Fundamentals of Programming in C++
CoSc 2031

CHAPTER 1
Functions in C++

Mekonnen K. (MSc)

Email: mekonnen307@gmail.com

Course Content

- Basic concept and need of function
- Declaring and defining a function
- Function components (parameters and arguments)
- Calling /invoking function by value and reference parameters
- Functions Recursion

Function

- Functions are building blocks of the programs.
- A **function** in the broader sense may be considered as a component of a big program. A large program may be divided into suitable small segments and each segment may be treated and defined as a function. These segments can be separately compiled, tested and verified.
- Function is a self-contained routines within a larger program that carry out specific tasks.
- One entity in programming which have a set of– One entity in programming which have a set of command to carry out specific task
- Sub-routine
- We can call these function every time we need (in programming sequences) – reusable.

Function

- All C++ programs must contain the function main( ).
- The execution of the program starts from the function main( ).

Why we need function?

- More manageable
- Code are easier to maintain
- Reduction in program size.
- Code duplication is avoided.
- Code reusability is provided.
- Functions can be called repetitively.
- Increase program readability.
- Divide a complex problem into many simpler problems.
- Reduce chances of error.
- Makes modifying a program becomes easier.
- Makes unit testing possible.

When we need function?

- When you need to repeat the same process over and over in a program.
- The function can be called many times but appears in-The function can be called many times but appears in the code once.

Types of function

1. Built in functions/pre-defined function

- are part of compiler package.
- Part of standard library made available by compiler.
- Can be used in any program by including respective header file.
- User needs to include pre-defined header file (i.e. math.h, time.h)
- functions such as
  - ✓ swap (), sqrt (), exit(), pow(), strlen() etc.

Types of function Cont'd… Example

```
#include<iostream>
#include<cstring>
using namespace std;
int main()
{
    int len; char str[80];
    cout<<"Enter the string: ";
    cin.getline(str, 80);
    len = strlen(str);
    cout<<"\nLength = "<<len;
    cout<<endl;
    return 0;
    }
```

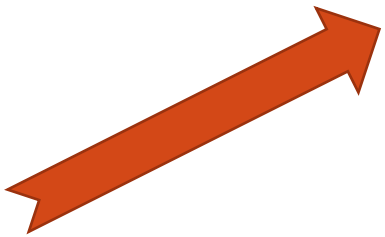
Types of function Cont'd…

2. User defined function

- The functions which are declared and defined by a programmer are called user defined functions or programmer defined functions.
- Created as per requirement of the program.

Types of function Cont'd… Example

```
#include<iostream>
using namespace std;
int stringLength(char []);
int main()
{
   int len;
   char str[80];
   cout<<"Enter the string: ";
   cin.getline(str, 80);
   len = stringLength(str);
   cout<<"\nLength = "<<len;
   cout<<endl;
   return 0;
}
```



```
int stringLength(char x[])
{
   int i=0, count=0;
   while(x[i] )
   {
      count++;
      i++;
   }
   return count;
}
```

Types of User defined function

- user-defined functions can be categorized as:
  - Functions with no parameters and no return value
    - void add(void);
  - Functions with no parameters and with return value
    - int add(void);
  - Functions with parameters and with no return value
    - void add(int,int);
  - Functions with parameters and with a return value
    - int add(int,int);

Function - Elements

- Three main elements in function :
  1. Prototype of Function / Declaration
  2. Define the function
  3. Called the function

Function Declaration/Prototype

- A function prototype is a declaration of a function that tells the program about the type of value returned by the function, name of function, number and type of arguments.

Syntax: Return_type function_name (parameter list/argument);

Example:

```
int add(int,int);

void add(void);

int add(float,int);
```

- The names of parameters may or may not be given in the prototype, however the names are necessary in the head of function in the function definition.
- No function can be declared within the body of another function.

Function Definition

- Function definition consists of a function header that identifies the function, followed by the function body containing the executable code for that function.
- Syntax:

```
return_type function_name(data_type variable1, data_type variable2,..)
        { //function body
                .............
                Statements
                .............
                return( variable);
        }
```

Function Definition Cont'd…

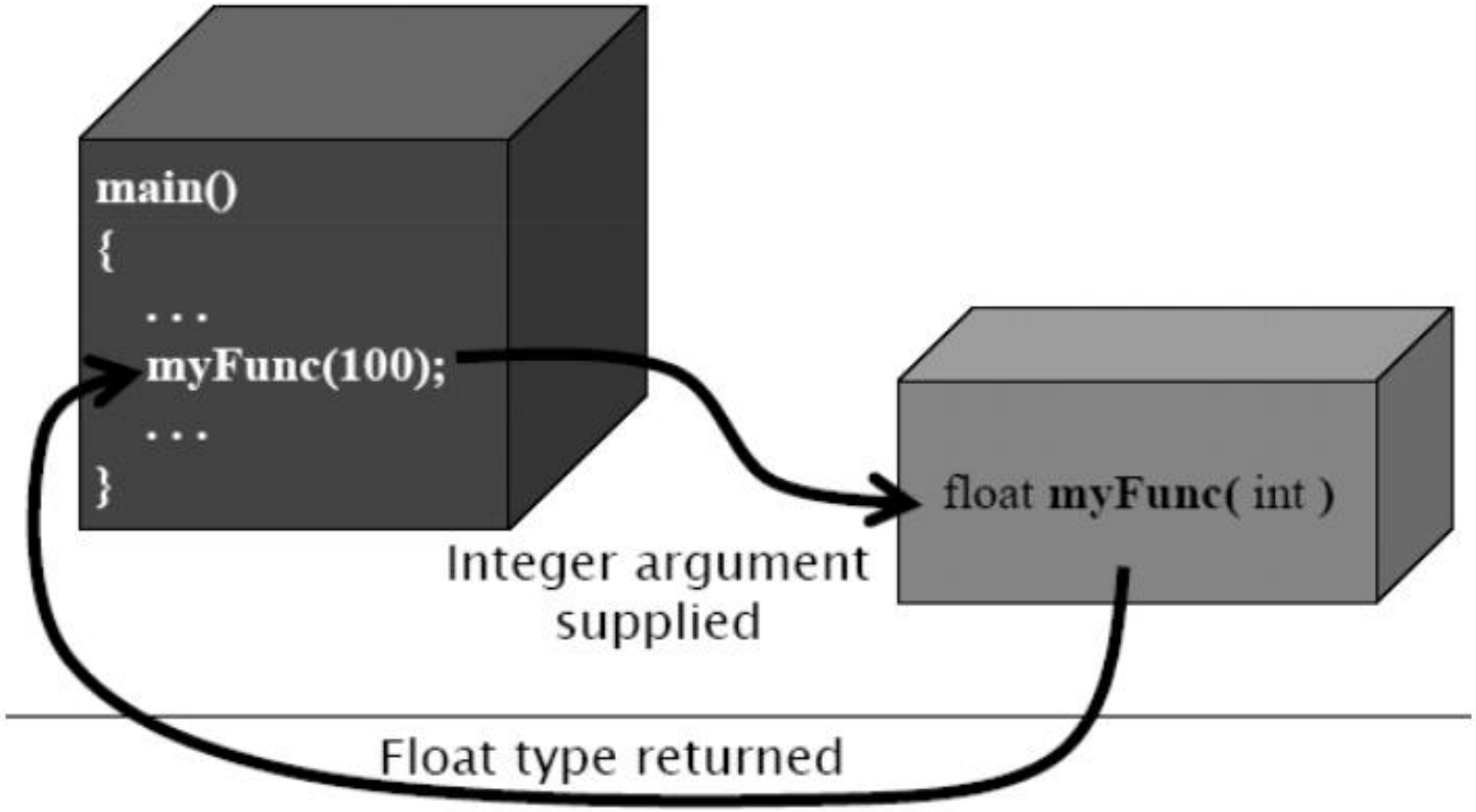
- When a function defined, space is allocated for that function in the memory.
- The number of and the order of arguments in the function header must be same as that given in function declaration statement.

Function Call

- The function call statement invokes the function.
- When a function is invoked the compiler jumps to the **called function** to execute the statements that are a part of that function.
- Once the called function is executed, the program control passes back to the **calling function**.

Syntax: function_name(variable1, variable2, ...)

Function Call Cont'd…



Function Call Cont'd…

- Names (not the types) of variables in function declaration, function call and function definition may vary.
- Arguments may be passed in the form of expressions to the **called function**.
- In such a case, arguments are first evaluated and converted to the type of formal parameter and then the body of the function gets executed.
- If the return type of the function is not void, then the value returned by the called function may be assigned to some variable as given below.

  variable_name = function_name(variable1, variable2, …);

Function Call Cont'd…

- The terms calling function and called function are derived from the telephone communication. The one who rings the number is the calling person and one who receives the call is the one called. The same terminology applies to functions.
- Thus the function which calls another function is the **calling function** and the function which is called is the **called function**.
- The different data variables that a function accepts for processing are called **parameters** of the function.
- The values that represent the parameters and are passed on to the function when it is called, are the **arguments** of the function.
- Some functions return a numeric value to the calling function. These are called **return type functions**. If a function does not return a numeric value we call it **void function**.

Function Call Cont'd…

- **Given the next program, which function is the calling function and which is the called function?**

```
#include<iostream.h>
#include<conio.h>
void nextMsg();
int main()
{
        cout<< "Hello!\n";
        nextMsg();
        cout<<"World\n";
        return 0;
}
void nextMsg() {
        cout<< "GoodBye!\n";
        return; }
```

Return Statement

- The return statement is used to terminate the execution of a function and return control to the calling function.
- When the return statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.
- A return statement may or may not return a value to the calling function.

  Syntax: return <expression> ;

- The value of expression, if present, is returned to the calling
- function. However, in case expression is omitted, the return value of the function is undefined.

Return Statement

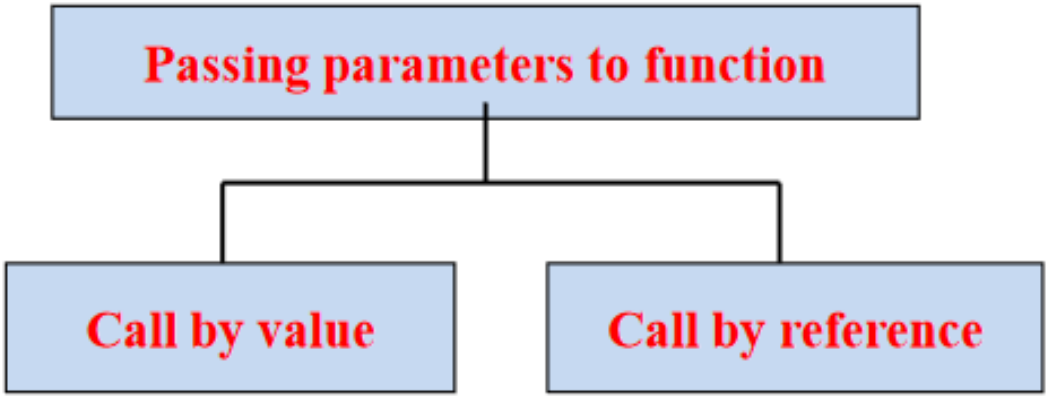
- The return statement is used to terminate the execution of a function and return control to the calling function.
- When the return statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.
- A return statement may or may not return a value to the calling function.

  Syntax: return <expression> ;

- The value of expression, if present, is returned to the calling
- function. However, in case expression is omitted, the return value of the function is undefined.

Passing Parameters To The Function

- There are two ways in which arguments(parameters) can be passed to the called function.



- Call by value in which values of the variables are passed by the calling function to the called function.
- Call by reference in which address of the variables are passed by the calling function to the called function.

Call by value

- The called function creates new variables (formal arguments) to store the value of the arguments passed to it. Therefore, the called function uses a separate copy of the actual arguments (formal arguments) to perform its intended task.

- If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function no change will be made to the value of the variables.

Call by value Cont'd…

- For instance:

```
#.....
void Foo(int num)
{
        Num = 0;
        cout<< "num = " << num << " \n";
}
int main(void)
{
        int x = 10;
        Foo(x);
        cout<< "x = "<<x<< "\n";
        getch();
        return 0;
}
```

Call by value Cont'd…

- When the function is called and x passed to it, num receives a copy of the value of x. As a result, although num is set to 0 by the function, this does not affect x.
- The program produces the following output:

  Num = 0

  x = 10

- Passing arguments in this way, where the function creates copies of the arguments passed to it is called passing by value.

Call by value Cont'd…

```
/* program to illustrate the concept of call by value */
#include<iostream.h>
#include<conio.h>
void add(int,int);
int main()
{
     int a,b;

     cout<<"enter values of a and b"<<endl;
     cin>>a>>b;
     add(a,b);
     return 0;
}
void add(int x,int y)
{
int c;
     c=x+y;
     cout<<"addition is"<<c;
}
```

Call by Reference

- In call by value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement.
- The better option when a function can modify the value of the argument is to pass arguments using call by reference technique.
- In call by reference, we declare the function parameters as references rather than normal variables.
- Any changes made by the function to the arguments it received are visible by the calling program.
- To indicate that an argument is passed using call by reference, an ampersand sign (&) is placed after the type in the parameter list.

Call by Reference

- In call by value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement.
- The better option when a function can modify the value of the argument is to pass arguments using call by reference technique.
- In call by reference, we declare the function parameters as references rather than normal variables.
- Any changes made by the function to the arguments it received are visible by the calling program.
- To indicate that an argument is passed using call by reference, an ampersand sign (&) is placed after the type in the parameter list.

Call by Reference Cont'd…

Program to illustrate call by reference

```
#include<iostream.h>
#include<conio.h>
void swap(int &,int &);
int main()
{
int a,b;
cout<<"enter the values of a and b";
cin>>a>>b;
cout<<"before swaping";
cout<<"A"<<a;
cout<<"B"<<b;
swap(a,b);
cout<<"after swaping";
cout<<"A"<<a;
cout<<"B"<<b;
getch();
}

void swap(int &X,&Y)
{
int temp;
temp=X;
X=Y;
Y=X;
}
```

Call by Reference Cont'd…

```
void Foo(int & num)
{
        num = 0;
        cout<< "num = " << num << " \n";
}
int main(void)
{
        int x = 10;
        Foo(x);
        cout<< "x = "<<x<< "\n";
        getch();
        return 0;
}
```

Call by Reference Cont'd…

- Any change made on num will be effected to x. Thus, the program produces the following output:

num = 0

x = 0

Function With Default Arguments

- C++ allows to call a function without specifying all its arguments.
- In such cases, the function assigns a default value to a parameter which does not have a matching arguments in the function call.
- Default values are specified when the function is declared. The complier knows from the prototype how many arguments a function uses for calling.

Function With Default Arguments Cont'd...

Example : float result(int marks1, int marks2, int marks3=75);

- A subsequent function call

```
average = result(60,70);
```

passes the value 60 to marks1, 70 to marks2 and lets the function use default value of 75 for marks3.

- The function call

```
average = result(60,70,80);
```

passes the value 80 to marks3.

Function With Default Arguments Cont'd…

- Example

```
using namespace std;
void fun(int a=10, int b=20);     //Declaration
OUTPUT
int main()
{
        fun(3,5);//Function Calling
        fun(3);//Function Calling
        fun();//Function Calling
        return 0;
}
void fun(int a, int b)//Definition
{
        cout<<"a:"<<a<<endl;
        cout<<"b:"<<b<<endl;
}
```

```
a:  3
b:  5
a:  3
b: 20
a: 10
b: 20
```

Global versus Local variables

Local Variable

- A variable defined inside a function (defined inside function body between braces) is called a local variable or automatic variable.
- Its scope is only limited to the function where it is defined. In simple terms, local variable exists and can be accessed only inside a function.
- The life of a local variable ends (It is destroyed) when the function exits.
- Each block in a program defines a local scope. Thus the body of a function represents a local scope. The parameters of a function have the same scope as the function body.

Global versus Local variables Cont'd…

- Variables defined within a local scope are visible to that scope only. Hence, a variable need only be unique within its own scope.
- Local scopes may be nested, in which case the inner scope overrides the outer scopes.

Eg:

```
int xyz;//xyz is global
void Foo(int xyz)//xyz is local to the body of Foo
{
        if(xyz > 0)
        {
        Double xyz;//xyz is local to this block
        }
}
```

Global versus Local variables Cont'd…

Global Variable

- If a variable is defined outside all functions, then it is called a global variable.
- The scope of a global variable is the whole program. This means, It can be used and changed at any part of the program after its declaration. Likewise, its life ends only when the program ends.

Global versus Local variables Cont'd…

- Everything defined at the program scope level (outside functions) is said to have a global scope, meaning that the entire program knows each variable and has the capability to change any of them.

Eg.
```
int year = 1994;//global variable
int max(int,int);//gloabal funcrion
int main(void)
{
        //…
}
```

Global versus Local variables Cont'd…

Example:-

```
#include<iostream>
using namespace std; //Declaring a global variable
int a = 5;
int main()
{
        //Printing the value of a
        cout<<"The value of a is "<<a<<"\n";
        //Modifying the value of a
        a--;
        //Printing the value of a
        cout<<"The value of a is "<<a<<"\n";
        return 0;
}
```

Scope Operator

- Because a local scope overrides the global scope, having a local variable with the same name as a global variable makes the latter inaccessible to the local scope.
- Eg

```
int num1;
void fun1(int num1)
{
        //…
}
```

- The global num1 is inaccessible inside fun1(), because it is overridden by the local num1 parameter.

Scope Operator

- This problem is overcome using the scope operator "::" which takes a global entity as argument.

```
int num1 = 2;
void fun1(int num1)
{
        //… num1=33;
        cout<<num1; // the out put will be 33
        cout<<::num1; //the out put will be 2 which is the
global
        if(::num1 != 0)//refers to global num1
                //…
}
```

Automatic versus static variables

- The terms **automatic** and **static** describe what happens to local variables when a function returns to the calling procedure.
- By default, all local variables are automatic, meaning that they are erased when their function ends. You can designate a variable as automatic by prefixing its definition with the term **auto**.

Eg. The two statements after main()'s opening brace declared automatic local variables:

```
main()
{
        int i;
        auto float x;
        …
}
```

Automatic versus static variables

- The opposite of an automatic is a static variable. All global variables are static and, as mentioned, all static variables retain their values. Therefore, if a local variable is static, it too retains its value when its function ends-in case this function is called a second time.
- To declare a variable as static, place the **static** keyword in front of the variable when you define it.

Automatic versus static variables

- **Static variables** can be declared and initialized within the function, but the initialization will be executed only once during the first call.
- If **static variables** are not declared explicitly, they will be declared to **0** automatically.
- Eg.

```
void my_fun()
{
        static int num;
        static int count = 2;
        count=count*5;
        num=num+4;
}
```

Automatic versus static variables

- In the above example:
  - ✓ During the first call of the function my_fun(), the static variable count will be initialized to 2 and will be multiplied by 5 at line three to have the value 10. During the second call of the same function count will have 10 and will be multiplied by 5.
  - ✓ During the first call of the function my_fun(), the static variable num will be initialized to 0 (as it is not explicitly initialized) and 4 will be added to it at line four to have the value 4. During the second call of the same function num will have 4 and 4 will be add to it again.
  - ✓ N.B. if local variables are static, their values remain in case the function is called again.

Overloading Function

- Functions with the same name are called overloaded functions. C++ requires that each overloaded functions differ in its argument list. Overloaded functions enable you to have similar functions that work on different types of data.
- N.B: if two or more functions differ only in their return types, C++ can't overload them. Two or more functions that differ only in their return types must have different names and can't be overloaded.

Overloading Function

- Example

```
#include<iostream>
using namespace std;
void fun();
void fun(int);
void fun(int,int);
int main()
{
        fun();
        fun(2);
        fun(5,6);
        return 0;
}
void fun()
{
    cout<<"\n Function with 0 parameter:"<<endl;
}
void fun(int a);
{
    cout<<"\n Function with 1 parameter:"<<endl;
}
void fun(int a,int b)
{
    cout<<"\n Function with 2 parameter:"<<endl;
}
```

Recursion Function

- A function which calls itself is said to be **recursive**. Recursion is a general programming technique applicable to problems which can be defined in terms of themselves.
- Take the factorial problem, for instance which is defined as:
  - factorial of 0 is 1
  - factorial of a positive number n is n time the factorial of n- 1.

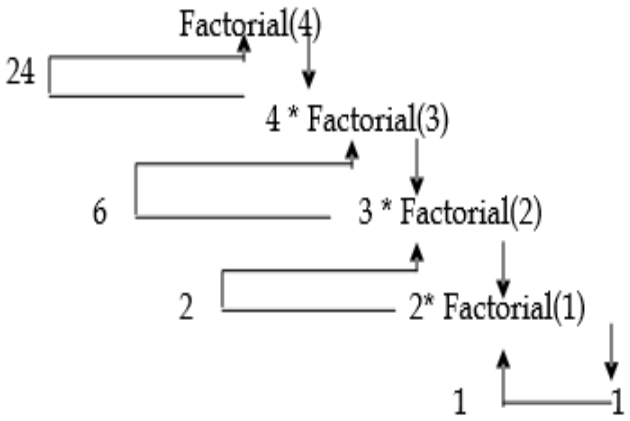
Recursion Function

- The second line clearly indicates that factorial is defined in terms of itself and hence can be expressed as a recursive function.

```
int Factorial(unsigned int n )
{
        return n = = 0 ? 1 : n * factrial(n-1);
}
```

Recursion Function

- For n set to 4, the following figure shows the recursive call:



- A recursive function must have at least one termination condition which can be satisfied. Otherwise, the function will call itself indefinitely until the runtime stack overflows.
- The three necessary components in a recursive method are:
  - A test to stop or continue the recursion
  - An end case that terminates the recursion
  - A recursive call(s) that continues the recursion

Recursion Function

Example

```
#include<iostream>
using namespace std;
void sum(int);
int main()
{
        int n;
        cout<<"Enter Numebr:";
        cin>>n
        int result=sum(n);
        cout<<"The sum of Numbers 0-"<<n<<"is:";
        cout<<result;
        return 0;
}
```

```
int sum(int n)
{
        if(n==0)
                return 0;
        else
                return n+sum(n-1);
}
```

Recursion versus iteration

- Both iteration and recursion are based on control structure. Iteration uses a repetition structure (such as for, while, do…while) and recursive uses a selection structure (if, if else or switch).
- Iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base case

Recursion versus iteration

- Recursion has disadvantage as well. It repeatedly invokes the mechanism, and consequently the overhead of method calls. This can be costly in both processor time and memory space.
- Each recursive call creates another copy of the method (actually, only the function's variables); this consumes considerable memory.
- **N.B: Use recursion if**:
  - ✓ A recursive solution is natural and easy to understand
  - ✓ A recursive solution doesn't result in excessive duplicate computation.
  - ✓ the equivalent iterative solution is too complex and
  - ✓ of course, when you are asked to use one in the exam!!

Inline Function

- C++ provides inline functions to reduce the function call overhead.
- **An inline function** is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call.
- This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

Inline Function

Syntax:

```
inline return-type function-name(parameters)
{
        // function code
}
```

```
inline void printSum(int num1, int num2){
    cout << num1 + num2 << "\n";
}
int main() {
    printSum(10, 20);
    printSum(2, 5);
    printSum(100, 400);
}
```

During Compilation

```
inline void printSum(int num1, int num2){
    cout << num1 + num2 << "\n";
}
int main() {
    cout << 10 + 20 << "\n";
    cout << 2 + 5 << "\n";
    cout << 100 + 400 << "\n";
}
```

Inline Function

- **The compiler may not perform inlining in such circumstances as:**
  - If a function contains a loop. (*for, while and do-while*)
  - If a function contains static variables.
  - If a function is recursive.
  - If a function return type is other than void, and the return statement doesn't exist in a function body.
  - If a function contains a switch or goto statement.

Passing Arrays to a Function

- You can pass array as an argument to a function just like you pass variables as arguments.
- When arrays are passed as a parameter to a function, specify the name of the array without any brackets in the function call.
- For a one-dimensional array, the function definition should include the brackets, but it is not necessary to specify the exact length of the array.

Passing Arrays to a Function Cont'd…

- For example, the following illustrates passing a single one-dimensional array to a function:

```
void SomeFunction(int arr[]) // Use empty brackets for 1-d array
{
        arr[0]=10;
}
int main()
{
        int intArray[100];
        SomeFuntion(intArray); // Leave brackets off
        cout << arr[0] << endl; // Will output 10 since array is
reference
}
```

Passing Arrays to a Function Cont'd…

- If we were to pass in the length of the array, the function prototype would look like:

```
void SomeFunction(int arr[], int length);
```

and would be invoked via:

```
int intArray[100];

SomeFunction(intArray, 100);
```

Example: Passing Single array to a function

```
// C++ Program to display single array element
   #include <iostream>
   using namespace std;
   void arr(int a )
   {
         cout << "Numbers " << a <<endl;
   }

   int main()
   {
      int myArr[] = {1, 2, 3};
      arr(myArr[1]); // pass array element myArr[1] only.
      return 0;
   }
```

Example: Passing 1D arrays to a function

```
#include <iostream>

using namespace std;
int SumValues (int [], int ); //function prototype
void main( )
{
int Array[10]={0,1,2,3,4,5,6,7,8,9};
int total_sum;
total_sum = SumValues (Array, 10); //function call
cout <<"Total sum is " <<total_sum;
}
int SumValues (int values[], int num_of_values) //function header
{
int sum = 0;
for( int i=0; i < num_of_values; i++)
sum+=values[i];
return sum;
}
```

Example: Passing 2D arrays to a function

```
// C++ Program to display the elements of 2D array by passing it to a function
#include <iostream>
using namespace std;
void display(int a[][2])
{
    cout << "The Elements are: " << endl;
    for (int i = 0; i < 2; ++i)
        {
                for (int j = 0; j < 2; ++j)
                {
                        cout << "arr[" << i << "][" << j << "]: " << a[i][j] << endl;
                }
        }
    }
int main()
{
        int arr[2][2] = { {1, 2}, {3, 4} };
        display(arr);
        return 0;
}
```

Thank You

Questions?