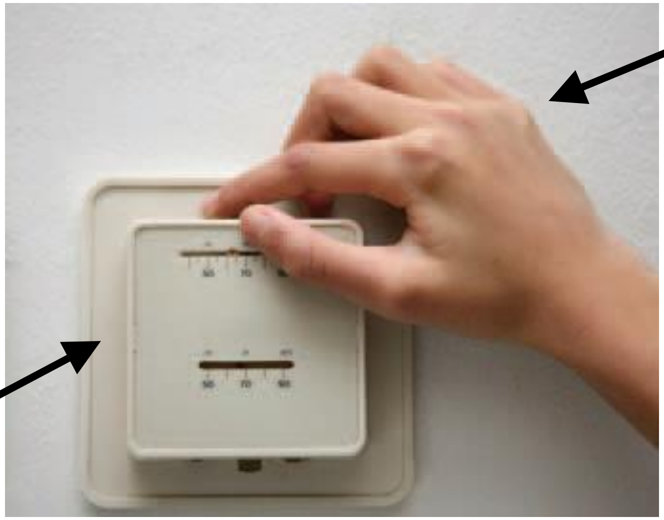
Chapter 04: Functions

Call a Function to Get Something Done

Calling a Function

A programmer *calls* a function
to have its instructions executed.

the caller
(getting things done)



the function
(has the *modify temperature* instructions)

Calling a Function

```
int main()
{
   double z = pow(2, 3);
   ...
}
```

By using the expression: **`pow(2, 3)`**
`main` *calls* the **`pow`** function, asking it to compute 2^3.

The **`main`** function is temporarily suspended.

The instructions of the **`pow`** function execute and compute the result.

The pow function *returns* its result back to **`main`**, and the **`main`** function resumes execution.

The Return Statement Does Not Display (Good!)

```
int main()
{
    double z = pow(2, 3);

    // display result of calculation
    // stored in variable z
    cout << z << endl;


    return 0;
}
```

Implementing Functions

When writing this function, you need to:

- Pick a good, descriptive name for the function

- Give a type and a name for each parameter.
  There will be one parameter for each piece
  of information the function needs to do its job.

- Specify the type of the return type

```
double cube_volume(double side_length)
```

Implementing Functions

The code the function names must be in a block:

```
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}
```

Implementing Functions

The parameter allows the caller to give the function information it needs to do it’s calculating.

```
double cube_volume(double side_length)
{
    double volume = side_length * side_length * side_length;
    return volume;
}
```

Implementing Functions

The **`return`** statement gives the function's result to the caller.

```
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}
```

A Complete Testing Program

cube.cpp

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

/**
   Computes the volume of a cube.
   @param side_length the side length of the cube
   @return the volume
*/
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}
```

A Complete Testing Program

ch05/cube.cpp

```
int main()
{
   double result1 = cube_volume(2);
   double result2 = cube_volume(10);
   cout << "A cube with side length 2 has volume "
      << result1 << endl;
   cout << "A cube with side length 10 has volume "
      << result2 << endl;

   return 0;
}
```

Implementing Functions

Syntax 5.1 Function Definition

Type of return value

Name of function

Type of parameter variable

Name of parameter variable

```
double cube_volume(double side_length)
{
    double volume = side_length * side_length * side_length;
    return volume;
}
```

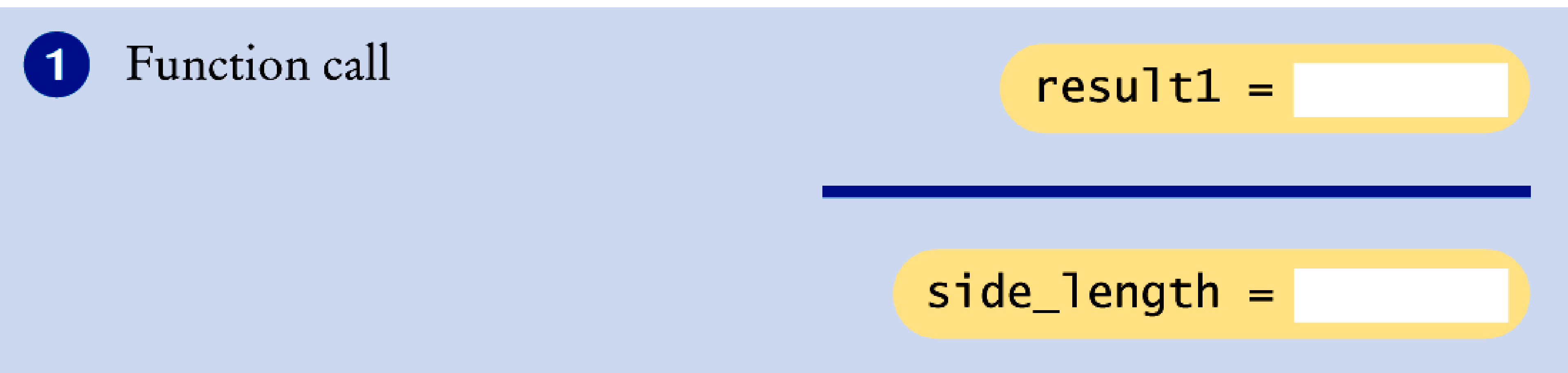
Function body, executed when function is called.

return statement exits function and returns result.

Parameter Passing

1. In the calling function, the local variable **`result1`** already exists. When the **`cube_volume`** function is called, the parameter variable **`side_length`** is created.

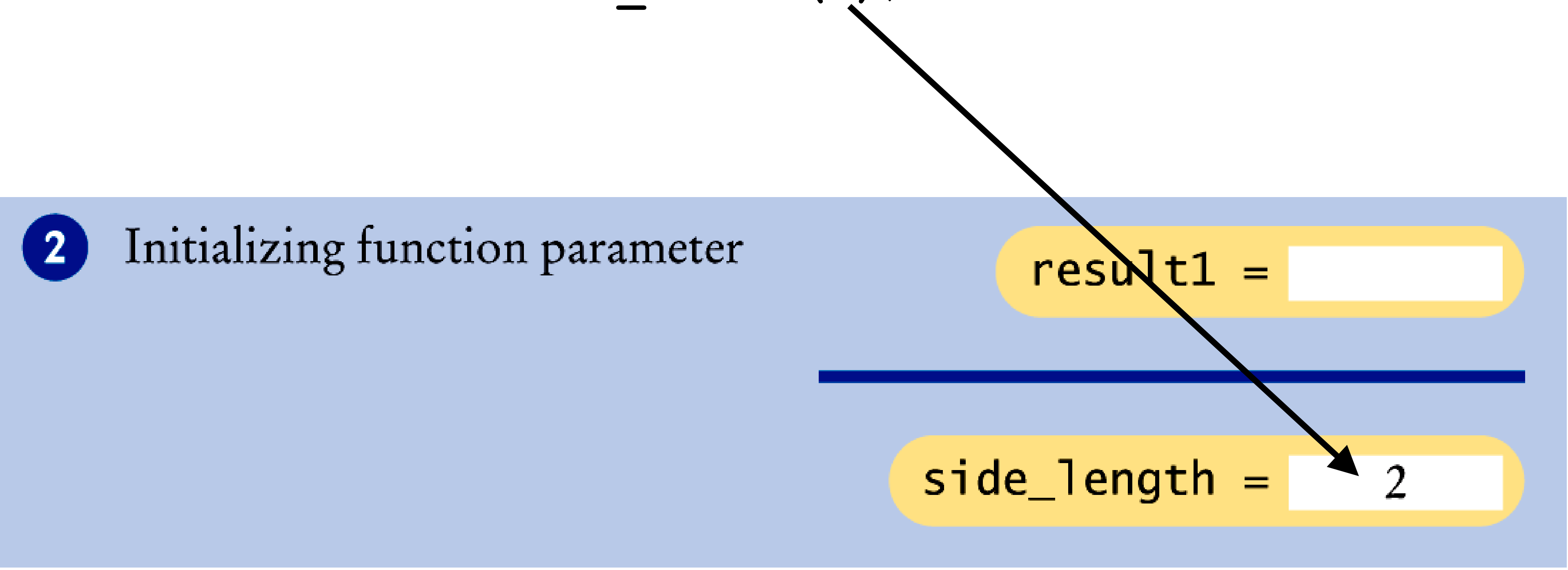
```
double result1 = cube_volume(2);
```



Parameter Passing

2. The parameter variable is initialized with the value that was passed in the call. In our case, **`side_length`** is set to 2.

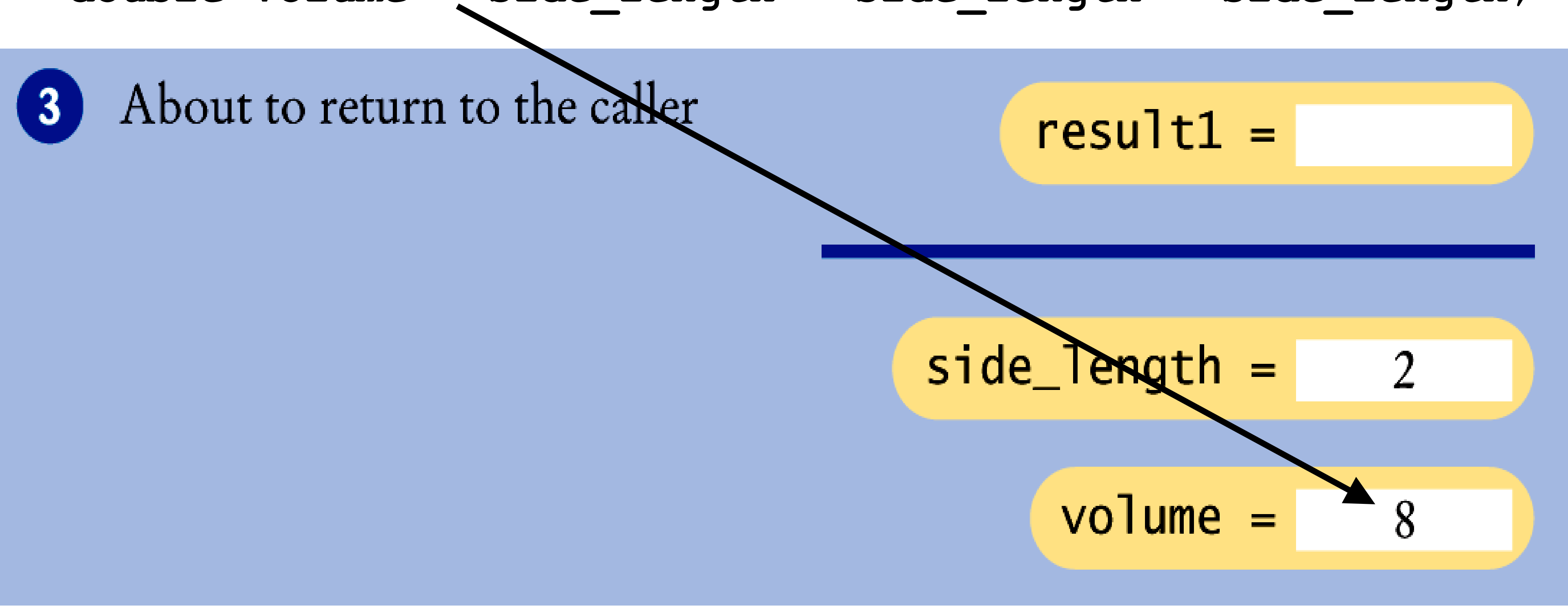
```
double result1 = cube_volume(2);
```



Parameter Passing

3. The function computes the expression **`side_length * side_length * side_length`**, which has the value 8. That value is stored in the local variable **`volume`**.

```
[inside the function]
double volume = side_length * side_length * side_length;
```



Parameter Passing

4. The function returns. All of its variables are removed.
   The return value is transferred to the caller, that is, the function calling the **`cube_volume`** function.

```
double result1 = cube_volume(2);
```

The function executed: **`return volume;`**
which gives the caller the value **`8`**

Parameter Passing

4. The function returns. All of its variables are removed. The return value is transferred to the caller, that is, the function calling the **`cube_volume`** function.

```
double result1 = cube_volume(2);
```

the returned 8 is about to be stored

The function is over.
`side_length` and **`volume`** are gone.

Parameter Passing

The caller stores this value in their local variable **`result1`**.

```
double result1 = cube_volume(2);
```

Return Values

The `return` statement yields the function result.

Return Values

This behavior can be used to handle unusual cases.

What should we do if the side length is negative?
We choose to return a zero and not do any calculation:

```
double cube_volume(double side_length)
{
   if (side_length < 0) return 0;
   double volume = side_length * side_length * side_length;
   return volume;
}
```

Return Values

The **`return`** statement can return the value of any expression.

Instead of saving the return value in a variable and returning the variable, it is often possible to eliminate the variable and return a more complex expression:

```
double cube_volume(double side_length)
{
   return side_length * side_length * side_length;
}
```

Common Error – Missing Return Value

Your function always needs to return something.

Consider putting in a guard against negatives
and also trying to eliminate the local variable:

```
double cube_volume(double side_length)
{
   if (side_length >= 0)
   {
      return side_length * side_length *
       side_length; }
}
```

Functions

- In C++, a function is a group of statements that is given a name, and which can be called from some point of the program.
- The most common syntax to define a function is:

```
return_type function_name ( parameter1, parameter2, ...)
{
    statements //body of the function
}
```

Where:

- `return_type` is the type of the value returned by the function.
- `function_name` is the identifier by which the function can be called.
- `parameters` (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma. Each parameter looks very much like a regular variable declaration (for example: int x), and in fact acts within the function as a regular variable which is local to the function. The purpose of parameters is to allow passing arguments to the function from the location where it is called from.
- `statements` is the function's body. It is a block of statements surrounded by braces { } that specify what the function actually does.

```cpp
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
}
```

No matter about the number of functions that you have, a C++ program always starts by calling main.

Functions you are already familiar with...

- main()
- pow(number, 0.5);
- sqrt(number);

Last two are predefined in math.h header.

Designing Functions – Turn Repeated Code into Functions

Consider how similar the following statements are:

```
int hours;
do
{
    cout << "Enter a value between 0 and 23:";
    cin >> hours;
} while (hours < 0 || hours > 23);

int minutes;
do
{
    cout << "Enter a value between 0 and 59: ";
    cin >> minutes;
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

The values for the high end of the range are different.

```
int hours;
do
{
    cout << "Enter a value between 0 and 23:";
    cin >> hours;
} while (hours < 0 || hours > 23);

int minutes;
do
{
    cout << "Enter a value between 0 and 59: ";
    cin >> minutes;
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

The names of the variables are different.

```
int hours;
do
{
    cout << "Enter a value between 0 and 23:";
    cin >> hours;
} while (hours < 0 || hours > 23);

int minutes;
do
{
    cout << "Enter a value between 0 and 59: ";
    cin >> minutes;
} while (minutes < 0 || minutes > 59);
```

Designing Functions – Turn Repeated Code into Functions

But there is common behavior.

```
int hours;
do
{
   cout << "Enter a value between _ and __:";
   cin >> hours;
} while (hours < _ || hours > __);
int minutes;
do
{
   cout << "Enter a value between _ and __: ";
   cin >> minutes;
} while (minutes < _ || minutes > __);
```

Designing Functions – Turn Repeated Code into Functions

Move the *common behavior* into *one* function.

```
int read_int_up_to(int high)
{
    int input;
    do
    {
        cout << "Enter a value between "
            << "0 and " << high << ": ";
        cin >> input;
    } while (input < 0 || input > high);
    return input;
}
```

Designing Functions – Turn Repeated Code into Functions

Here we read one value, making sure it's within the range.

```
int read_int_up_to(int high)
{
    int input;
    do
    {
        cout << "Enter a value between "
            << "0 and " << high << ": ";
        cin >> input;
    } while (input < 0 || input > high);
    return input;
}
```

Designing Functions – Turn Repeated Code into Functions

Then we can use this function as many times as we need:

```
int hours = read_int_up_to(23);
int minutes = read_int_up_to(59);
```

Note how the code has become much easier to understand.

And we are not rewriting code

– code reuse!

Good Design – Keep Functions Short

- And you should keep your functions short.
- As a rule of thumb, a function that is so long that its will not fit on a single screen in your development environment should probably be broken up.
- Break the code into other functions

Variable Scope

You can have only *one* **`main`** function
but you can have as many variables and parameters
spread amongst as many functions as you need.

Can you have the same name in different functions?

Variable Scope

A variable or parameter that is defined within a function is visible from the point at which it is defined until the end of the block named by the function.

This area is called the *scope* of the variable.

Variable Scope

```
double cube_volume(double side_len)
{
   double volume = side_len * side_len * side_len;
   return volume;
}
int main()
{
   double volume = cube_volume(2);
   cout << volume << endl;
   return 0;
}
```

Each **`volume`** variable is defined in a separate function, so there is not a problem with this code.

Variable Scope

Names inside a block are called *local* to that block.

A function names a block.

Recall that variables and parameters do not exist after the function is over—because they are local to that block.

But there are other blocks.

Variable Scope

It is ***not legal*** to define two variables or parameters with the same name in the same scope.

For example, the following is not legal:

```
int test(double volume)
{
   double volume = cube_volume(2);
   double volume = cube_volume(10);
// ERROR: cannot define another volume variable
// ERROR: or parameter in the same scope
...
}
```

ERRORS!!!

Variable Scope – Nested Blocks

However, you can define another variable with the same name in a *nested block*.

```
double withdraw(double balance, double amount)
{
    if (...)
    {
        double amount = 10;
        ...
    }
    ...
}
```

a variable named **`amount`** local to the **`if`**'s block

– *and* a parameter variable named **`amount`**.

Variable Scope – Nested Blocks

The scope of the parameter variable **`amount`** is the entire function, *except* the nested block.

Inside the nested block, **`amount`** refers to the local variable that was defined in that block.

You should avoid this *potentially confusing situation* in the functions that you write, simply by renaming one of the variables.

Why should there be a variable with the same name in the same function?

Global Variables

Global variables are defined *outside* any block.

They are visible to every function defined after them.

Global Variables

In some cases, this is a good thing:

The **`<iostream>`** header defines these global variables:

```
cin
cout
```

This is good because there should only be one of each of these and every function who needs them should have direct access to them.

Global Variables

But in a banking program, how many functions should have direct access to a balance variable?

Global Variables

```
int balance = 10000; // A global variable

void withdraw(double amount)
{
    if (balance >= amount)
    {
        balance = balance - amount;
    }
}

int main()
{
    withdraw(1000);
    cout << balance << endl;
    return 0;
}
```

Global Variables

In the previous program there is only one function that updates the **`balance`** variable.

But there could be many, many, many functions that might need to update **`balance`** each written by any one of a huge number of programmers in a large company.

Then we would have a problem.

Global Variables

When multiple functions update global variables, the result can be *difficult* to predict.

Particularly in larger programs that are developed by multiple programmers, it is very important that the effect of each function be clear and easy to understand.

Local vs global variables

Local variables: Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions/blocks outside their own.

Global variables: Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

```cpp
#include <iostream>
using namespace std;
int globalVar1 = 10;

int add(int x, int y)
{
    int result;
    result = x + y;
    cout << "Global Varibale accessed in add method: " << globalVar1 << endl;
    // << "Main Varibale accessed in add method: " << mainVar1;
    return result;
}

int main()
{
    int mainVar1 = 25;
    cout << "Global Varibale accessed in main method: " << globalVar1 << endl;
    cout << "Main Varibale accessed in main method: " << mainVar1 << endl;
    for (int i = 0; i < 3; i++)
    {
        cout << "Print local variable inside loop: " << i << endl;
        int localVar = 15;
        cout << "Print locally declared variable inside loop: " << localVar << endl;
        cout << "Print local variable inside loop: " << i << endl;
        cout << "Print main method variable inside loop: " << mainVar1 << endl;
        cout << "Print global variable inside loop: " << globalVar1 << endl;
    }
```

```cpp
#include <iostream>
using namespace std;
int globalVar = 10;

float add(float a, float b)
{
    float re = a+b;
    /* inside this function the following variables
    can be accessed : a, b, re and globalVar */
    return re;
}
int main()
{
    int x = 3, y = 4;
    int tmp = add(x,y);
    /* inside this function the following variables
    can be accessed : x, y, tmp and globalVar */
}
```

Write down the output of the following code.

```cpp
#include <iostream>
using namespace std;
int globalVar = 10;

float add(float a, float b)
{
    cout<<"Inside add ("<<a<<","<<b<<")"<<endl;
    float re = a+b;
    a = 10;
    b = 10;
    cout<<"Inside add ("<<a<<","<<b<<")"<<endl;
    return re;
}
int main()
{
    int a = 3, b = 4;
    cout<<"Inside main ("<<a<<","<<b<<")"<<endl;
    int tmp = add(a,b);
    cout<<"Inside main ("<<a<<","<<b<<")"<<endl;
}
```

- A void function performs a task, and then control returns back to the caller -but it does not return a value.

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
  cout << "I'm a function!";
}

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

Function prototypes

- Functions cannot be called before they are declared.
- In all the previous examples of functions, the functions were always defined before the main function.
- If main were defined before the other functions, this would break the rule that functions shall be declared before being used, and thus would not compile.

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
    int a = add(3,4);
    cout<<a;
}

int add(int a, int b)
{
    return a+b;
}
```

error: 'add' was not declared in this scope

Function prototypes

- The prototype of a function can be declared without actually defining the function completely, giving just enough details to allow the types involved in a function call to be known.
- Naturally, the function shall be defined somewhere else, like later in the code.

Function prototype

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

int addition(int x, int y);
// or int addition(int, int);

int main()
{
    int z,y;
    z = addition(23,5);
    y = addition(55, 23);
    cout << "The result is: " << z << endl;
}

int addition(int x, int y)
{
    return x + y;
}
```

Function overloading

- C++ allows you to specify more than one definition for a function name which is called function overloading.
- An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

Function overloading

```
void print()
{    cout<<"function with empty arguments\n"; }
void print(int i)
{    cout<<"function with int argument\n"; }
void print(int a, int b)
{    cout<<"function with two int argument\n"; }
void print(double i)
{    cout<<"function with double argument\n"; }
void print(string s)
{    cout<<"function with string arguments\n"; }

int main()
{
    print();
    print("Test");
    print(10);
    print(10.0);
    print(10, 10);
}
```

```
function with empty arguments
function with string arguments
function with int argument
function with double argument
function with two int argument
```

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

int sum(int a, int b)
{
    return a + b;
}

double sum(double a, double b)
{
    return a + b;
}

int main()
{
    cout << sum(7, 8) << '\n';
    cout << sum(1.7, 10.5) << '\n';
    return 0;
}
```

Example

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

    double radius;
    void compute_area(double r)
    {

        radius = r;

        double area = 3.14 * radius * radius;

        cout << "Radius is: " << radius << endl;
        cout << "Area is: " << area;
    }

int main()
{

    compute_area(1.5);
    return 0;
}
```

- In C++, there are 3 access modifiers:
  - public
  - private
  - protected

Class

```cpp
#include <iostream>
using namespace std;
class Circle {
private:
    double radius;
public:
    void compute_area(double r)
    {

        radius = r;
        double area = 3.14 * radius * radius;
        cout << "Radius is: " << radius << endl;
        cout << "Area is: " << area;
    }
};
int main()
{

    Circle obj;
    obj.compute_area(1.5);

    return 0;
}
```

Constructor

```cpp
class MyClass {     // The class
  public:           // Access specifier
    MyClass() {     // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;    // Create an object of MyClass (this will call the constructor)
  return 0;
}
```

Constructor

```
class Car {        // The class
  public:          // Access specifier
    string brand;  // Attribute
    string model;  // Attribute
    int year;      // Attribute
    Car(string x, string y, int z) { // Constructor with parameters
      brand = x;
      model = y;
      year = z;
    }
};

int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

Constructor

```
class Car {        // The class
  public:          // Access specifier
    string brand;  // Attribute
    string model;  // Attribute
    int year;      // Attribute
    Car(string x, string y, int z); // Constructor declaration
};

// Constructor definition outside the class
Car::Car(string x, string y, int z) {
  brand = x;
  model = y;
  year = z;
}

int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

Outside Method

```
class MyClass {        // The class
  public:              // Access specifier
    void myMethod();   // Method/function declaration
};

// Method/function definition outside the class
void MyClass::myMethod() {
  cout << "Hello World!";
}

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

Recursive Functions (Optional)

- A recursive function is a function that calls itself
- A recursive computation solves a problem by using the solution of the same problem with simpler inputs
- For a recursion to terminate, there must be special cases for the simplest inputs

Recursive Triangle Example

```
void print_triangle(int side_length)
{
  if (side_length < 1) { return; }
  print_triangle(side_length - 1);
  for (int i = 0; i < side_length; i++)
  {
    cout << "[]";
  }
  cout << endl;
}
```

Special Case

Recursive Call

- **The function will call itself (and not output anything) until `side_length` becomes < 1**
- **It will then use the return statement and each of the previous iterations will print their results**
  - **1, 2, 3 then 4**

```
[]
[][]
[][][]
[][][][]
```

Print the triangle with side length 3.
Print a line with four [].

Resursive Calls and Returns

- The call `print_triangle(4)` calls `print_triangle(3)`.
  - The call `print_triangle(3)` calls `print_triangle(2)`.
    - The call `print_triangle(2)` calls `print_triangle(1)`.
      - The call `print_triangle(1)` calls `print_triangle(0)`.
        - The call `print_triangle(0)` returns, doing nothing.
      - The call `print_triangle(1)` prints `[]`.
    - The call `print_triangle(2)` prints `[][]`.
  - The call `print_triangle(3)` prints `[][][]`.
- The call `print_triangle(4)` prints `[][][][]`.

Recursion / recursive functions

- Recursion is the property that functions must be called by themselves.

$$n! = n * (n-1) * (n-2) * (n-3) \ldots * 1$$

- E.g. find the factorial of a number

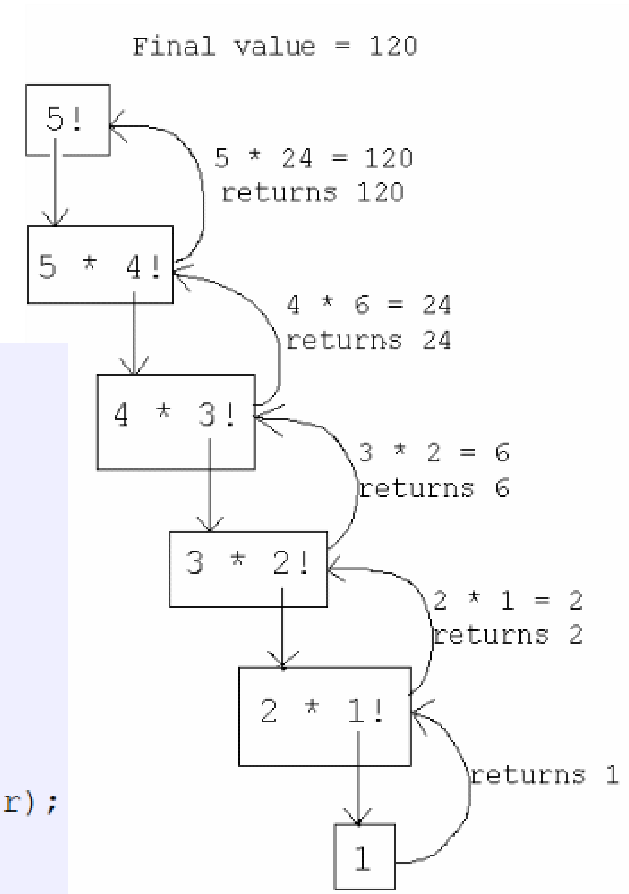
```
long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

int main ()
{
  long number = 9;
  cout << number << "! = " << factorial (number);
  return 0;
}
```

Recursion

```
long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

int main ()
{
  long number = 9;
  cout << number << "! = " << factorial (number);
  return 0;
}
```



Recursion - Example

- Calculate sum of natural numbers using recursion

```
Enter a positive integer: 6
Sum =  21
Press any key to continue . . .
```

Recursion – Example - answer

- Calculate sum of natural numbers using recursion

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

int add(int n);

int main()
{
    int n;

    cout << "Enter a positive integer: ";
    cin >> n;

    cout << "Sum =  " << add(n) << endl;

    return 0;
}

int add(int n)
{
    if (n != 0)
        return n + add(n - 1);
    return 0;
}
```

Fibonacci numbers

- Fibonacci numbers 1,1,2,3,5,8,13,…

Fibonacci relationship

$$F_1 = 1$$
$$F_2 = 1$$
$$F_3 = 1 + 1 = 2$$
$$F_4 = 2 + 1 = 3$$
$$F_5 = 3 + 2 = 5$$

In general :

$$F_n = F_{n-1} + F_{n-2}$$

or

$$F_{n+1} = F_n + F_{n-1}$$

- Using a loop?
- Using recursion?

Prime numbers

- A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.

  2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,...

Write a code to print first 10 prime numbers using

- Loop
- recursion