.e., across files).	Till the program ends.

Memory Layout & Allocation for Program Components

- **In this section, we will explore the layout of the computer's memory (RAM) and will also see how the various program components** (by program components, we mean program instructions and the variables) **are allocated within the memory (RAM).**
- **The computer's memory** (so far, a 'C' program's storage area is concerned) **is typically divided into five areas / segments:**
  1. **Text / Code segment: Holds the compiled code of the program.**
  2. **Data segment: Holds the static** (both block static and file static) **and global variables that are initialized to non-zero values.**
  3. **BSS (Block Started by Symbol) segment: Holds the static** (both block static and file static) **and global variables that are initialized to zero values by default.**
  4. **Heap segment: Free space used for dynamic memory allocation** (dynamic memory management is discussed in detail in another Chapter).
  5. **Stack segment: Holds the local (automatic) variables.**

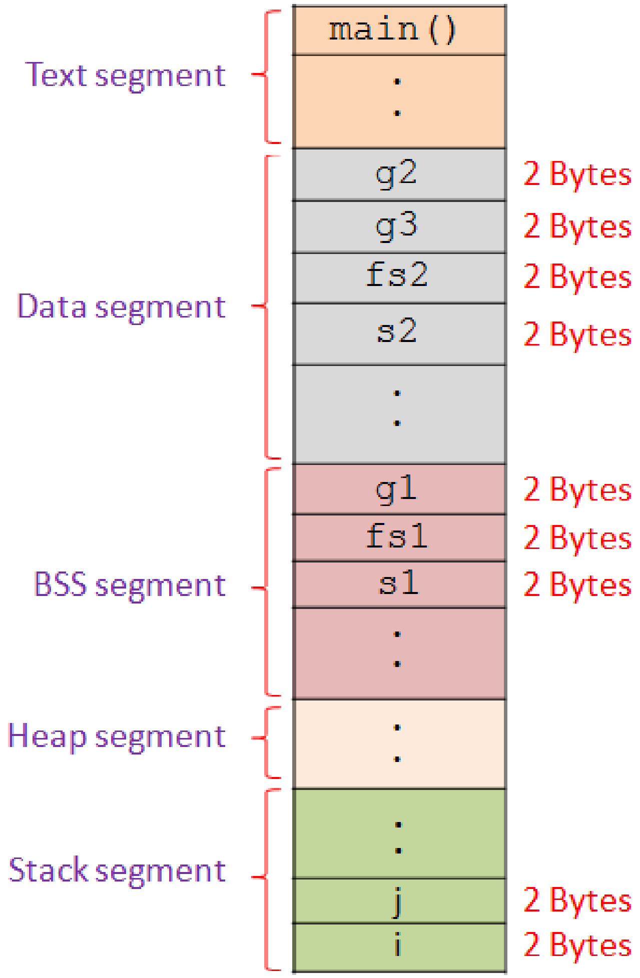
The adjacent diagram shows the memory layout of a computer.

The arrows in each segment indicates the order in which the program components (instructions and variables) are stored in accordance with their declarations.



- **The following sample codes will help in clarifying the concept:**

Sl. #	Sample Code	Memory Layout

1.	`/* Global variables */`
	`int g1, g2 = 1, g3 = 2;`
	
	`/* File static variables */`
	`static int fs1, fs2 = 3;`
	
	`void main()`
	`{`
	`    /* Block static variables */`
	`    static int s1, s2 = 5;`
	
	`    /* Local (automatic) variables */`
	`    int i = 7, j = 8;`
	`    ...`
	`}`
		

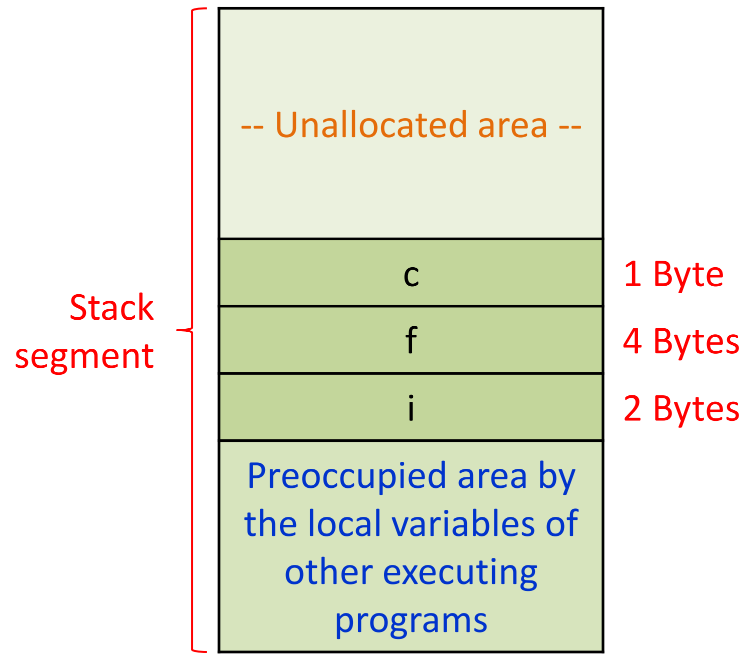
[NOTE]: In the sample code given in the last slide, we have explicitly assumed that this is the only program (say, our program name is PR_X) executing in the computer.

However, if one more program, say PR_Y, is executing simultaneously with PR_X, which has started prior to PR_X, then the program components (instructions and variables) of PR_Y are placed first in the appropriate sections in accordance with their declaration, and then the program components of PR_X are placed.

Let us see some more examples involving the allocation of local (automatic) variables only, because it is little bit tricky.

2.

```
void main()
{
    /* Local (automatic) variables */
    int i;
    float f;
    char c;
}
```

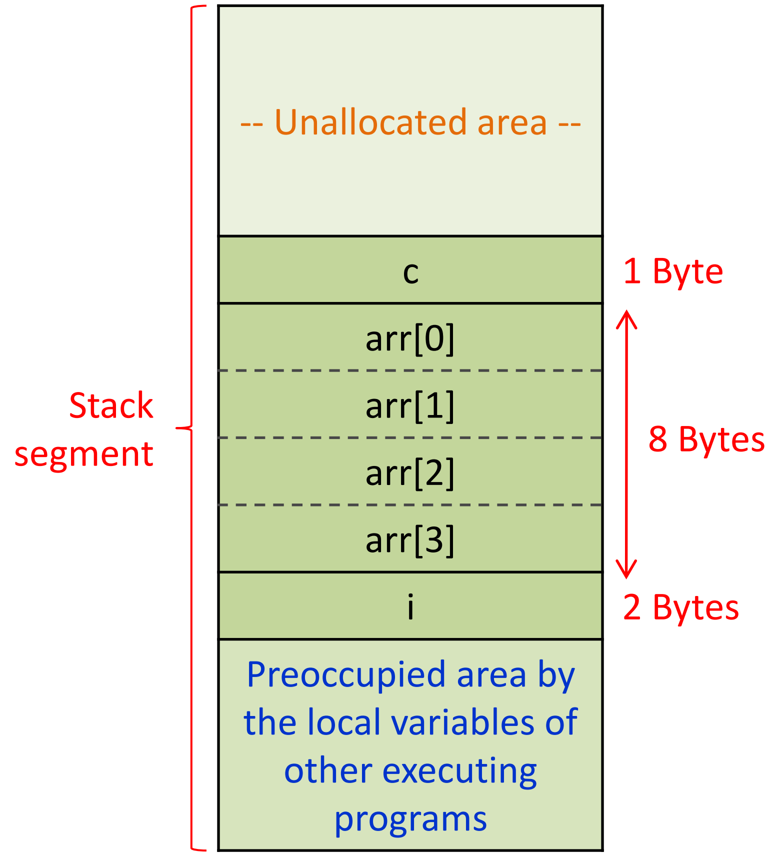


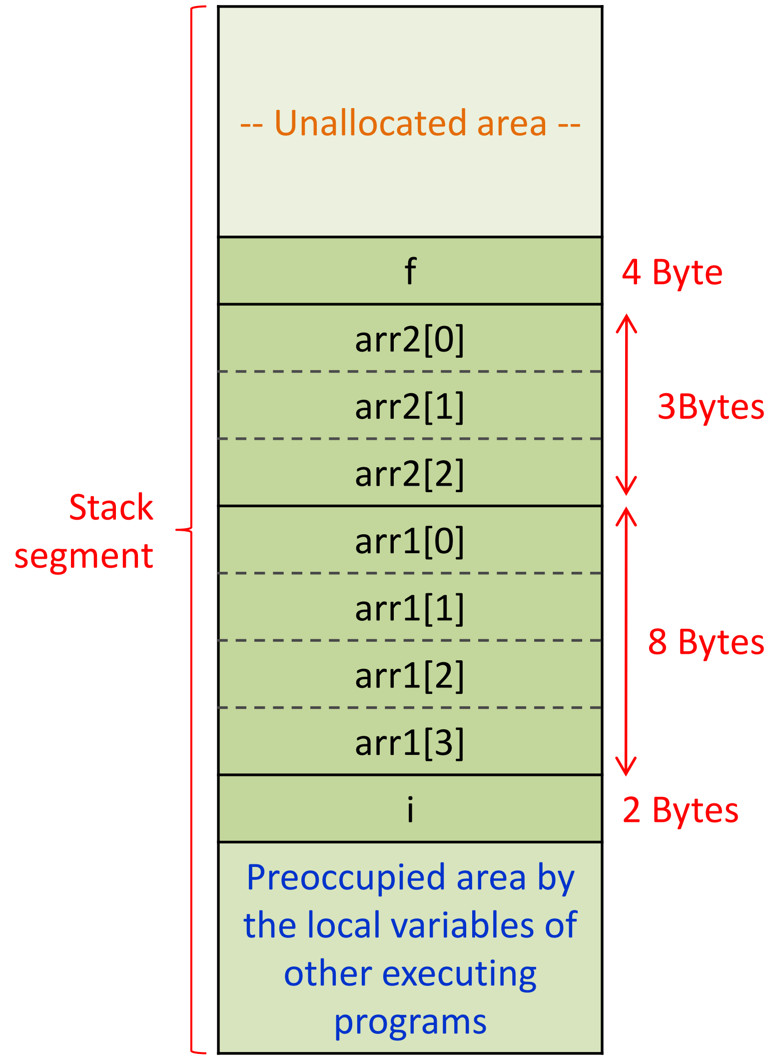
3.

```
void main()
{
    /* Local (automatic) variables */
    int i;
    int arr[4];
    char c;
}
```

[NOTE]: In the 'memory layout' notice how the individual elements of the array 'arr[4]' are allocated.

It is because, ‘arr[4]’ refers to <u>only one variable</u> where the individual elements are stored in contagious memory locations (i.e., increasing order of memory locations).



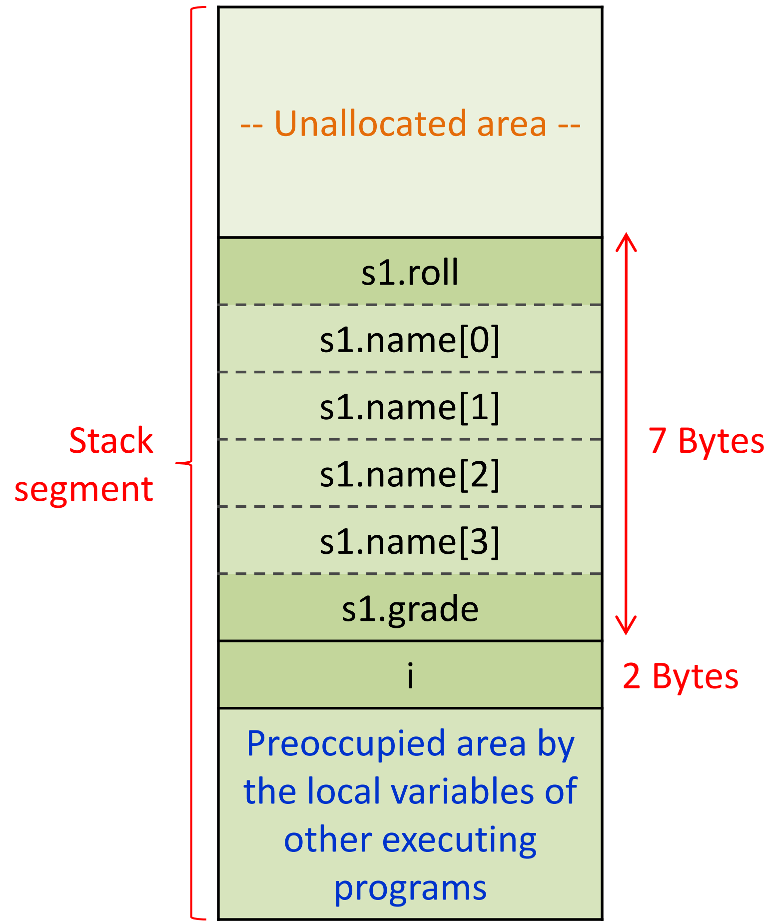
4.	`void main()` `{` `/* Local (automatic) variables */` `int i;` `int arr1[4];` `char arr2[3];` `float f;` `}`	

5.

```
struct Student
{
    int roll;
    char name[4];
    char grade;
};
void main()
{
    /* Local (automatic) variables */
    int i;
    struct Student s1;
}
```

[NOTE]: Notice how the individual elements of the structure ‘s1’ are allocated (Structures are discussed in the next chapter).

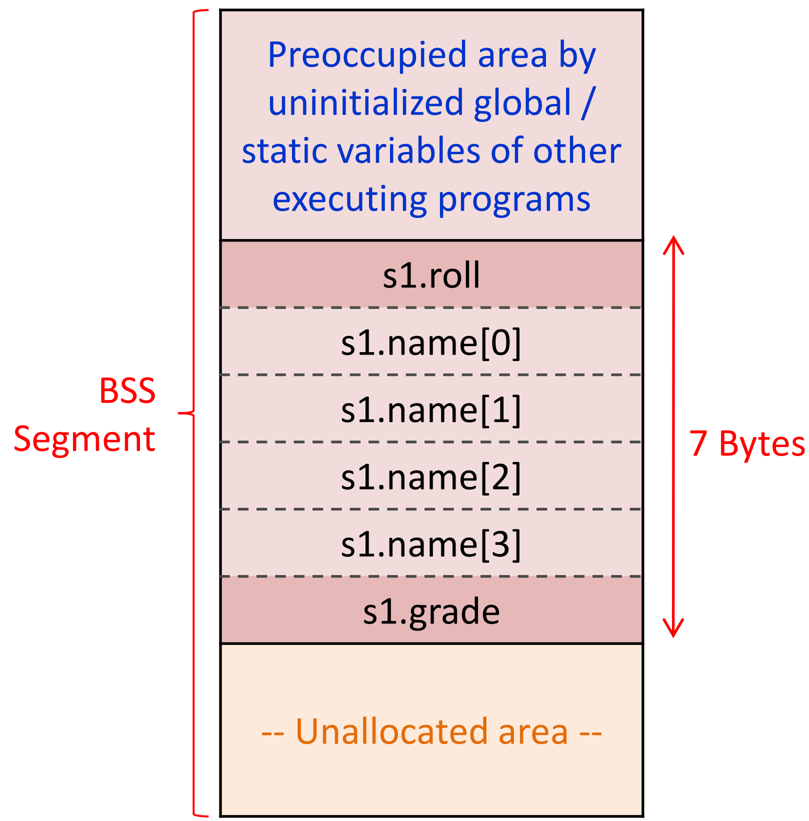
It is because, ‘s1 refers to only one variable where the individual elements are stored in contagious (increasing) memory locations



[NOTE]: In the last example (Sl. #5), the structure is *declared* within main(), hence it becomes an ‘auto’ variable and therefore, stored in the stack section. However, if it is declared as a global variable (declared outside of all functions), the it should be stored within the data or BSS segment depending upon whether it is *initialized* or not. This is shown below:

6.

```
struct Student
{
    int roll;
    char name[4];
    char grade;
}s1;
```



End of Chapter 9