Programming Language-II

Lecture # 6

Lecture Content

- Functions
  - Function and it types
  - Defining a function
  - Function Declaration
  - Function calling
  - Function arguments
  - Default arguments
  - Argument passing in function
    - Pass by value
    - Pass by reference
    - Pass by pointer(will study after studying pointers)
  - Inline functions
  - Function overloading
  - Function overriding(will study in OOP)
  - Recursive functions

Functions

- Reusable entity
- A function is a group of statements that together perform a task.
- Every C++ program has at least one function, which is **main().**
- You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

Function types

- Pre defined functions
  - C++ standard library provides numerous built-in functions that your program can call. For example, function **`ceil(x)`** to find ceiling of real number and function **floor(x)** to find floor of real number and many more functions.
- User defined functions
  - Those function which developer write for own purpose.

Predefined function example

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

void main()
{
// Getting a double value
double x;
cout << "Please enter a real number: ";
cin >> x;
// Compute the ceiling and the floor of the real number
cout << "The ceil(" << x << ") = " << ceil(x) << endl;
cout << "The floor(" << x << ") = " << floor(x) << endl;
system("pause");
}
```

User-Defined C++ Functions

- Although C++ is shipped with a lot of standard functions, these functions are not enough for all users, therefore, C++ provides its users with a way to define their own functions (or user-defined function)
- For example, the <math.h> library does not include a standard function that allows users to round a real number to the i^{th} digits, therefore, we must declare and implement this function ourselves

Defining a function

- The general form of a C++ function definition is as follows

```
return_type function_name(parameter list ) {
 body of the function
}
```

Parts of function

- Return type
- Function name
- Parameter/Argument list
- Function body

Parts of function

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parts of function(Cont…)

- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Identify four parts which are discussed in previous slide,

```
int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Function Declaration

- A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.
- A function declaration has the following parts −

```
return_type function_name( parameter list );
```

Function Declaration(Cont…)

- For the above defined function max(), following is the function declaration −

  int max(int num1, int num2);

- Parameter names are not important in function declaration only their type is required, so following is also valid declaration −

  int max(int, int);

- Function declaration is required when you define a function below then where it call. In such case, you should declare the function at the top of the file calling the function.

Function Signature

- The function signature is actually similar to the function declaration the only difference is no return type is included in function signature.
- Function signature is used by compiler to perform overloaded resolution.

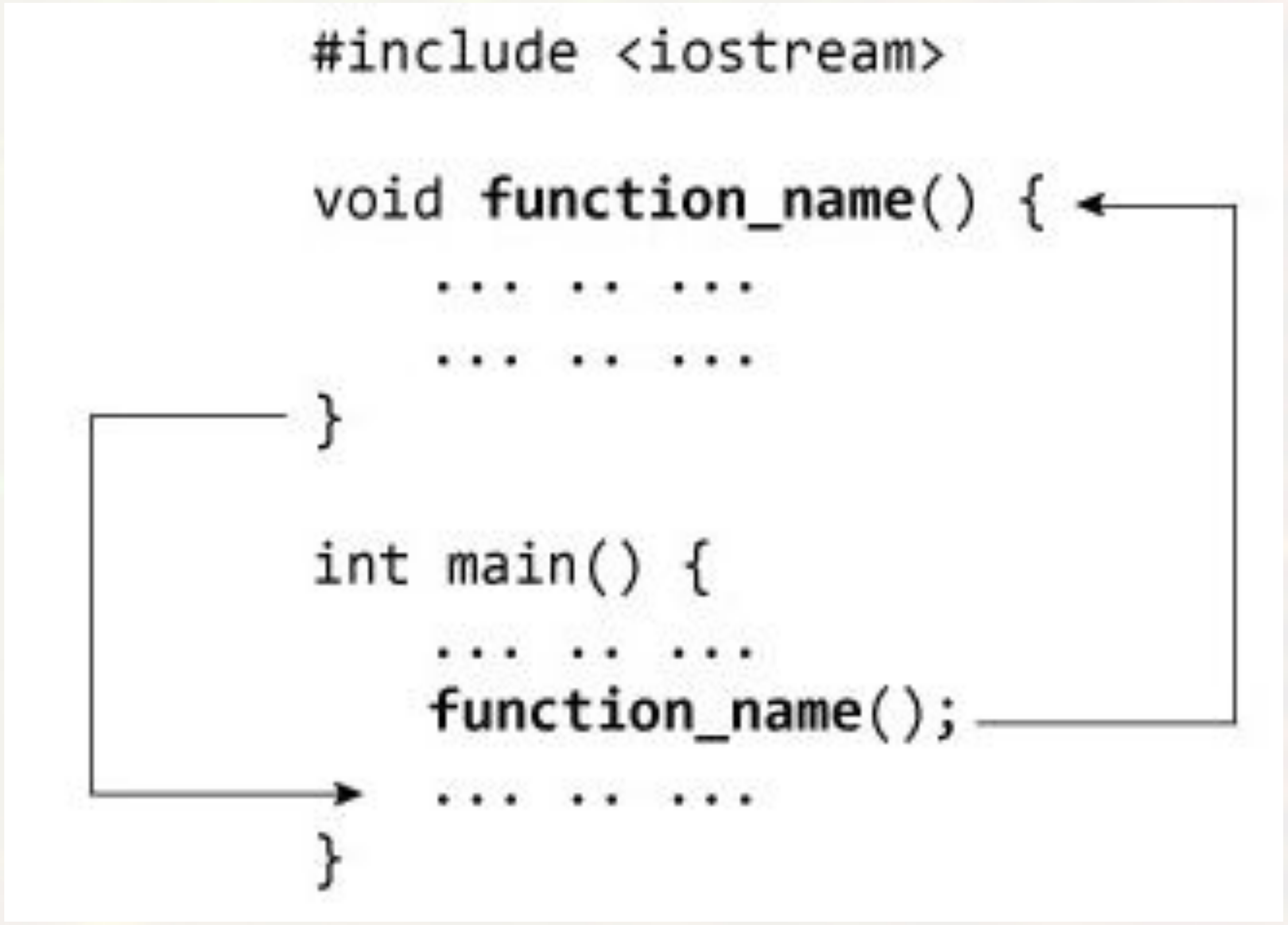
Why Do We Need Function declarations?

- For Information Hiding
  - If you want to create your own library and share it with your customers without letting them know the implementation details, you should declare all the function signatures in a header (.h) file and distribute the binary code of the implementation file
- For Function Abstraction
  - By only sharing the function signatures, we have the liberty to change the implementation details from time to time to
    - Improve function performance
    - make the customers focus on the purpose of the function, not its implementation

Function calling

- While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.
- When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it’s return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.
- To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example −

Function calling flow



```
#include <iostream>

void function_name() {
    ... .. ...
    ... .. ...
}

int main() {
    ... .. ...
    function_name();
    ... .. ...
}
```

Example

```
//Header file here
// function declaration
int max(int, int);

int main() {
int a = 100;
int b = 200;
int ret;
// calling a function to get max value.
ret = max(a, b);
cout << "Max value is : " << ret << endl;

return 0;
}
```

Example(Cont…)

```
// function definition
int max(int num1, int num2) {
// local variable declaration
int result;

if (num1 > num2)
result = num1;
else
result = num2;

return result;
}
```

Exercise

1) Write function which display your name when we call it.

Note:First identify return type and parameter list of displayName function.

2) Write function which take two arguments as an input and display sum of these.

3) Write function which take two arguments as an input and return multiplication of these.

Exercise

4) Write a C++ program that will display the calculator menu.

```
            MENU
*************************
           1. Add
           2. Subtract
           3. Multiply
           4. Divide
           5. Modulus
   Enter your choice: 1
   Enter your two numbers: 12 15
   Result: 27
```

where we take numbers from user?

Default arguments in functions

- A default argument is a value provided in function declaration that is automatically assigned by the compiler if caller of the function doesn’t provide a value for the argument with default value.
- In next slide, a simple C++ example to demonstrate use of default arguments.

Example

```
// A function with default arguments, it can be called
with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z = 0, int w = 0)
{
return (x + y + z + w);
}

int main()
{
cout << sum(10, 15) << endl;
cout << sum(10, 15, 25) << endl;
cout << sum(10, 15, 25, 30) << endl;
return 0;
}
```

Advantage and limitation of default arguments

- In previous example we don’t have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.
- Once default value is used for an argument, all subsequent arguments must have default value.
- // Invalid because z has default value, but w after it doesn't have default value

```
int sum(int x, int y, int z=0, int w)
```

Function Arguments

- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters of** the function.
- The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- While calling a function, there are two ways that arguments can be passed to a

Function Arguments

Sr. No	Call Type & Description
1	**Call by Value**This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	**Call by Reference** This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	**Call by Pointer** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Function Arguments

- By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

C++ function call by value

- The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

Example

```
void swap(int x, int y);
int main()
{
int a = 100;
int b = 200;
cout << "Before swap, value of a :" << a;
cout << "Before swap, value of b :" << b;
swap(a, b);
cout << "After swap, value of a :" << a;
cout << "After swap, value of b :" << b;
return 0;
}
```

Example explanation

- When the above code is put together in a file, compiled and executed, it produces the following result

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

- Which shows that there is no change in the values though they had been changed inside the function.

C++ function call by reference

- The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

- To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

Example

```
void swap(int &x, int &y);
int main() {
int a = 100;
int b = 200;
cout << "Before swap, value of a :" << a ;
cout << "Before swap, value of b :" << b ;
swap(a, b);
cout << "After swap, value of a :" << a ;
cout << "After swap, value of b :" << b ;
return 0;
}
```

C++ function call by pointer

- The **call by pointer** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.
- To pass the value by pointer, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer

Example

```
void swap(int *x, int *y);
int main() {
int a = 100;
int b = 200;
cout << "Before swap, value of a :" << a;
cout << "Before swap, value of b :" << b;
swap(&a, &b);
cout << "After swap, value of a :" << a;
cout << "After swap, value of b :" << b;
return 0;
}
```

Example explanation

- When the above code is put together in a file, compiled and executed, it produces the following result –

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Inline functions

- If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.
- Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

Inline functions

- To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.
- Following is an example, which makes use of inline function to return max of two numbers.

Inline functions

```
inline int Max(int x, int y) {
return (x > y) ? x : y;
}
int main() {
cout << "Max (20,10): " << Max(20, 10) << endl;
cout << "Max (0,200): " << Max(0, 200) << endl;
cout << "Max (100,1010): " << Max(100, 1010) << endl;
return 0;}
```

When the above code is compiled and executed, it produces the following result

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

Inline functions

- Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:
- If a function contains a loop. (for, while, do-while)
- If a function contains static variables.
- If a function is recursive.
- If a function contains switch or goto statement.

Inline functions provide following advantages

1) Function call overhead doesn't occur.
2) It also saves the overhead of push/pop variables on the stack when function is called.
3) It also saves overhead of a return call from a function.
4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.

Inline function disadvantages

- The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

Inline function disadvantages

- If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

Function overloading

Same name, different argument

```
void sameName();
int sameName(int);
void sameName(int,float);
void sameName(float,int);
```

Example#1

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

void display(int);
void display(float);
void display(int, float);

int main() {
int a = 5;
float b = 5.5;
display(a);
display(b);
display(a, b);
return 0;
}
```

Example#1(Cont…)

```
void display(int var) {
cout << "Integer number: " << var << endl;
}
void display(float var) {
cout << "Float number: " << var << endl;
}
void display(int var1, float var2) {
cout << "Integer number: " << var1;
cout << " and float number:" << var2;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Example#2

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

int absolute(int);
float absolute(float);

int main() {
int a = -5;
float b = 5.5;
cout << "Absolute value of " << a << " = " <<
absolute(a) << endl;
cout << "Absolute value of " << b << " = " <<
absolute(b);
return 0;
}
```

Example#2

```
int absolute(int var) {
if (var < 0)
var = -var;
return var;
}

float absolute(float var) {
if (var < 0.0)
var = -var;
return var;
}
```

<u>Output</u>

```
Absolute value of -5 = 5
Absolute value of 5.5 = 5.5
```

Recursive functions

<u>What is Recursion?</u>

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function

- **<u>What is base condition in recursion?</u>**
  - In recursive program, the solution to base case is provided and solution of bigger problem is expressed in terms of smaller problems.

Example

```
int fact(int n){
   if (n < = 1) // base case
   return 1;
   else
   return n*fact(n - 1);
}
```

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

Stack overflow problem in recursive functions

- If base case is not reached or not defined, then stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
// wrong base case (it may cause stack overflow).
if (n == 100)
return 1;

else
return n*fact(n - 1);
}
```

Explanation of previous example

- If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on stack, it will cause stack overflow error.

Advantages of recursive functions

- Recursion provides a clean and simple way to write code.
- The main advantage of recursion is that for problems like finding factorial it make the algorithm a little easier and more "elegant".

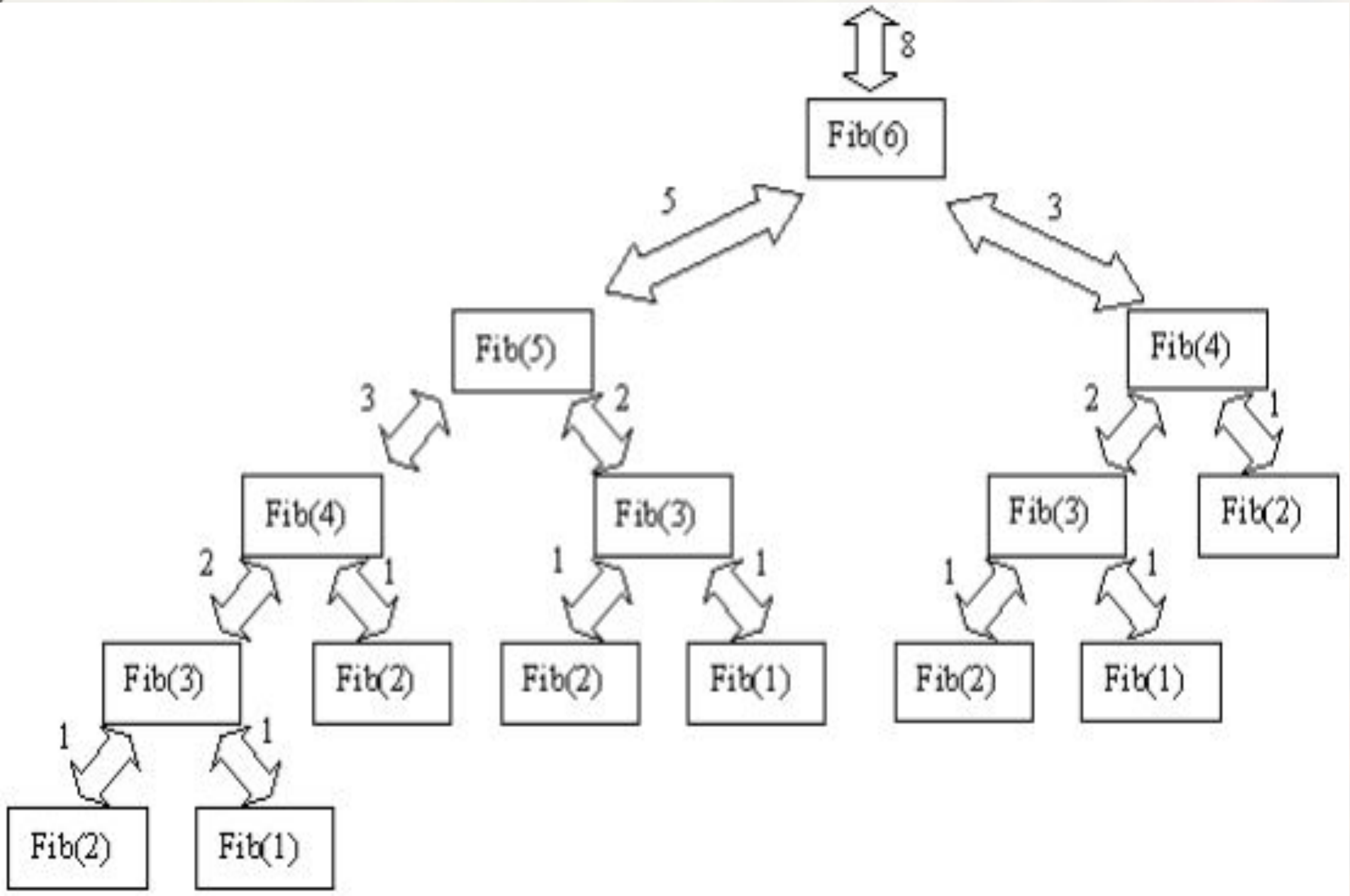
Another example of recursive function

- Fibonacci can be defined as follows:
- fibonacci of 1 or 2 is 1
- fibonacci of N (for N>2) is fibonacci of (N-1) + fibonacci of (N-2)

Iterative version	**recursive version**

// iterative version
int fib (int N) {
  int k1, k2, k3;
  k1 = k2 = k3 = 1;
  for (int j = 3; j <= N; j++) {
      k3 = k1 + k2;
      k1 = k2;
      k2 = k3;
  }
  return k3;
}
	// recursive version
	int fib (int N) {
	  if ((N == 1) || (N == 2)) return 1;
	  else return (fib (N-1) + fib (N-2));
	}

Dry run of fib function



Sum of Natural Numbers Using Recursion

```
int sum(int n);
int main(){
int number, result;
cout<<"Enter a positive integer: ";
cin>>number;
result = sum(number);
cout<< result<<endl;
}
int sum(int num){
if (num != 0)
return num + sum(num - 1); // sum() function calls
itself
else
return num;
}
```

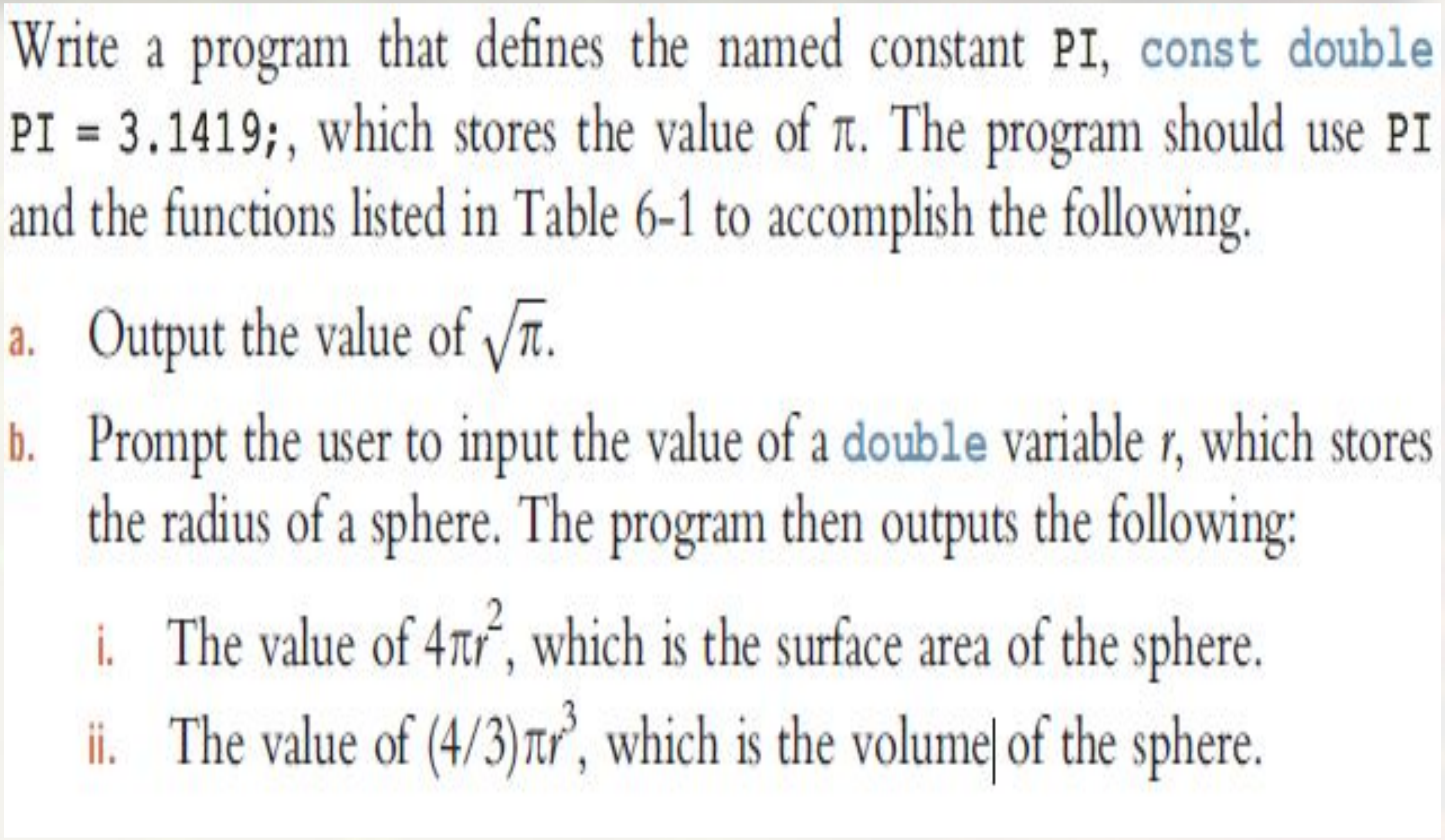
GCD of Two Numbers using Recursion

```
int hcf(int n1, int n2);
int main(){
int n1, n2;
cout<<"Enter two positive integers: ";
cin>> n1>> n2;
cout<<"G.C.D of "<<n1<<" and"<<n2<<"is"<<hcf(n1, n2);
return 0;
}
int hcf(int n1, int n2){
if (n2 != 0)
return hcf(n2, n1%n2);
else
return n1;
}
```

Exercise

1) Write a program that uses the function isNumPalindrome given in Example 6-5 (Palindrome Number). Test your program on the following numbers: 10, 34, 22, 333, 678, 67876, 44444, and 123454321.
2) Write a value-returning function, isVowel, that returns the value true if a given character is a vowel and otherwise returns false.
3) Write a program that prompts the user to input a sequence of characters and outputs the number of vowels. (Use the function isVowel written in question 2.)

Exercise



Write a program that defines the named constant `PI`, `const double PI = 3.1419;`, which stores the value of π. The program should use `PI` and the functions listed in Table 6-1 to accomplish the following.

a. Output the value of $\sqrt{\pi}$.

b. Prompt the user to input the value of a `double` variable r, which stores the radius of a sphere. The program then outputs the following:

   i. The value of $4\pi r^2$, which is the surface area of the sphere.

   ii. The value of $(4/3)\pi r^3$, which is the volume of the sphere.

Exercise

6) Write a function, reverseDigit, that takes an integer as a parameter and returns the number with its digits reversed. For example, the value of reverseDigit(12345) is 54321; the value of reverseDigit(5600) is 65; the value of reverseDigit(7008) is 8007; and the value of reverseDigit(-532) is -235.
7) During the tax season, every Friday, J&J accounting firm provides assistance to people who prepare their own tax returns. Their charges are as follows.

Exercise

a) If a person has low income (<¼ 25,000) and the consulting time is less than or equal to 30 minutes, there are no charges; otherwise, the service charges are 40% of the regular hourly rate for the time over 30 minutes.

b) For others, if the consulting time is less than or equal to 20 minutes, there are no service charges; otherwise, service charges are 70% of the regular hourly rate for the time over 20 minutes.

(For example, suppose that a person has low income and spent 1 hour and 15 minutes, and the hourly rate is $70.00. Then the billing amount is 70.00 0.40 (45 / 60) ¼ $21.00.)

Write a program that prompts the user to enter the hourly rate, the total consulting time, and whether the person has low income. The program should output the billing amount. Your program must contain a function that takes as input the hourly rate, the total consulting time, and a value indicating whether the person has low income. The function should return the billing amount. Your program may prompt the user to enter the consulting time in minutes.