Tishk International University
Science Faculty
IT Department

Programming II - IT-118

Functions

1st Grade - Fall Semester

Lecture #2

Instructor: Hemin Ibrahim

Email: hemin.ibrahim@tiu.edu.iq

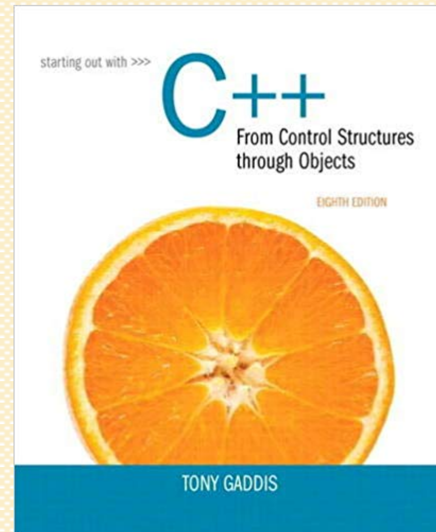
Overview

- ✓ Introduction to functions
  - ✓ Define function
  - ✓ Call function
  - ✓ Function with parameters
- ✓ Function Prototype
- ✓ Pass by Value
- ✓ The `return` Statement

Textbook Source

Tony Gaddis, Starting Out with C++: From Control Structures through Objects, 8th Edition

- ✓ **Chapter 8 (from page 305 to 363)**



People who code as a hobby

People who code for a living

Modular Programming

- Modular programming: breaking a program up into smaller units
- Function: a collection of statements to perform a specific task
- Advantages for modular programming
  - *Simplifies the process of writing programs*
  - *Improves debugging of programs*

Modular Programming

This program has one long, complex function containing all of the statements necessary to solve a problem.

```
int main()
{
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
}
```

In this program the problem has been divided into smaller problems, each handled by a separate function.

main function

```
int main()
{
   statement;
   statement;
   statement;
}
```

function 2

```
void function2()
{
   statement;
   statement;
   statement;
}
```

function 3

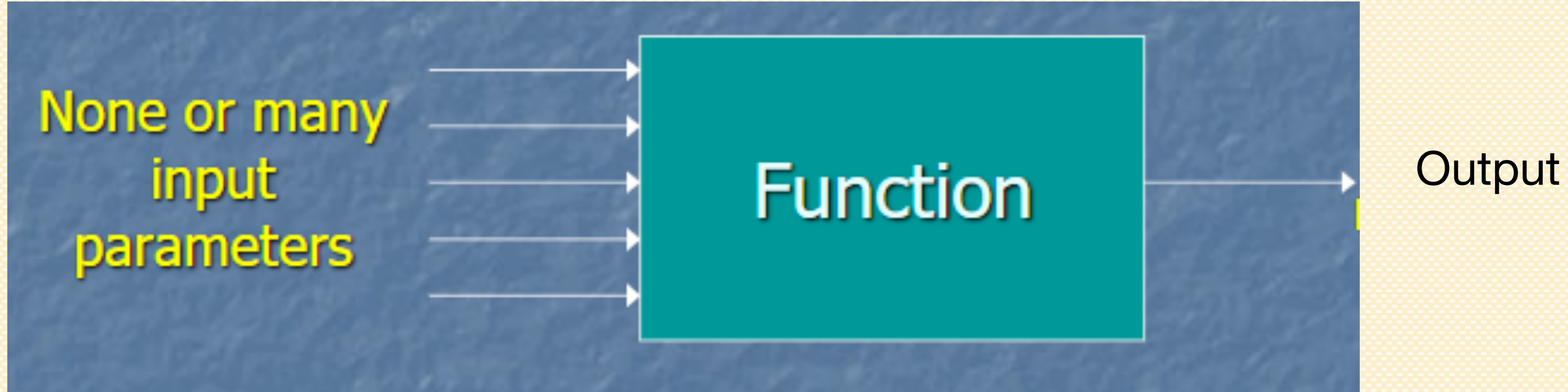
```
void function3()
{
   statement;
   statement;
   statement;
}
```

function 4

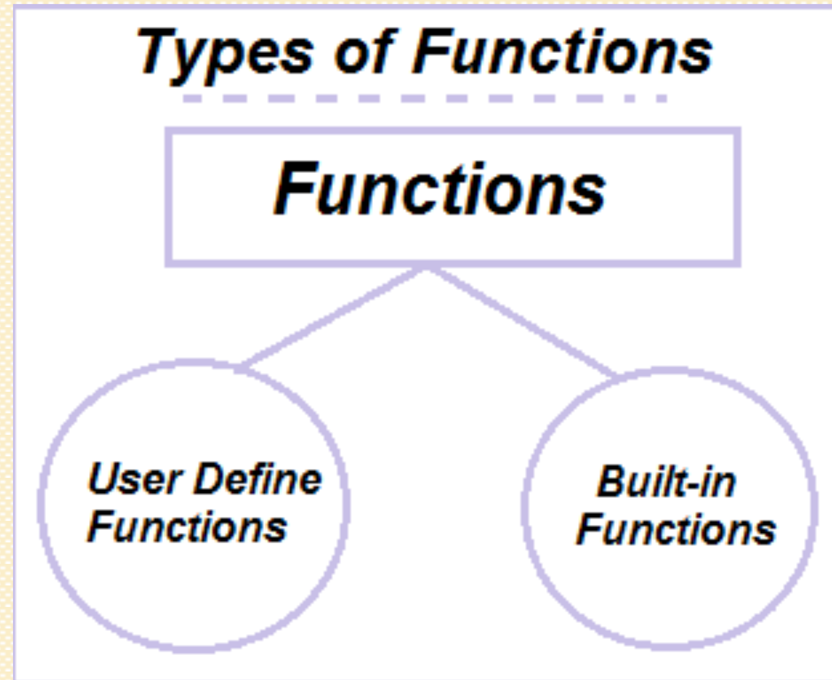
```
void function4()
{
   statement;
   statement;
   statement;
}
```

"Just keep coding.
We can always fix it later."

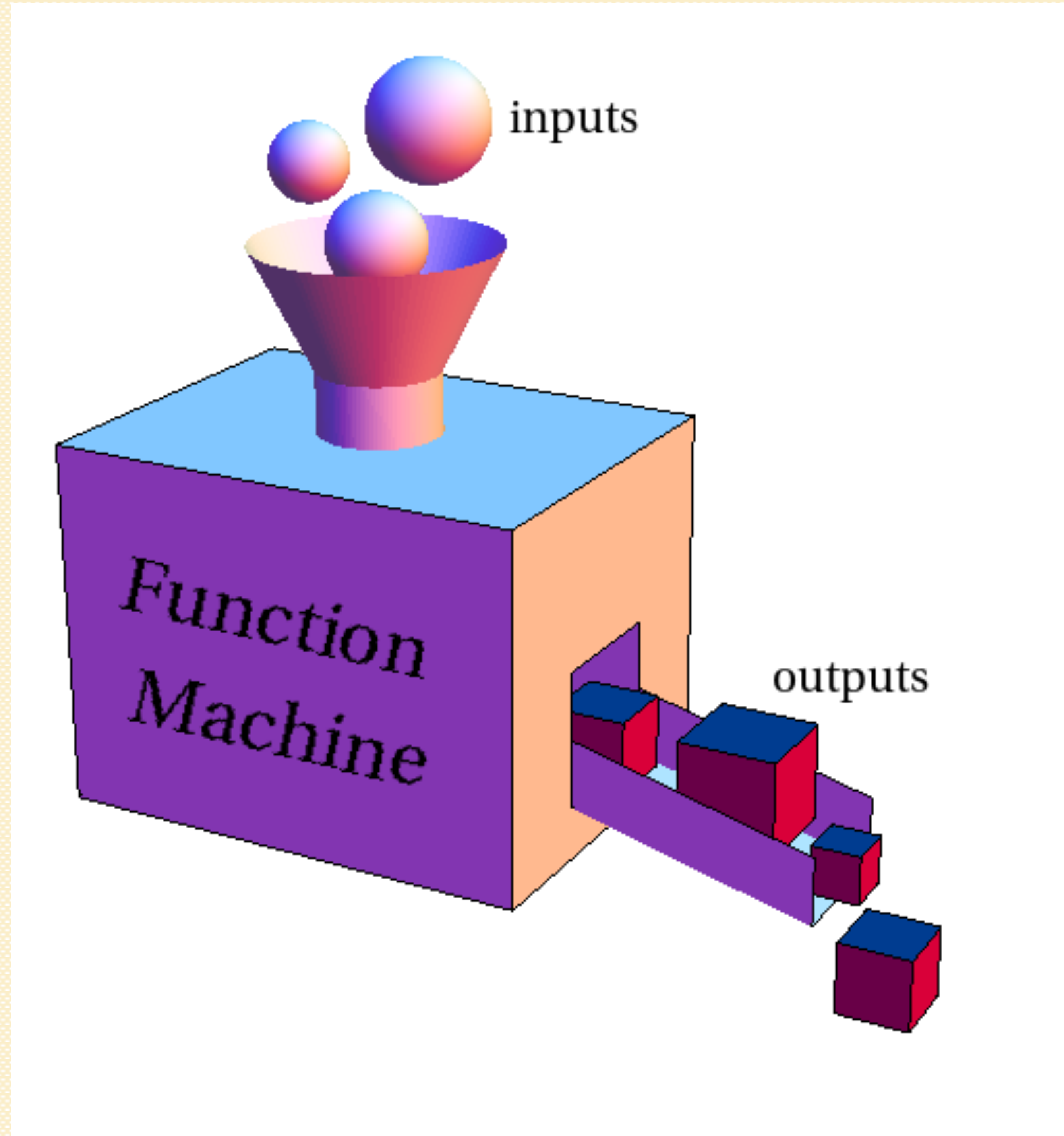
Function Concept



Output



Function Concept



Function Definition

- Definition includes

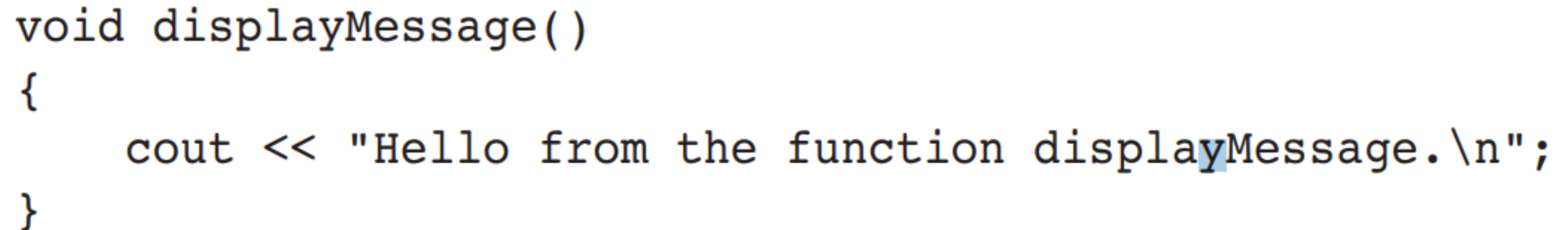
  name: name of the function. Function names follow same rules as variable names

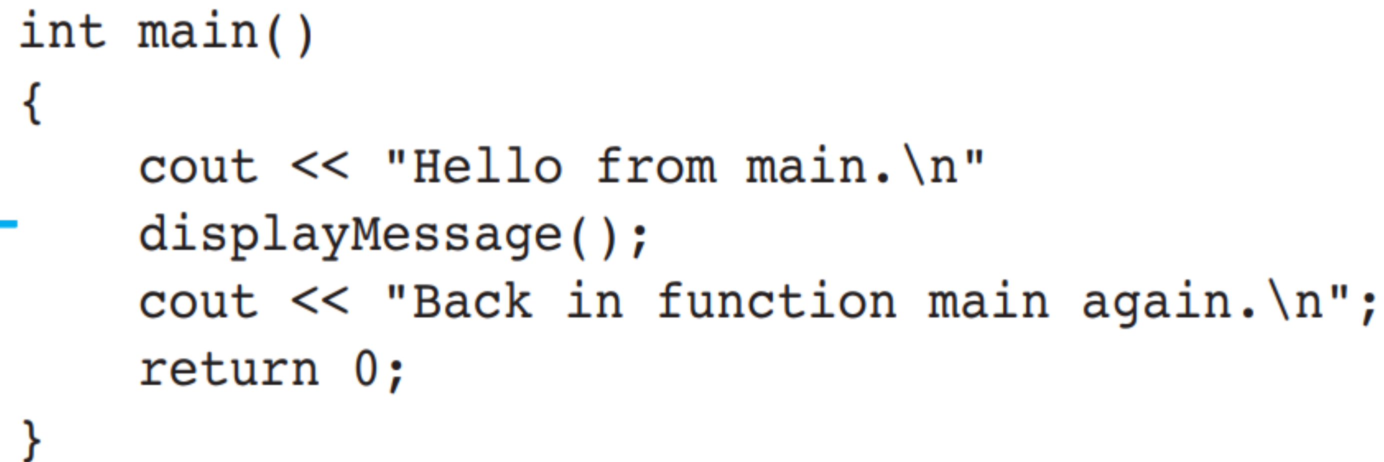
  parameter list: variables that hold the values passed to the function

  body: statements that perform the function's task

  return type: data type of the value the function returns to the part of the program that called it

Define and Call Function in C++





Define and Call Function with parameters

```
2   #include <iostream>
3   using namespace std;
4
5   int sum(int x, int y){
6       return x+y;
7   }
8
9   int main() {
10
11  cout<<sum(3,6);
12   return 0;
13  }
```

Function Type

Function parameters

Call function

Calling a Function

- `main` is automatically called when the program starts
- `main` can call any number of functions
- Functions can call other functions

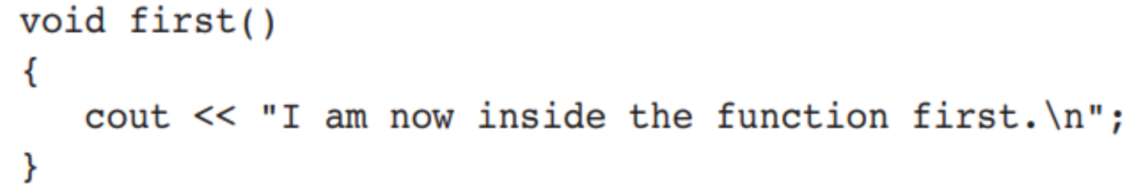
Function called inside loop

```
1   #include <iostream>
2   using namespace std;
3
4   // This function displays a greeting.
5   void displayMessage(){
6   cout << "Hello from the function displayMessage.\n";
7   }
8
9   int main(){
10  cout << "Hello from main.\n";
11
12  for (int count = 0; count < 3; count++){
13      displayMessage(); // Call displayMessage
14  }
15
16  cout << "Back in function main again.\n";
17  return 0;
18  }
```

Output

```
Hello from main.
Hello from the function displayMessage.
Hello from the function displayMessage.
Hello from the function displayMessage.
Back in function main again.
```

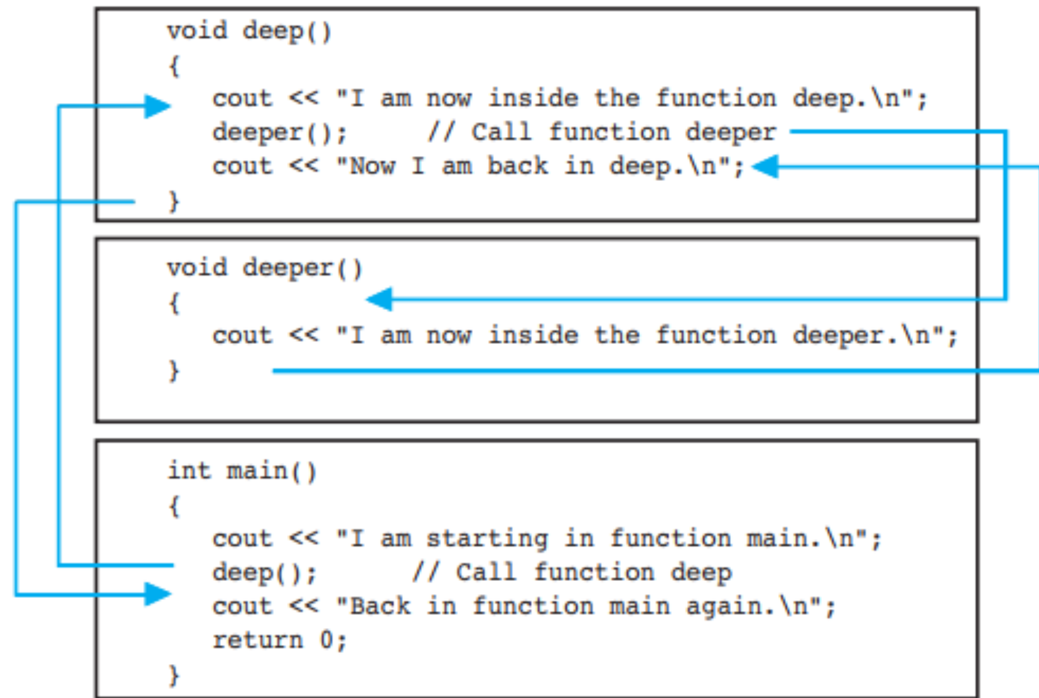
Function Procedure



```
void first()
{
   cout << "I am now inside the function first.\n";
}

void second()
{
   cout << "I am now inside the function second.\n";
}

int main()
{
   cout << "I am starting in function main.\n"
   first();
   second();
   cout << "Back in function main again.\n";
   return 0;
}
```

Functions inside each other

```
1   #include <iostream>
2   using namespace std;
3
4   /** deeper * * This function displays a message. **/
5    void deeper(){
6    cout << "I am now inside the function deeper.\n";
7    }
8
9    /* * deep * * This function displays a message. **/
10  void deep(){
11       cout << "I am now inside the function deep.\n";
12       deeper(); // Call function deeper
13       cout << "Now I am back in deep.\n";
14   }
15
16  int main(){
17       cout << "I am starting in function main.\n";
18       deep(); // Call function deep
19       cout << "Back in function main again.\n";
20       return 0;
21   }
```



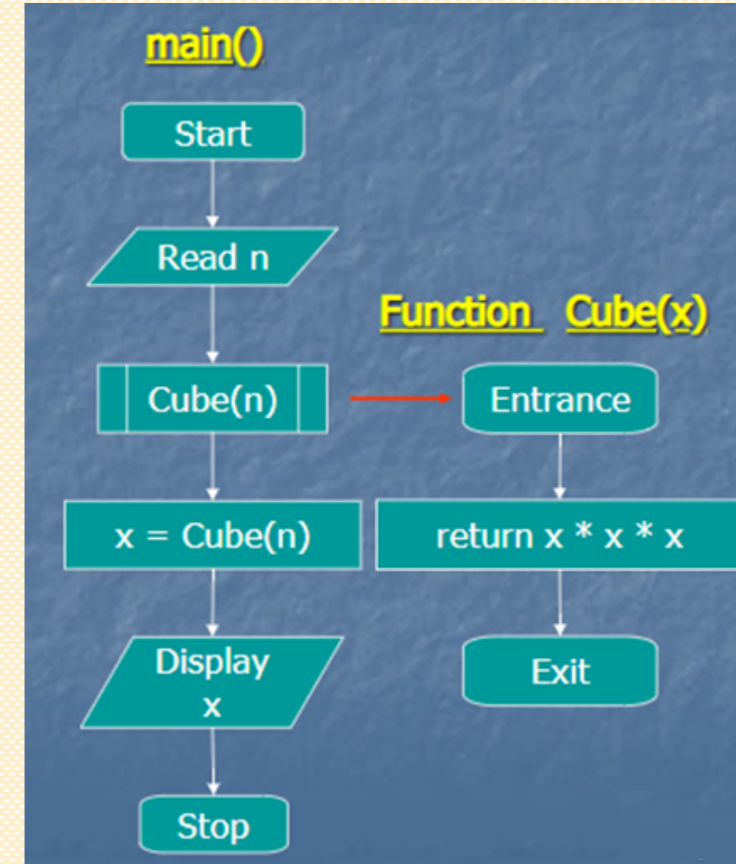
Output

```
I am starting in function main.
I am now inside the function deep.
I am now inside the function deeper.
Now I am back in deep.
Back in function main again.
```

Cube Function, using Parameters

```
1   #include <iostream>
2   using namespace std;
3
4   double Cube(double a) {
5       return a*a*a;
6   }
7
8   int main(){
9       double v, a;
10      cout << "Give the cube side ";
11      cin >> a;
12      v = Cube(a);
13      cout << " The Volume is " << v;
14      cout << endl;
15      return 0;
16  }
```

Parameter



Simple Sum Function

```
2   #include <iostream>
3   using namespace std;
4
5   int sum(int x, int y){
6       return x+y;
7   }
8
9   int main() {
10
11  cout<<sum(3,6);
12   return 0;
13  }
```

Sum Function

- Write a function "Sum" that will take two integer numbers n and m and return the sum of all the numbers from n to m.

i.e.

- Sum(1,4) = 1 + 2 + 3 + 4 = 10
- Sum(4,9) = 4 + 5 + 6 + 7 + 8 + 9 = 39
- Sum(7,7) = 7
- Sum(7,2) = 0

Sum Function

```
1     #include <iostream>
2     using namespace std;
3
4     int Sum(int n, int m){
5         int s=0;
6         for (int i = n; i <= m; i++){
7             s = s + i;
8         }
9         return s;
10    }
11
12    int main(){
13        int x,y,z;
14        cout << "Input First number";
15        cin >>x;
16        cout << "Input Second number";
17        cin >>y;
18        z= Sum(x, y);
19        cout << " The sum is " << z;
20        return 0;
21    }
```

Factorial Calculation

- A classic example of recursion is calculating the factorial of a number.
- The factorial of a non-negative integer n is denoted as n! and is defined as the product of all positive integers less than or equal to n.
- For example, 5! (read as "5 factorial") is calculated as $5 \times 4 \times 3 \times 2 \times 1 = 120$.

Factorial Calculation

- A classic example of recursion is calculating the factorial of a number.
- The factorial of a non-negative integer n is denoted as n! and is defined as the product of all positive integers less than or equal to n.
- For example, 5! (read as "5 factorial") is calculated as 5 × 4 × 3 × 2 × 1 = 120.

```
1  #include <iostream>
2  using namespace std;
3
4  int myFact(int a){
5      int fact = 1;
6      for(int i=1;i<=a;i++){
7          fact=fact*i;
8      }
9      return fact;
10 }
11 int main() {
12     int x=5;
13     cout<<x<<"!= "<<myFact(x);
14     return 0;
15 }
```

Function Prototype

- Program must include either prototype or full function definition before any call to the function, otherwise a compiler error occurs
- When using prototypes, function definitions can be placed in any order in the source file.
- **function prototype** is similar to the heading of the function except in the parameters list we show types only and we put at the end of the prototype.
- By having a function prototype, the compiler can ensure that the function is called correctly, with the appropriate parameters and types, and that the function returns a value of the correct type.
- In this course we will define the functions before calling them, so we do not need prototype.

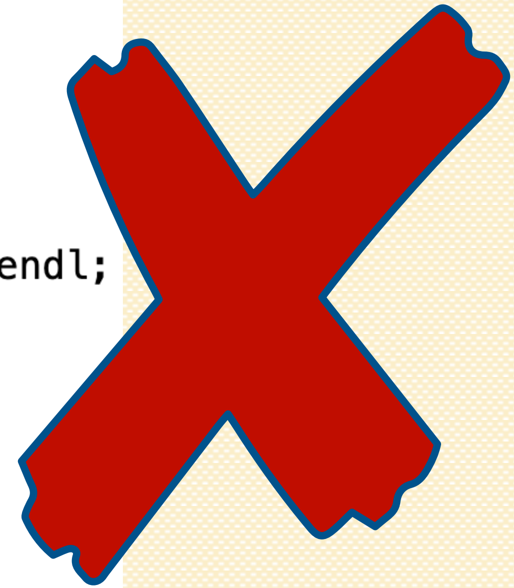
Function Prototype

```
1   #include <iostream>
2   using namespace std;
3
4   // function prototype
5   int add(int x, int y);
6
7   int main() {
8       int a = 10, b = 20;
9       int z = add(a, b);
10      cout << "The sum of " << a << " and " << b << " is " << z << endl;
11      return 0;
12  }
13
14  // function definition
15  int add(int x, int y) {
16      return x + y;
17  }
```

Function without Prototype

The function (add) is coming after main function and it hasn't defined as a function prototype

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       int a = 10, b = 20;
6       int z = add(a,b);
7       cout << "The sum of " << a << " and " << b << " is " << z << endl;
8       return 0;
9   }
10
11  // function definition without prototype
12  int add(int x, int y) {
13      return x + y;
14  }
```



Function without Prototype

The function (add) is coming before main function and it is correct and doesn't need a function prototype

```
1   #include <iostream>
2   using namespace std;
3
4   // function definition without prototype
5   int add(int x, int y) {
6       return x + y;
7   }
8
9   int main() {
10      int a = 10, b = 20;
11      int z = add(a,b);
12      cout << "The sum of " << a << " and " << b << " is " << z << endl;
13      return 0;
14  }
```

Pass by Value

- Pass by value: when argument is passed to a function, a copy of its value is placed in the parameter
- Function cannot access the original argument
- Changes to the parameter in the function do not affect the value of the argument in the calling function

Pass by Value

```
1   #include <iostream>
2   using namespace std;
3
4   void func1(double, int); // Function prototype
5
6   int main(){
7       int x = 0;
8       double y = 1.5;
9       cout << x << " " << y << endl;
10      func1(y, x);
11      cout << x << " " << y << endl;
12      return 0;
13  }
14
15  void func1(double a, int b){
16      cout << a << " " << b << endl;
17      a = 0.0;
18      b = 10;
19      cout << a << " " << b << endl;
20  }
```

Output

```
0     1.5
1.5   0
0     10
0     1.5
```

The `return` Statement

- Used to end execution of a function
- Can be placed anywhere in a function
  - *Any statements that follow the* `return` *statement will not be executed*
- Can be used to prevent abnormal termination of program
- Without a `return` statement, the function ends at its last `}`

Functions that don't return a value use **void** as the return type.

Returning a Value From a Function

- `return` statement can be used to return a value from the function to the module that made the function call
- Prototype and definition must indicate data type of return value (not `void`)
- Calling function should use return value, *e.g.*,
  - *assign it to a variable*
  - *send it to* `cout`
  - *use it in an arithmetic computation*
  - *use it in a relational expression*

Returning a Value From a Function

- Calling function should use return value, *e.g.*,
  - *assign it to a variable*
  - *send it to* `cout`
  - *use it in an arithmetic computation*
  - *use it in a relational expression*

```
#include <iostream>
using namespace std;

int square(int x) {
    return x * x;
}

int main() {
    int number = 5;
    int squaredValue = square(number);
    cout << "The square of " << number << " is: " << squaredValue << endl;

    return 0;
}
```

Returning a Value From a Function

- Calling function should use return value, *e.g.*,
  - *assign it to a variable*
  - *send it to* **`cout`**
  - *use it in an arithmetic computation*
  - *use it in a relational expression*

```
#include <iostream>
using namespace std;
int square(int x) {
    return x * x;
}

int main() {
    int number = 5;
    cout << "The square of " << number << " is: " << square(number) << endl;

    return 0;
}
```

Returning a Value From a Function

- Calling function should use return value, *e.g.*,
  - *assign it to a variable*
  - *send it to `cout`*
  - *use it in an arithmetic computation*
  - *use it in a relational expression*

```
#include <iostream>
using namespace std;

int square(int x) {
    return x * x;
}

int main() {
    int number = 5;
    int squaredValue = square(number);
    int multipliedValue = squaredValue * 2;
    cout << "The square of " << number << " and * by 2 is: " << multipliedValue << endl;

    return 0;
}
```

Returning a Value From a Function

- Calling function should use return value, *e.g.*,
  - *assign it to a variable*
  - *send it to* **`cout`**
  - *use it in an arithmetic computation*
  - *use it in a relational expression*

```
#include <iostream>
using namespace std;

int square(int x) {
    return x * x;
}

int main() {
    int number = 5;
    int squaredValue = square(number);
    if (squaredValue > 20) {
        cout << "The square of " << number << " is greater than 20." << endl;
    } else {
        cout << "The square of " << number << " is not greater than 20." << endl;
    }

    return 0;
}
```

Returning a Value – the `return` Statement

- Format: `return expression;`
- ***`expression`*** may be a variable, a literal value, or an expression.
- ***`expression`*** should be of the same data type as the declared return type of the function (will be converted if not)

Using return

```
int max(int x, int y){
    int largest;
   if (x>y){
    largest = x;
   } else {
    largest = y;
   }
   return largest;
}
```

```
int max(int x, int y){
    if (x>y){
     return x;
    } else {
     return y;
    }
}
```

Same result

A function with Boolean `return`

```
bool isValid(int val){
    int min = 0, max = 100;
    bool status;
    if(val >= min && val <= max){
        status = true;
    } else {
        status = false;
    }
    return status;

}
```

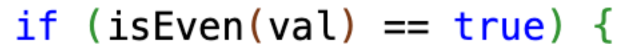
```
bool isValid(int val){
    if(val >= 0 && val <= 100){
        return true;
    } else {
        return false;
    }

}
```

Same result

Function of Boolean type

```
1   #include <iostream>
2   using namespace std;
3
4   /** isEven *
5   This Boolean function tests if the integer argument
6   receives is even or odd. It returns true if the
7   argument is even and false if it is odd. */
8   bool isEven(int number){
9       if (number % 2 == 0){
10          return true; // The number is even if there's no remainder
11      } else {
12          return false; // Otherwise, the number is odd
13      }
14  }
15
16  int main(){
17      int val; // Holds the value to be tested
18      cout << "Enter an integer and I will tell you ";
19      cout << "if it is even or odd: ";
20      cin >> val;
21
22      // Indicate whether it is even or odd
23      if (isEven(val)) {
24          cout << val << " is even.\n";
25      } else {
26          cout << val << " is odd.\n";
27      }
28      return 0;
29  }
```

It can be written like this too



```
if (isEven(val) == true) {
```

All the same

```
bool isEven(int number){
    if (number % 2 == 0){
        return true;
    } else {
        return false;
    }
}
```

```
bool isEven(int number){
    bool answer;
    if (number % 2 == 0){
        answer = true;
    }
    return answer;
}
```

```
bool isEven(int number){
    bool answer = false;
    if (number % 2 == 0){
        answer = true;
    }
    return answer;
}
```

Recursion

- Recursion is when a function calls itself.
- It's useful for solving problems that can be divided into smaller, **similar** sub-problems.
- Having the right initial conditions is vital to avoid never-ending loops in your code.
- Recursion must have a **base case**, a condition that determines when the recursion should stop. *Without a base case, recursion would lead to **infinite** calls.*

Recursion - Factorial Calculation

- If n is 0, the factorial is 1 (0! = 1). This is the simplest case.
- If n is greater than 0, the factorial of n can be calculated as n times the factorial of (n - 1).
- Recursion is used in various algorithms and data structures, such as tree traversal, searching, and sorting.
- it simplifies the solution for problems that have a self-similar structure.

Recursion - Factorial Calculation

```
1  #include <iostream>
2  using namespace std;
3
4  int myFact(int n) {
5      if (n == 0) {
6          return 1;
7      } else {
8          return n * myFact(n - 1);
9      }
10 }
11 int main() {
12     int x=5;
13     cout<<x<<"!= "<<myFact(x);
14     return 0;
15 }
```

Recursion - Sum Function

- Write a function “Sum” that will take two integer numbers n and m and return the sum of all the numbers from n to m.

i.e.

- Sum(1,4) = 1 + 2 + 3 + 4 = 10
- Sum(4,9) = 4 + 5 + 6 + 7 + 8 + 9 = 39
- Sum(7,7) = 7
- Sum(7,2) = 0

Recursion - Sum Function

```
1  #include <iostream>
2  using namespace std;
3  int Sum(int n) {
4      if (n == 0) {
5          return 0;
6      } else {
7          return n + Sum(n - 1);
8      }
9  }
10
11 int main() {
12     int x = 5;
13     int result = Sum(x);
14     cout << "Sum of numbers from 0 to " << x << " is: " << result << endl;
15     return 0;
16 }
```

Recursion - Fibonacci

- Fibonacci (fee·buh·naa·chee): The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones. The sequence typically starts with 0 and 1.
- The sequence typically starts with the numbers 0 and 1, and it continues infinitely in both directions.
- Here are the first few numbers in the Fibonacci sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Recursion - Fibonacci

```
1  #include <iostream>
2  using namespace std;
3
4  int Fibonacci(int n) {
5      if (n == 0) {
6          return 0;
7      } else if (n == 1)  {
8          return 1;
9      } else {
10        return Fibonacci(n-1)+Fibonacci(n-2);
11     }
12 }
13
14 int main() {
15     int x = 8;
16     int result = Fibonacci(x);
17     cout<< "The Fibonacci number at position " << x << " is: " << result << endl;
18     return 0;
19 }
```