Returning arrays

Returning an array is similar to passing the array into the function. The name of the array is returned from the function.

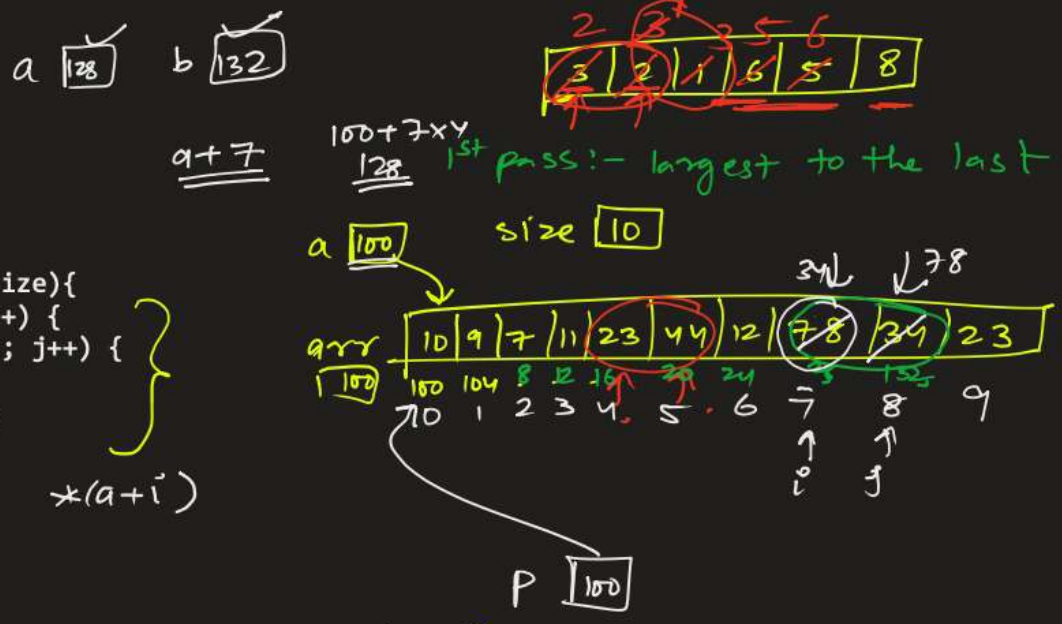
```
int * Function_name() {
   //some statements;
   return array_type;
}
```

```
#include<stdio.h>

void swap(int* a, int* b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int* Bubble_Sort(int a[], int size){
    for(int i = 0; i<size-1; i++) {
        for(int j = i+1; j<size; j++) {
            if(a[j] < a[i])
                swap(a+i, a+j);
        }
    }
    return a;
}

void main(){
    int arr[10] = { 10, 9, 7, 11, 23, 44, 12, 78, 34, 23};
    int* p = Bubble_Sort(arr, sizeof(arr)/sizeof(int));
    for(int i=0; i<10; i++) {
        printf("%d ",*(p+i));
    }
}
```



Function Pointers

The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer.

```c
#include<stdio.h>
int main(){
    printf("Address of main() function is %p", main);
    return 0;
}
```

name of the function → address of the function

Declaration of function pointers

```c
type (*ptr_name)(type1, type2…);

float (*fp) (int , int);    // Declaration of a function pointer.
float func( int , int );    // Declaration of  function.
fp = func;                  // Assigning address of func to the fp pointer.
```

int * p :- ptr to an integer

float (*p) (int, int)

() [] left to right

*

p is a pointer to a function which takes two integers as arguments and returns a float

Calling function pointers

```
result = func(a , b);      // Calling a function using usual ways.
result = (*fp)( a , b);    // Calling a function using function pointer.
result = fp(a , b);        // Indirection operator can be removed.
```

```
#include<stdio.h>
int add(int a, int b){
    return a+b;
}
int sub(int a, int b){
    return a-b;
}
int mul(int a, int b){
    return a*b;
}
int div(int a, int b){
    return a/b;
}

int perform(int a, int b, char op){
    int (*opn) (int, int);
    switch(op) {
        case'+':
            opn = add;
            break;
        case'-':
            opn = sub;
            break;
        case'*':
            opn = mul;
            break;
        case'/':
            opn = div;
    }
    return opn(a, b);
}

int main(){
    int a = 3, b = 5;
    char op = '/';
    printf("Result of %d %c %d = %d", a, op, b, perform(a, b, op));

    return 0;
}
```

add → 100

sub → 200

mul → 300

div → 400

perform → 500

a = 3, b = 5, op = '/'

int (fun *) (int, int)

opn → 400 (8B)

(*opn)(a,b);

opn(a,b) a op b

3/5 3+5

+, -, *, /

a 3 b 5 op '/' +

Passing function pointers as arguments

```c
#include<stdio.h>
void func1(void (*)());
void func2();

int main(){
    func1(func2);
    return 0;
}

void func1(void (*ptr)()){
    printf("Function1 is called \n");
    (*ptr)();
}

void func2(){
    printf("Function2 is called \n");
}
```

declaration of func1 → function which takes a (function pointer with no arguments and returns nothing) as argument and returns nothing

declaration of func2

No arguments and returns nothing

≡ func2()

void fun()

ptr | 300 (8B)

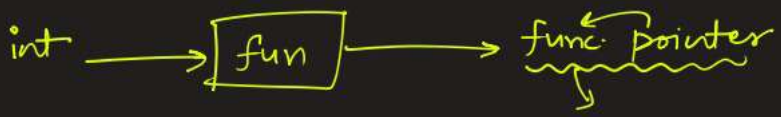
func1 | 200

func2 | 300

Returning function pointers

( ) highest
left to right

Design A function fun which takes an integer as arguments a returns (a function pointer which takes char as argument and returns float)



( )

char argument
float return

```
float f1 (char a){
    //--- blah
}
```

definition of fun

```
float (*fun (int x)) (char a) {
```

inside out

```
ptr to f1
fun (int x) {
    // logic
    return f1;
}
```

```
return f1;
}
```

C Programming Lecture 11

Friday, 21 June 2024 8:17 PM

Functions in C

Calling a function

Constant Arguments

- Function arguments can be declared as const
- A const variable is one whose value cannot be changed once initialised
- To make variable const in the function, put keyword const corresponding the concerned argument's formal parameter

```
Function definition
int function(const int a, char b, float c, int d)
{
      a+=10;
      return a+b+c+d;
}
Function call
int out = function(5, 1.167, 'a', 2021)
```

float char

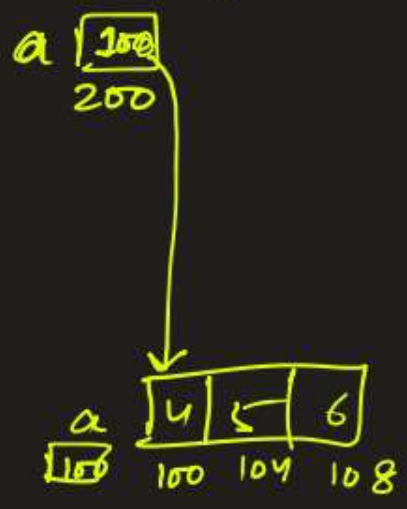
a = 5 c 'a'

b 1.167 d 2021

error: cannot assign to variable 'a' with const-qualified type

```
int fun ( int *a, int n){
    sizeof (a)  8B
    a[0]  a[1]  a[2]
    *(a+0)
}
int main (){
    int a[3] = {4, 5, 6};
    fun (a, 3);
        100
    sizeof (a)   12B
}
```

int a[] int a[3]



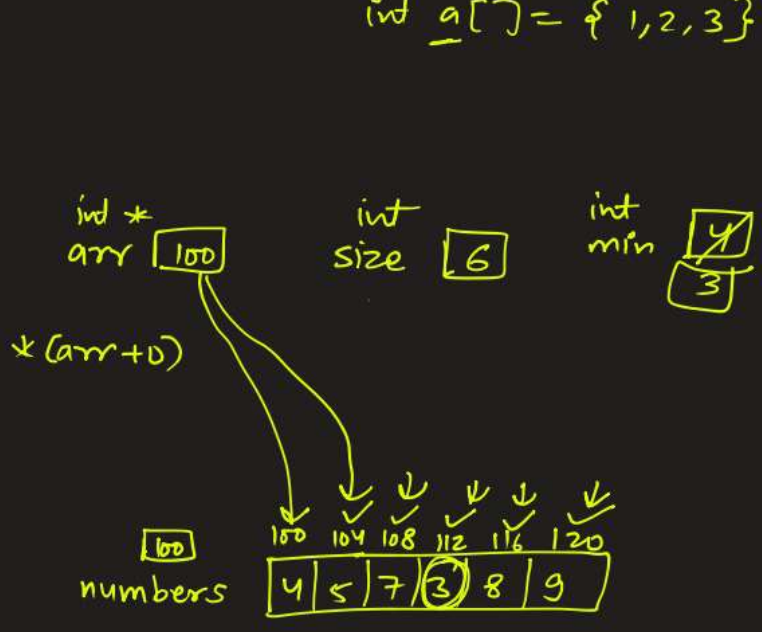
Passing arrays

As we know that the array_name contains the address of the first element. Here, we must notice that we need to pass only the name of the array in the function which is intended to accept an array.

The array defined as the formal parameter will automatically refer to the array specified by the array name defined as an actual parameter.

```
return_type function(type arrayname[])
return_type function(type arrayname[SIZE])
return_type function(type *arrayname)

functionname(arrayname);//passing array
```




```
#include<stdio.h>
int minarray(int arr[], int size){
    int min = arr[0];
    int i = 0;
    for(i = 1; i<size; i++){
    if(min>arr[i])
        min=arr[i];
    }
    return min;
}

int main(){
    int i=0, min=0;
    int numbers[] = {4,5,7,3,8,9};

    min = minarray(numbers, 6); //passing array with size
    printf("minimum number is %d \n",min);
    return 0;
}
```

```c
#include<stdio.h>
void fun(int a[5], int b[], int* c){
    printf("%d, %d \n", sizeof(a), a[0]); // size of int ptr, 1st element
    printf("%d, %d \n", sizeof(b), b[1]); // size of int ptr, 2nd element
    printf("%d, %d \n", sizeof(c), c[2]); // size of int ptr, 3rd element
}
int main(){
    int a[5] = {1,2,3};
    printf("%d, %d \n", sizeof(a), a[0]); // size of array a, 1st element
    int b[] = {1,2,3};
    printf("%d, %d \n", sizeof(b), b[1]); // size of array b, 2nd element
    int* c = a;
    printf("%d, %d \n", sizeof(c), c[2]); // size of int ptr, 3rd element
    fun(a, b, c);

    return 0;
}
```