CS1100 - Lecture 27

Instructor : Jasine Babu
Teaching Assistants : Nikhila K N, Veena Prabhakaran

User defined data types

So far, we were using standard data types provided by the language, like *int*, *float*, *char* etc . and also the arrays of these different types. There are provisions for defining user defined data types in C. In this lecture, we will discuss about user defined data types.

Let us begin with an example, suppose we want to manipulate the list of records related to student data such as roll number, name, marks, total mark etc. . Using methods we have studied so far, we will be defining different arrays for handling each of these different fields of data and then access the data of each student using the array index value.
Eg:

```
int rollNo[150];
int mark1[150];
int mark2[150];
int total[150];
char name[150];
```

We were assuming that the i^{th} element in the `rollNo` array, the i^{th} element in the `mark1` array, the i^{th} element in the `mark2` array, the i^{th} element in the `total` array and the i^{th} element in the `name` array are all of the same person. But this assumption is only there in the mind of the programmer. There is no apparent relation between these arrays. This problem can be solved by using the concept of user defined data types called *structures*. The basic syntax for defining a *structure* named `student` is given below.

```
struct student
{
   int rollNo;
   int mark1;
   int mark2;
   int total;
   char name[50];
};
```

In the above syntax, `struct` is the keyword and `student` is the name of the structure. The names `rollNo`, `mark1`, `mark2`, `total` and `name` are known as member fields of the structure `student`. Usually the declaration of a structure as given above is written before the `main()` function.

Note that, `student` is not a variable here; it is only a name given to the structure to represent the five fields `rollNo, mark1, mark2, total` and `name` together as one unit. The syntax to declare a variable `s` of type `struct student` is as follows.

```
struct student s;
```

This means that, `s` is a variable of type `struct student`.

How to store and access structure data

The dot (.) operator helps to store and access the structure fields. Suppose `s` is a variable of type `student`. To refer to the member field `rollNo` of student `s`, we use the notation `s.rollNo`. Similarly, `s.mark1, s.mark2, s.total,` and `s.name` will represent respectively the member fields `mark1, mark2, total` and `name` of student `s`.

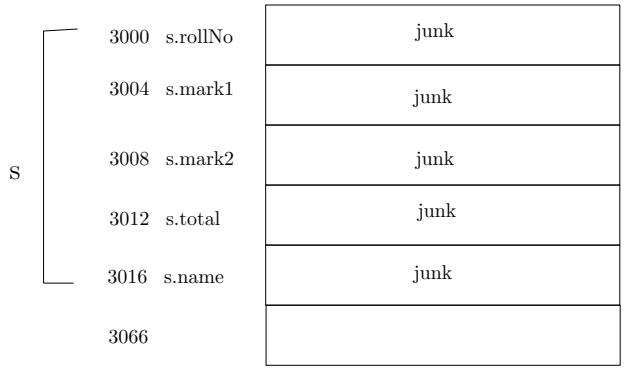
The following instruction helps to read a string from the terminal and store it to the `name` field of the student `s`.

```
scanf("%19s",s.name);
```

To change the roll number of student `s` to 101, we can use the following instruction.

```
s.rollNo=101;
```

If we declare a variable `s` of type `struct student` in a function, memory is allocated for storing the variable `s` as shown in the figure below.



Notice that, each member field of `s` has got an address of its own and they behave like variables whose values can be accessed or modified seperately as mentioned earlier. However, note that, `s` itself is a variable which has got an address and it can occur on the left hand side of an assignment statement. As per the above figure, `s` is a variable with address `3000` and it fills in locations up to `3065`. This is completely different from arrays because, array names are not variables, they just represent a group of variables.

Suppose, we have two variables `s1` and `s2` of type `struct student` and we make an assignment statement `s1=s2;`. As per our discussion above, each of these variables occupy 66 bytes of memory. When this instruction is executed, the contents of the 66 bytes of `s2` are copied to occupy the 66 bytes used to store `s1`.

Functions using structure variables

It is possible to define functions that return a structure variable. It is also possible to define functions that take structure variables as parameters. Similarly, pointers to a structure variable can also be returned from a function or taken as parameter to a function.

Let us see how to write a function `inputStudentDetails()` that takes the details of a student as input from the terminal, store these details to a variable of type `struct student` and returns this variable. This function can be declared as follows.

```
struct student inputStudentDetails();
```

The definition of this function can be written as follows.

```
struct student inputStudentDetails()
/*Input details of one student and return it */
{
      struct student s;
      printf("\nGive roll number, name, mark1, mark2 of student\n");
      scanf("%d", &s.rollNo);
      scanf("%14s", s.name);
      scanf("%d", &s.mark1);
      scanf("%d", &s.mark2);
      return s;
}
```

This function is analogous to a function that reads an integer as input from the terminal and returns it as follows.

```
int getInteger()
{  int k;
   scanf("%d", &k);
   return k;
}
```

A function `printStudentDetails()` that takes a variable `s` of type `struct student` as parameter and prints all the details of student `s` can be declared as follows.

```
void printStudentDetails(struct student);
```

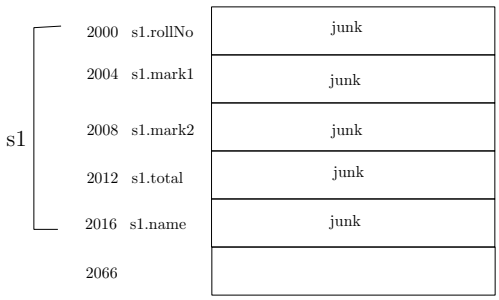
This function can be defined as follows.

```
void printStudentDetails(struct student s)
/*Print details of student s */
{
      printf("%d\t", s.rollNo);
      printf("\t%s\t", s.name);
      printf("%d\t", s.mark1);
      printf("%d\t", s.mark2);
      printf("%d\n", s.total);
}
```

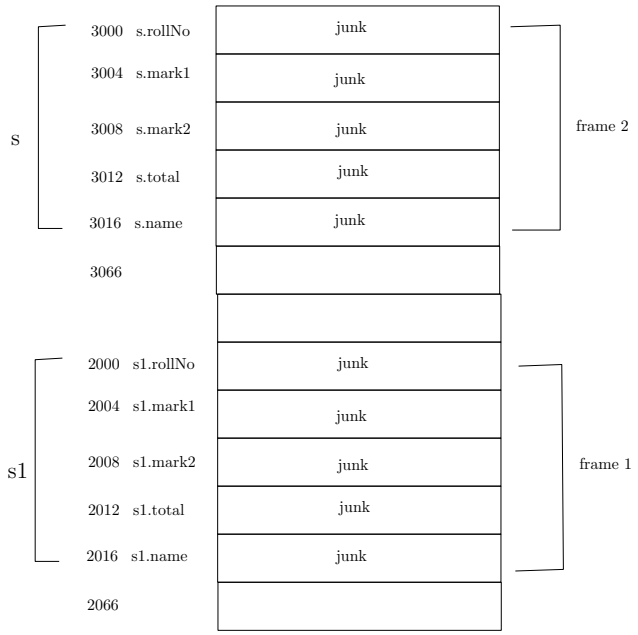
The following `main()` function uses the function `inputStudentDetails()` to read values into structure variable `s1` of type `struct student` and the function `printStudentDetails()` is used to display the details of `s1`.

```
int main()
{
   struct student s1;
   s1=inputStudentDetails();
   s1.total=s1.mark1+s1.mark2;
   printStudentDetails(s1);
   return (0);
}
```

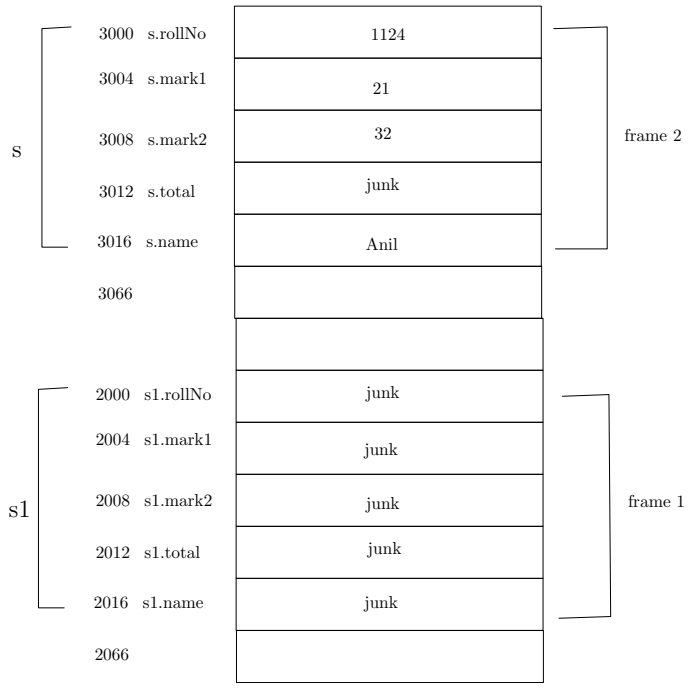
Let us consider the execution of the above `main()` function. When `main()` function starts executing, a stack frame gets created in memory which allocates space for the variable `s1`. This variable will be used to store the data of a single student. The memory stack diagram before the function `inputStudentDetails()` is invoked is given below.



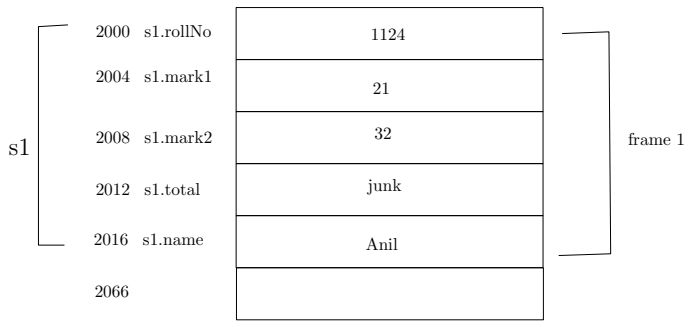
When the function `inputStudentDetails()` is invoked, a new stack frame gets created in the memory. Now, the memory state diagram is as follows.



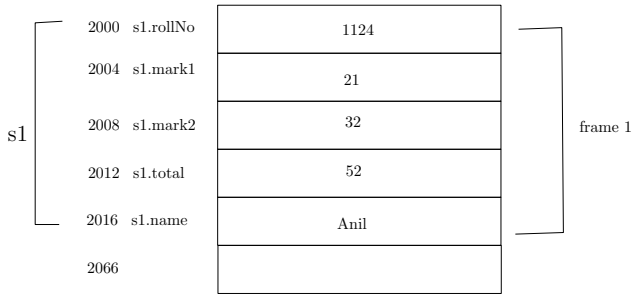
The following figure shows the contents of the memory locations after executing all the `scanf` instructions in the function `inputStudentDetails()` .



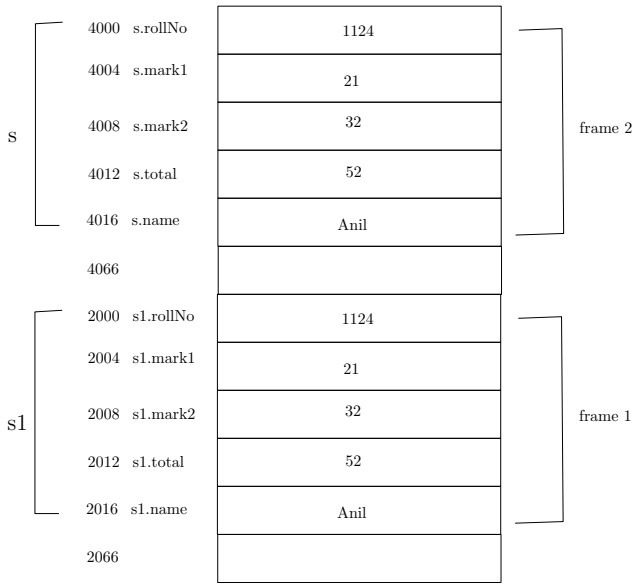
When the execution of the function `inputStudentDetails()` finishes by executing the instruction `return s;`, the program control goes back to `main()` and at this time, since `s1` is on the left hand side of the function call to `inputStudentDetails()`, the contents of the location of `s` are copied to the location of `s1` and frame 2 is deleted. That is, the `66` bytes of data starting from location `3000` is copied to the `66` bytes of memory starting from location `2000`. At this point, the memory state diagram is as follows.



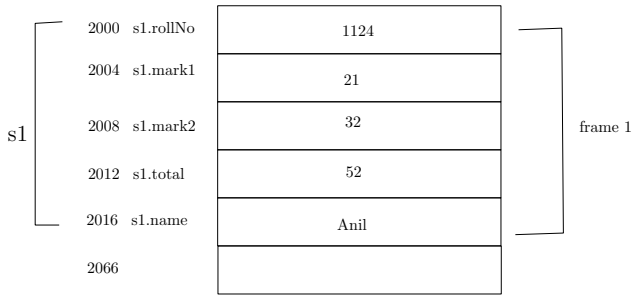
After executing the statement `s1.total=s1.mark1+s1.mark2;` the value of `s1.total` gets updated. At this point, the contents of the memory will be as follows.



When the function call `printStudentDetails(s1);` is invoked, a new frame is created and the contents of the location of `s1` will be copied to the location of `s` in the new frame.



When `printStudentDetails` is executed, the details of student `s` are printed. After this, the function finishes its execution, frame 2 gets deleted and the program control goes back to `main()`. At this point, the contents of the memory will be as follows.



After this, the `main()` function finishes execution.

Working with arrays of structures

The following instruction creates an array `l` of 10 elements of type `struct student`. This array can store the details of 10 students.

```
struct student l[10];
```

Consider the following updated structure definition and `main()` function declaration.

```
struct student
{  int roll;
   unsigned int marks[2];
   unsigned int total;
   char name[15];
};

int main()
{
   struct student l[SIZE];
   int i, n;
   printf("enter number of students (<%d) \n", SIZE);
   scanf("%d",&n);
   if(n > SIZE)
   {
      printf("error \n");
      return 0;
   }
   for(i=0; i<n; i++)
   {
      printf("\nGive roll number, name, mark1, mark2 of student %d\n", i+1);
      l[i]=inputStudentDetails();
      printf("\n");
      updateTotal(&l[i]);
   }
   swap(&l[0], &l[n-1]);
   printf("\nupdated list is \n");
   for(i=0; i<n; i++)
   {
     printStudentDetails(l[i]);
   }
}
```

The above `main()` function invokes four functions; `inputStudentDetails()`,`swap()`, `printStudentDetails()` and `updateTotal()` . The function call `inputStudentDetails()` in the instruction `l[i]=inputStudentDetails()` is intended to read the details of the i^{th} student to the location of `l[i]`. and the function call `printStudentDetails(l[i])` is used to display the details of `l[i]`. The function call `updateTotal(&l[i]);` is used to update total mark of the student whose details are stored in `l[i]` and the function call

`swap(&l[0], &l[n-1]);` is intended to exchange the details of students stored as `l[0]` and `l[n-1]`.

Since the definition of the structure is different from our earlier definition, we can rewrite the `inputStudentDetails()` function as follows.

```
struct student inputStudentDetails()
/*Input details of one student and return it */
{
      struct student s;
      scanf("%d", &s.roll);
      scanf("%14s", s.name);
      scanf("%d", &s.marks[0]);
      scanf("%d", &s.marks[1]);
      return s;
}
```

The function `swap()` can be declared as follows.

```
void swap(struct student *, struct student *);
```

The definition of the function can be written as follows.

```
void swap(struct student *ps1, struct student *ps2)
{
   struct student temp;
   temp=*ps1;
   *ps1=*ps2;
   *ps2=temp;
}
```

Notice the similarity of the above function definition with the `swap()` function we studied earlier for exchanging the values of two integer variables.

When the statement `temp=*ps1` is executed, the values of all member fields of the structure variable whose address is stored in `ps1` are copied to the locations of the corresponding fields of `temp`. Note that, if some member fields is an array like `marks` in the above example, all the elements of the array get copied. Thus, the single statement `temp=*ps1` is equivalent to executing the following set of instructions.

```
temp.roll=(*ps1).roll;
temp.marks[0]=(*ps1).marks[0];
temp.marks[1]=(*ps1).marks[1];
temp.total=(*ps1).total;
strcpy(temp.name,(*ps1).name);
```

Now, let us consider the declaration of the function `updateTotal()`. Since, we need to change the `total` field of a student using this function, it is necessary to pass the address of the student variable whose total has to be updated as the parameter. Therefore, as

in the case of the `swap()` function defined before, the parameter of this function should also be of type `struct student *`. The declaration of the function `updateTotal()` is as follows.

```
void updateTotal(struct student *);
```

The definition of this function can be made as follows

```
void updateTotal(struct student *ps)
/*Update the total marks of one student whose address is the parameter */
{
     (*ps).total=(*ps).marks[0]+(*ps).marks[1];
}
```

Note that, the brackets around `*ps` is necessary, because the dot (.) operator has got higher precedence than the star (*) operator.