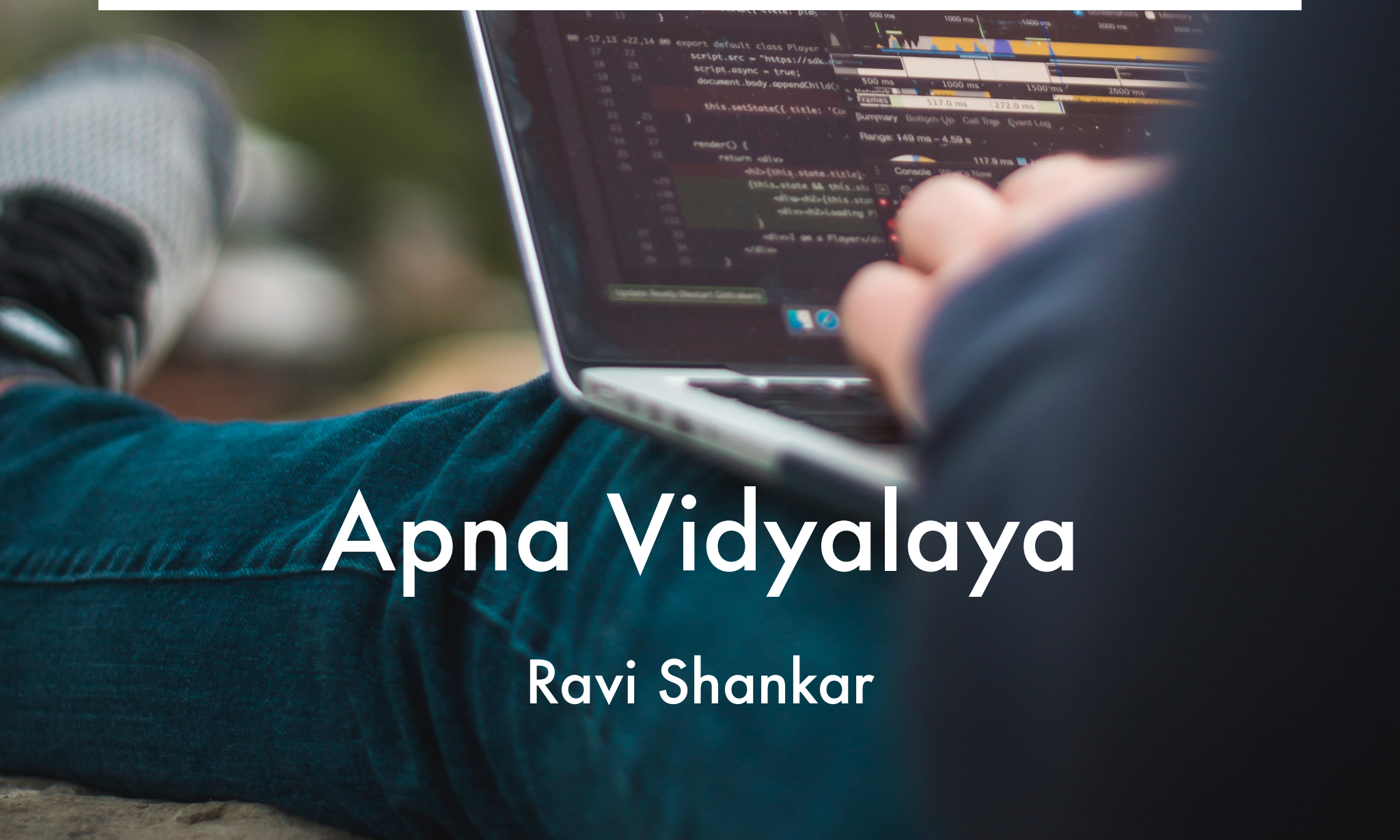
BASIC DATA STRUCTURES AND ALGORITHMS

USING C/C++

This notes is incomplete

(Still in development)



Apna Vidyalaya

Ravi Shankar

1. POINTERS

If you are at home, and I know your address, I can come at your address and meet you. Similarly, if you know a variable's address in C, you can go to its address and access its value. That's the idea behind pointers. In this chapter, we will learn to write code for pointers.

INTRODUCTION

What happens when you declare in integer?

```
int number = 10;
```

The computer choose some location in memory, let's say 4096, the memory will be given the name: number, and a value 10 will be put into that memory. It will also mark that this memory is of type number.

ACCESS VALUES IN A VARIABLE USING POINTER

How can you access the value of number? Till now, we have been using the name of the variable, i.e. number to access the values. But, we can also access the value if we knew the address where number is stored. To store the address of a variable, you need another variable. The variable which stores the address of another variables is called pointers.

Let's see an example:

```
#include<stdio.h>

int main() {
  int num = 8;
  int * pt = &num;
  printf("%d\n", *pt);
  return 0;
}
```

Here, pt is a variable of type (int *). Which means, pt is a pointer which can hold address of a variable of type int. If I were to break the above program, I would do the following

```
#include<stdio.h>

int main() {
  int num = 8;
  int * pt;
  pt = &num;
```

```
  printf("%d\n", *pt);
  return 0;
}
```

The second program adds more clarity. pt, which is a variable if type 'int *', was assigned the address of num(using &num). And the value at address pt was being outputted using *pt (value at the address pt).

Just to be clear, * in both the following lines means the different things.

```
int * pt;
```

and

```
printf("%d\n", *pt);
```

In the first line, * is associated to int, this is (int *), which is a data type. And the overall meaning of this statement is that pt is a variable of type int *.

In the second line, * is associated to pt, this is (*pt), which means that the value at the address pt.

For clarity, you can read the following constructs as:

```
  int * pt // integer pointer pt
  &num // address of num
  *pt // value at the address pt
```

PRINTING THE ADDRESS OF A VARIABLE

If you want to print the value of a pointer, i.e. the address that it contains, then you can do so. The format specifier is %p. Many books such as "Let's C by Yashwant Kanetkar", writes that you can do so using %u, but that is incorrect. Let's see an example:

```
#include<stdio.h>

int main() {
  int num = 8;
  int * pt;
  pt = &num;
  printf("The address where pt points to is: %p \n", pt);
  return 0;
}
```

Well, the more correct version would be

```
#include<stdio.h>

int main() {
```

```
  int num = 8;
  int * pt;
  pt = &num;
  printf("The address where pt points to is: %p \n", (void *)pt);
  return 0;
}
```

The difference in in the way pt is being passed to printf function. In the second case, pt is explicitly type casted to a pointer of type void, whereas in the first case that is implied.

When you print the memory address, you will almost certainly get a hex number. The standard doesn't define whether it should be hex or decimal, but almost all the implementations gives out the address in hex format.

CHANGING VALUES OF A VARIABLE USING POINTER

We will first see a program, and then we will try to understand.

```
#include<stdio.h>

int main() {
  int num = 8;
  int * pt = &num;
  *pt = *pt + 10; // value at the address pt = value at the address pt + 10
  printf("The new value in num is: %d \n", num);
  return 0;
}
```

Read the comment, and the given line will become clear to you.

TYPES OF POINTERS

All the data types that exist in C can have a pointer. Including the user defined data types also. In the following program, we will use a pointer of type char and float.

```
#include<stdio.h>

int main() {
  char myChar = 'A';
  char * myPointer = &myChar;
  *myPointer = 'B';
  //
  float myFloat = 3.5;
  float * myFloatPointer = &myFloat;
  *myFloatPointer = 4.6;
```

```
  printf("The values contained in myChar and myFloat are: %c and %f \n", myChar, myFloat);
  return 0;
}
```

POINTERS IN FUNCTION CALLS

In C, by default function calls are call by value. Which means that if you pass a variable to C program as an argument, then that function will make a local copy of that variable. If the function changes the value in its own body, the original variable will remain unaffected.

But, if you pass a pointer of a variable to a function call, and the called function changes the value at that pointer, then the original variable will gets changed. This is quite logical. We all know that pointers are also a variable, a variable that can hold address. When you pass a pointer to a function, the function will have its own local copy of the pointer variable, but the address that this pointer points to remain the same. If you go to that address and make some change, that change will be permanent. This may sound a bit confusing at first reading, but will become clear with time. Let's see an example.

```
#include<stdio.h>
void doubleTheNum(int *pt) {
  *pt = *pt * 2;
}
int main() {
  int num = 7;
  doubleTheNum(&num);
  printf("New value at num = %d \n", num);
  return 0;
}
```

Did you notice some difference in the pointer treatment. Instead of creating a new pointer variable, we just passed &num to the function that accepts a pointer to an integer.

OPERATIONS ON POINTER

What will happen if you add 1 to an integer pointer? Will in increate by one?

The answer is No. When you add something, let's say A to a pointer, the pointer first find out the size of the data type to which it points to, let's say it is 4. Then it calculates a number, say N (= 4 * A), which would be the space required to write A number of integer in memory. And then it moves N steps forward in memory. In our case, it will be A * 4.

If the integer pointer points to 4096 and size of int is 4 bytes. Then adding one to the given pointer will make it 4100. Let's see it through an example.

```c
#include<stdio.h>

int main() {
  int num = 90;
  int * p = &num;
  printf("p points to: %p \n", p);
  p = p + 1;
  printf("Now, p points to: %p\n", p);
  return 0;
}
```

The output on my system is:

```
p points to: 0x7ffeee6f89b8
Now, p points to: 0x7ffeee6f89bc
```

If I run the program again, the output will be different. But, the difference between the two hex values printed will remain the same. On my system, integer in C is of four bytes. So, increasing the pointer by one has increased the pointer value by 4.

You can subtract a point from another pointer of compatible type, you will get the value which is difference of the pointers values divided by the size of the data types it points to. Whereas, you cannot add two pointers.

2. ARRAYS

How will you store the marks of each student in a class? Will you have a variable corresponding to every student? If yes, then won't it become tough to manage? Don't worry, the answer is No. And the solution lies in Array.

INTRODUCTION

Arrays are a single variable that holds multiple data of same types. Let's see an example to understand it better.

```
#include<stdio.h>

int main() {
  int marks[] = { 2, 3, 90, 43, 67 };
  for(int i = 0; i < 5; i++) {
    printf("Value at index %d is %d\n", i, marks[i]);
  }
  return 0;
}
```

The output would be:

```
Value at index 0 is 2
Value at index 1 is 3
Value at index 2 is 90
Value at index 3 is 43
Value at index 4 is 67
```

marks is an array which contains 5 integers, and those integers can be accessed using their indices. Let's learn it in detail.

ARRAY DECLARATION

Array declaration is similar to variable declaration, except that here we have to add a pair of brackets after the array name with the size of the array. Example:

```
int marks[10];
```

If you want to define array while declaring then you can skip explicitly giving the size. In that case the syntax will become:

```
int marks[] = { 2, 3, 90, 43, 67 };
```

If you don't assign values to array while declaring, then you will have to assign values to each elements of array later on.

To access elements of an array, you can write array name with [], with the index of the element inside []. Note that, in C, arrays are 0 indexed. So, first element is at 0, second element at index 1, and so on. Let's see these concepts through an example:

```
#include<stdio.h>

int main() {
  int battingAverage[5];
  for(int i = 0; i < 5; i++) {
    battingAverage[i] = 30 + i;
  }
  for(int i = 0; i < 5; i++) {
    printf("Batting average of %dth player is: %d\n", i + 1, battingAverage[i]);
  }
  return 0;
}
```

You can have arrays on any data types, int, char, struct or any other data types.

ARRAY REPRESENTATION IN MEMORY

Arrays in C are nothing but a series of data put together in memory on continuous locations. So, if you get a pointer to the first element of the array, and you increase the pointer by 1, your pointer will now point to the second element of the array.

C doesn't care to store any other info about arrays. If you create an array of size 10 and you try to assign a value to 11th element, C will let you do so. It will overwrite the data stored at the location next to 10th element of the array with the value provided by you. If that location had something important, then that would be lost. This is considered to be a very big disadvantage with C language. For example: this program which is accessing values outside arrays will compile fine and won't give any error.

```
#include<stdio.h>

int main() {
  int battingAverage[5];
  for(int i = 0; i < 7; i++) {
    battingAverage[i] = 30 + i;
  }
  for(int i = 0; i < 10; i++) {
    printf("Batting average of %dth player is: %d\n", i + 1, battingAverage[i]);
  }
```

```
  return 0;
}
```

Since, arrays has no bounds checking mechanism, the program is responsible for managing and respecting the boundary of the array.

ARRAYS AND POINTERS

Arrays and pointers and more similar than you can think intuitively. We will learn the similarity one by one with program examples.

They both can be accessed using same syntax.

```
#include<stdio.h>

int main(){
  int d[] = {2, 93, 4};
  int *p = d;

  printf("Accessing array using pointer with array syntax\n");
  for(int i = 0; i < 3; i++){
    printf("%d\n", p[i]);
  }
  printf("\nNow accessing array using array name with pointer syntax\n");
  for(int i = 0; i < 3; i++){
    printf("%d\n", *(p +i));
  }

  return 0;
}
```

The output would be:

```
Accessing array using pointer with array syntax
2
93
4

Now accessing array using array name with pointer syntax
2
93
4
```

DIFFERENCE BETWEEN ARRAY AND POINTERS

You can perform arithmetic on pointers be adding one to it, but you cannot do the same to an array. Example:

```
#include<stdio.h>

int main() {
  int arr[] = {3, 4, 5, 6};
  int *pt = arr;
  pt += 1;// works fine
  printf("2nd value of arr using pointer is: %d\n", *pt);

  // arr += 1;// Gives compilation error
}
```

Sizeof operator returns size of entire array when operated upon array, but in case of pointers, it returns the size of the pointer. Example:

```
#include<stdio.h>

int main() {
  int arr[] = {3, 4, 5, 6, 8, 2, 34};
  int *pt = arr;
  printf("sizeof array is: %lu\n", sizeof(arr));

  printf("sizeof pointer is: %lu\n", sizeof(pt));
}
```

The output on my system is:

```
sizeof array is: 28
sizeof pointer is: 8
```

Assigning address to an array variable is not allowed.

```
#include<stdio.h>

int main() {
  int arr[] = {3, 4};
  int myInt = 8;
  // arr = &myInt; // This will cause compilation error
  return 0;
}
```

If the character array holds a string value, then the string can be modified. But, is the character pointer points to a string literal, then the value of the string cannot be modified. Example:

```
#include<stdio.h>

int main() {
  char myStr[] = "myFirstString";
  printf("%s\n", myStr);
  myStr[1] = 'R'; // This is allowed
  printf("%s\n", myStr);

  char *charP = "mySecondStr";
  printf("%s\n", charP);
  // *(charP + 1) = 'T'; // This will compile fine, but won't run properly
  printf("%s\n", charP);
  return 0;
}
```

PASSING ARRAYS TO A FUNCTION

When you pass an array to a function, the called function will receive a pointer, even if the callee has passed an array and the callee's formal argument is declared using square brackets.

```
#include<stdio.h>

void arrayProp(int arr[]) {
  int arrSize = sizeof(arr);
  printf("sizeof arr in arrayProp: %d \n", arrSize);
  arr += 1;// okay because arr is treated as pointer
  printf("2nd element of array is: %d\n", *arr);
}

int main(){
  int arr[] = {4, 1, 0, 89, 60, 2, 24};
  int sizeOfArr = sizeof(arr);
  printf("sizeof arr in main: %d \n", sizeOfArr);
  // arr += 1;// This line will give compilation error because arr is an array
  // printf("second element of array is: %d \n", *arr);

  arrayProp(arr);
  return 0;
}
```

The output on my system is:

```
sizeof arr in main: 28
sizeof arr in arrayProp: 8
2nd element of array is: 1
```

ARRAYS OF DYNAMIC SIZE

Just the way, we have been initialising arrays of constant size, we can also initialise arrays of dynamic size. //TODO: performance analysis

```c
#include<stdio.h>

int main() {
  int n;
  printf("Enter the size of array you want: ");
  scanf("%d", &n);
  int myArr[n];

  for(int i = 0; i < n; i++) {
    myArr[i] = 30 + i;
  }
  for(int i = 0; i < n; i++) {
    printf("Value at index %d is: %d\n", i, myArr[i]);
  }
  return 0;
}
```

There is one more way to initialise an array of dynamic size. That is though pointers.

```c
#include<stdio.h>
#include<stdlib.h>

int main(){
  int *p;
  int n;
  printf("Enter the size of array you want: ");
  scanf("%d", &n);
  p = (int *)malloc(sizeof(int *) * n);

  for(int i = 0; i < n; i++) {
    p[i] = 30 + i;
  }
  for(int i = 0; i < n; i++) {
    printf("Value at index %d is: %d\n", i, p[i]);
  }
  return 0;
}
```

Some time back (not long back), dynamic sized arrays were possible only through pointer method. So, if you are writing code that will be compiled on old compilers, then you should prefer pointer method.

MULTIDIMENSIONAL ARRAYS

Arrays don't need to be just one dimensional. It can be 3D, 4D and so on. A 2D array is called a matrix. Let's look at the syntax for initialising a 2D array.

```
#include<stdio.h>

int main() {
  int my2DArr[2][3] = {
    { 2, 5, 3 },
    { 5, 2, 5 }
  };
  printf("my2DArr: %d\n", my2DArr[1][2]);
  //
  int myNext2DArray[2][3] = {1, 5, 4, 0, 7, 3 };
  printf("myNext2DArray: %d\n", myNext2DArray[1][2]);
  //
  int myThird2DArr[][2] = {
    2, 3, 4, 7
  };
  printf("myThird2DArr: %d\n", myThird2DArr[1][1]);
  return 0;
}
```

The way we have declared my2DArr is the most popular and recommended way of declaring 2D arrays. But all the syntaxes shown above are valid syntaxes.

Notice the way we declared myThird2DArr array, we omitted the size of first dimension. In multi-dimensional arrays, if we are initialising an array while declaring, then we can skip the size of first dimension. C will infer it from the number of elements provided.

3. STRUCTURE

Arrays are a way to handle similar data, but what if the data is not simple like an integer or character. How will we store data of a book, such as publisher, price etc. They are all related, but their data type is not similar.

WHY STRUCTURES?

To handle data, which are not similar, but are closely related, C provides us with structures. The way integers and characters are a data type that are defined by C, structures are user defined data type. Once you have created a data type using struct, you can then go on to create variables and even arrays of this data type.

Let's see how we can create a struct.

```
#include<stdio.h>
#include<string.h>

struct book {
  char title[20];
  float price;
  char author[20];
};

int main() {
  struct book c_programming;
  strcpy(c_programming.author, "Ravi Shankar");
  strcpy(c_programming.title, "C Programming");
  c_programming.price = 999;

  printf("Books details are: \nTitle: %s\nAuthor: %s\nPrice: Rs %.2f\n",
c_programming.title, c_programming.author, c_programming.price);
  return 0;
}
```

As you can see, we created a data type called book. And we created a variable called c_programming of this data type. And individual members of c_programming are accessed by using a dot (.) notation.

At the beginning, structures seems a bit overwhelming. But, with time, it becomes very easy. This topic is very important, because structures are heavily used when we implement data structures using C/C++.

ARRAY OF STRUCTURES

The way we create arrays of other data types, we can also create arrays of structs. Let's see an example to understand this.

```c
#include<stdio.h>
#include<string.h>

struct book {
  char title[20];
  float price;
  char author[20];
};

int main() {
  struct book my_books[2];

  for(int i = 0; i < 2; i++) {
    printf("Enter title, author and price of the book(separated by space): ");
    scanf("%s %s %f", my_books[i].title, my_books[i].author, &my_books[i].price);
  }

  for(int i = 0; i < 2; i++) {
    printf("Books details are: \nTitle: %s\nAuthor: %s\nPrice: Rs %.2f\n",
my_books[i].title, my_books[i].author, my_books[i].price);
  }

  return 0;
}
```

STRUCTURE DECLARATION

When we declare structure, it doesn't reserve any space in memory. A space in memory is only reserved when we declare a variable of the above structure type.

The following, will only define the form of a user defined data type:

```c
struct book {
  char title[20];
  float price;
  char author[20];
};
```

While declaring structures, we can also declare variables of this data type. Let's see an example:

```c
#include<stdio.h>
#include<string.h>

struct book {
  char title[20];
  float price;
  char author[20];
} book1, book3, *book3;

struct {
  char name[20];
} nameStruct;
```

As you can see that, while declaring structure book, we also declared three two variables and one pointer of type 'struct book'.

The second struct that we declared, we didn't give it a name. But, we created a variable, 'nameStruct' of this type.

NESTING OF STRUCTURE

Well, a structure can have a variable of type structure which can again have a variable of type structure and so on. Let's understand through an example.

```c
#include<stdio.h>
#include<string.h>

struct Address {
  char village[20];
  char state[20];
};

struct Person{
  char name[20];
  struct Address address;
};
```

STRUCTURE COPY

A structure can be copied into another structure of same type in two ways, piece-meal or at one shot. Let's see an example:

```c
#include<stdio.h>
#include<string.h>

struct book {
```

```c
  char title[20];
  float price;
  char author[20];
};

void displayBookInfo(struct book b) {
  printf("Books details are: \nTitle: %s\nAuthor: %s\nPrice: Rs %.2f\n", b.title, b.author,
b.price);
}

int main() {
  struct book book1, book2, book3;
  strcpy(book1.author, "Ravi Shankar");
  strcpy(book1.title, "C Programming");
  book1.price = 999;

  book2 = book1;

  // book3.author = book1.author; // This will cause error
  strcpy(book3.author, book1.author);
  strcpy(book3.title, book1.title);
  book3.price = book1.price;

  printf("\nDisplaying details of book1\n");
  displayBookInfo(book1);

  printf("\nDisplaying details of book2\n");
  displayBookInfo(book2);

  printf("\nDisplaying details of book3\n");
  displayBookInfo(book3);


  return 0;
}
```

The output would be:

```
Displaying details of book1
Books details are:
Title: C Programming
Author: Ravi Shankar
Price: Rs 999.00

Displaying details of book2
Books details are:
Title: C Programming
Author: Ravi Shankar
Price: Rs 999.00
```

```
Displaying details of book3
Books details are:
Title: C Programming
Author: Ravi Shankar
Price: Rs 999.00
```

PASSING STRUCTURE TO A FUNCTION

The way we pass normal variables to a function, struct can also be passed. Please refer to the example in the last section to understand it better.

4. LINKED LISTS