Day: Date: / /

Functions

We ~~have~~ already know what functions are and we have used void functions so we are going to start:

Function with parameters/arguments;

Normally we had worked with global variables if we wanted to change something every where in the code but those are not a good practice to use.

So to send a local variable to another function we have arguments.

e.g we have used the pow function

```
pow(2, 3);
```

Arguments

Arguments and Parameters same things.

- How to declare a function with args?

Next page.

Date: / /

~~Function declaration:~~

void & identifier (parameter / ~~argument~~)

void → Datatype
identifier → name
parameter → as many as want.

e.g

void sum (int a, int b).

- The ~~argument~~ Parameter can have any data type regardless of the function data type

  e.g int sum (string a , char b, float c);

- As many as want.

- Always in order that is pow(2, 3) and pow(3, 2) give different answers.

Prototype:

If we are making a prototype then we don't need to write the variable name

ie

void sum (int , int);

enough, you can write ~~also~~

Yousaf

Day: Date: / /

- we can even change the variable name and program still works. e.g

```
void sum (int c, int d);
int main () {
    body
}
void sum (int a, int b){
    cout<< a+b << endl;
}
```

allowed but don't use, will mess up your code.

Passing by value:

- Here is I did say that parameters and arguments are same, though they are very close, they are NOT SAME.

Parameters are values of the functions.
Arguments are values that we put in function.

Parameters can change in a function but arguments will remain constant.

e.g

```
sum (a, b);
}
int sum ( int x, int y) {
}
```

arguments

parameters

Date: / /

But how does this happen?:

* From arguments to parameters only values are passed.,

this is called pass by value.

so only the value is sent from the argument to the parameters NOT the variable.

Function Datatypes and returning values:

Function can also be of different data types e.g int, double, float, char or string even unsigned long long int () { }.

* when we declare a function except void we need to return that particular datatype e.g

```
int sum () {
    return any integer value;
}
```

Yousaf

Date: / /

```
char ch() {
    return character or ofc a number as char is an int
}

string sh() {
    return string;
}
```

If we don't return these values than an error or warning will come.

* Return is the value passed back to the original place e.g:

```
int main() {
    int a, b;
    cin >> a >> b;
    int c = sum(a, b);      // This returns x+y which are a+b
    cout << c << endl;      // c = a+b
}
int sum(int x, int y) {
    return x+y;
}
```

= A function can have ONLY ONE return.

Day: Date: / /

* where ever there is a return the function will end and return to the true place.

* if a function has more than one return then function will go back at the first return.
or we can add conditions

e.g
```
if (a>b)
    return a*b;

return a + b;
```

Calling a returning functions:

whenever we call a value returning function we need to add all the arguments
i.e arguments = parameters.

e.g
if sum has two parameters we will write
sum (a,b) ✓
sum (a) ✗ } not allowed
sum (a,b,c) ✗

Yousuf

Date / /

Important points

1) we can make different functions of the same name.

HOW?

number of parameters OR
the types of parameters
should be different.

ie

```
int sum ( int x, int y ) {
        }                          ✓ allowed.
float sum ( float x, float y) {
        }
```

How? if the arguments are float then float sum will run if arguments are int then

But in the case and only in this case if even x's or y's [arguments] are different eg One is int and other is float it is an ERROR. only if we use variable ie sum (a, b) X wrong
int float

sum (4, b) ✓ allowed
float function will run because

Day: Date:

2) Overloading:

e.g If we want the sum of 2 and sometimes 3 numbers then what?

we can do:

```
int sum (int a, int b, int c){

}
int sum (int a, int b) {

}
```

allowed

This is called over loading.

3) A Better approach to over loading:
(Default arguments).

we can declare one function & do the same as over loading. HOW?

default arguments

```
int sum (int a, int b, int c=0, int d=0){

    return a+b+c+d;
}
```

a sum will have a minimum of two value so a and b will be covered and if he enter a 3rd value that will be the value of c. if he doesn't enter a value then c and d will have 0 value.

Yousaf

Day: Date: / /

e.g

sum (2,3,2) // returns 7 — allowed
sum (2,3) // returns 5 —
sum (2,3,5,7) // returns 17 —

sum (2) // ✗ Not allowed
This is not allowed as b was not initialized.

Remember, start initializing from right side.

```
sum (int a=0, int b, int c) {
}
```

This works but doesn't make sense.

Static local variables:

These are a great replacement for global variables. Whenever a function terminates a variable is destroyed but what if we want to keep the variable. We have Static local variables.

Next page

Example

```
void showlocal ( ) {
    static int local = 2;
    local++;
    cout << local << endl;
}
int main ( ){
    local ( );
    local ( );
    local ( );
    local ( );
}
```

output:
3
4
5
6

- A static variable is initialized only once in the whole program.
- It's scope is local and can only be used in the particular function.
- We can have SAME Name & static local in different functions and they will work seperately without harming each other.

Yousaf

Day: Date: / /

e.g

```
void local1(){
    static int local = 1;
    local ++;
    cout << local << endl;
}

void local2(){
    static int local = 1;
    local ++;
    cout << local << endl;
}

int main(){
    local1();
    local2();
    local1();
    local1();
    local2();
}
```

output

2 - 1
2 - 2
3 - 1
4 - 1
3 - 2

same with each other

same was incremented

* If any static variable is not initialized it is initialized with 0.

Day: Date: / /

Scope Resolution Operator (::)

- if we have a global variable:

```
int num = 5;
```

and then we have a local variable.

```
int num = 2;
```

Then in the function if write:

```
cout << num << endl;
```

result: 2

- But if want to use the global num then:

```
cout << ::num << endl;
```

result: 5

:: is the scope resolution operator.

- This can only provide global variables. if no global variable exist of that particular name it doesn't work.

- This works only with global, it will NOT work like:

```
{ int num = 7;
    { int num = 4;
      cout << ::num;   X error as there is no global num.
    }
}
```

Yousaf

Day: Date: / /

Pass by Reference:

- Pass by value creates a new variable and we require a return type to return the value. This wastes memory.
- What if we could alter the same variable?
- That is what pass by reference does.

e.g

```
void fun2(int &x);
int main(){
    int a = 5;
    fun2(a);

    cout << a << endl;
}

void fun2(int &x){

    x *= 2
    x += 2
}
```

Terminal:

12

&, This symbol ampersent, sends the memory address.

now x can be said to be an alias of a.

x is the same memory address as a so whatever we do to x changes a.

Yousuf

Day: Date: / /

- so in reality x doesn't take any more space, as it is just representing a.

- We can also make functions like:

```
int & Hello (int &x , int &y, int sum){

    return ---
}
```

in this ~~sort~~ function if I return return sum; then it will be an ERROR or dangling reference as sum is a local variable and is de allocated once the function ends.

// You will understand more in pointers.

Yousaf

Day: Date: / /

Passing Arrays:

Now we know how to pass a certain variable but how to pass an array

-Arrays can only be passed by reference

~~Passing an arr~~

Function name and parameters when passing array:

```
void display (int xyz[], int sze)
```

Calling a Function:

```
int main (){

    int arr[5] = {1,2,3,4,5};

    display (arr, 5);

}
```

Prototypes:

In prototypes for array, we don't need to write the name and can even change it but we need to write "[]".

```
void display ( int [], int );
```
✓ correct

Yousaf

Day: Date:

Writing more than 1D arrays in arguments.

- For writing more than 1D arrays we also need to include the size in brackets.

e.g

```
void display ( int arr[][5] , int rows)      ✓ allowed.

void display (int arr [2][5])                ✓ allowed.

void display ( int arr [][] ,int rows, int cols)   ✗ not allowed.

void display ( int arr [2][] , int cols)     ✗ not allowed

void display (int arr [][1][5], int row)     ✓ allowed.

void display ( int arr [2][1][5] )           ✓ allowed.

void display ( int arr [][][1] , int rows, int cols)   ✗ not allowed

void display ( int arr [][5][], int row, int col-1)    ✗ not allowed.
```

- In 2D arrays or bigger only first '[]' can be left empty, others need to have a number.

Yousaf