CSD101: Introduction to computing and programming (ICP)

Pointer declaration and use I

- `int *ip;` declares that `ip` is the address-of/pointer-to an integer. Consider the declaration `int i = 5, j;` then `ip = &i;` assigns the memory location for the variable `i` to `ip`. So, `j = *ip;` will take the contents of the memory address in `ip`, namely the value `5` and assign it to `j`.
- While an address is just a positive integer in the linear sequence of bytes that is memory the & operator returns the address tagged with the type of the variable whose address it is. So, `&i` returns the address of an integer variable. So, an address/pointer is a pointer to a particular type of object.
- Note the difference in meaning when a variable name is mentioned on the LHS and RHS of an assignment statement. So, statement `i=j` means store the value in variable `j` in the address of variable `i`. So, in the code fragment:

Pointer declaration and use II

```
int i=2, j=4;
int *ip=&i;
//ip is a pointer to i, i.e. contains address of i
*ip=j;//will change contents of i to 4
```

the value of `i` will change to `4`.

- If pointers are not initialized properly they can lead to run time errors like segmentation faults - caused by illegal memory references. So, pointers can lead to obscure errors.
- Never return a pointer to a local variable from a function.

When are pointers used?

- To pass arguments when value changes inside a function must be visible outside (that is in the calling function).
- When multiple values have to be returned. (Discussed later after dynamic memory).
- When using arrays. Conserves memory by using the same chunk of memory via a single pointer.
- When dynamic, linked structures are needed. (After dynamic memory)

Pointers and arrays

- Array elements can be accessed by using pointers. For example,

```
int a[5]={1,2,3,4,5};
int *ap;
ap=&a[0]//ap points to the 1st element of array a[]
printf("%d", *(ap+4));//prints the 5th element of array a[]
*(ap+2)=*(ap+2)+10;//adds 10 to 3rd element of array a[]
```

- The compiler while passing array arguments actually passes a pointer to the beginning of the array to the function. This is the reason changes to an array are reflected back in the calling function.

Structures

- Data in the real world often comes as a data bundle or set of values that pertain to a single logical entity or object and not as separate values.
- For example, a student has several attributes like: name, roll number, department, age, login ID, address, etc. associated with each student. It makes sense to think of a student as an entity or object that has values for each of these attributes.
- Modern languages allow this using the concept of a class that can be instantiated multiple times to create different objects of that class all with the same attributes. So, in C++, Java, Python etc. we can define a Student class with attributes that all students have and then instantiate such a class to create individual students.
- **C** has a more rudimentray feature called *structures* that allows one to define a type that has a set of attributes with given types.

Structure example

```
struct {
   char name[N];
   int rollNo;
   int age;
   char loginID[N];
   int quizzes[10];
} s1={"Arun Gupta", 20201001,18,"ag100",{5,8,9}};
```

Structure tag, type definition

```
struct Student {
  char name[N];
  int rollNo;
  int age;
  char loginID[N];
  int quizzes[10];
};

typedef struct {
  char name[N];
  int rollNo;
  int age;
  char loginID[N];
  int quizzes[10];
} Student;
```

Structure properties

- Structure can be arguments, returned from functions as values and can be pointed to. They can be assigned but cannot be compared.
- Structures can also be nested.