

EE-103

PROGRAMMING IN C++

Programming through functional decomposition: Functions (void and value returning), parameters, scope and lifetime of variables, passing by value, passing by reference, passing arguments by constant reference; Design of functions and their interfaces (concept of functional decomposition), **recursive functions**; Function overloading and **default arguments**; Library functions; Matters of style, naming conventions, comments.

- Recursion
 - see Recursion
- GNU
 - GNU is Not UNIX

Recursion is a programming technique that allows the programmer to express operations in terms of themselves.

In C++, this takes the form of
a function that calls itself

Recursion

- Functions can call themselves! This is called recursion.
- Recursion is very useful – it's often very simple to express a complicated computation recursively.

Basic idea behind recursive algorithms:

To solve a problem,

- solve a subproblem that is a smaller instance of the same problem,
- and then use the solution to that smaller instance to solve the original problem.

Writing Recursive Functions

General form of a recursive function has the following:

```
ReturnType Functionname( Pass appropriate arguments )  
{  
    if a simple case, return the simple value  // base case / stopping condition  
    else  
        call function with simpler version of problem  
}
```

For a recursive function to stop calling itself we require some type of stopping condition. If it is not the base case, then we simplify our computation using the general formula.

Example

Computing Factorials

```
int factorial( int x ) {  
    if (x == 1)  
        return(1);  
    else  
        return(x * factorial(x-1));  
}
```

- When computing $n!$, we solved the problem of computing $n!$ (the original problem) by solving the sub-problem of computing the factorial of a smaller number, that is, computing $(n-1) \cdot (n-1)!$ (the smaller instance of the same problem), and then using the solution to the subproblem to compute the value of $n!$.
- In order for a recursive algorithm to work, the smaller sub-problems must eventually arrive at the base case. When computing $n!$, the sub-problems get smaller and smaller until we compute $0!$. Make sure that eventually, you hit the base case.

Designing Recursive Functions

- Define “Base Case”:
 - The situation in which the function does **not** call itself.
- Define “recursive step”:
 - Compute the return value with the help of the function itself.

Recursive functions

- a function calling itself (in its definition)
- a finite recursion has base case(s)
- function called with base case returns base result
- a recursive call resembles the original call but with a simpler argument
- the recursive call may result into many more recursive calls
- at each call the problem becomes simpler
- finally recursive call terminates in a base case
- when the recursive call executes, the original call waits
- from the base case, a sequence of returns traces the path all the way up till the original call

```
#include <iostream>
```

```
using namespace std;
```

```
void recurse ( int count ) // Each call gets its own count
```

```
{
```

```
    cout<< count <<"\n";
```

```
    // It is not necessary to increment count since each function's
```

```
    // variables are separate (so each count will be initialized one greater)
```

```
    recurse ( count + 1 );
```

```
}
```

```
int main()
```

```
{
```

```
    recurse ( 1 ); //First function call, so it starts at once
```

```
    system("pause");
```

```
    return 0;
```

```
}
```

Normally, a recursive function will have a variable that performs a similar action; one that controls when the function will finally exit.

The condition where the function will not call itself is termed the **base case** of the function.

Basically, it is an if-statement that checks some variable for a condition (such as a number being less than zero, or greater than some other number) and if that condition is true, it will not allow the function to call itself again.

Or, it could check if a certain condition is true and only then allow the function to call itself.

```
void CountDown(int nValue)
{
using namespace std;
cout << nValue << endl;
CountDown(nValue-1);
}
```

```
int main()
{
CountDown(10);
return 0;
}
```

When CountDown(10) is called, the number 10 is printed, and CountDown(9) is called. CountDown(9) prints 9 and calls CountDown(8). CountDown(8) prints 8 and calls CountDown(7).

The sequence of CountDown(n) calling CountDown(n-1) is continually repeated, effectively forming the recursive equivalent of an infinite loop.

- Every function call causes data to be placed on the call stack.
- Because the `CountDown()` function never returns (it just calls `CountDown()` again), this information is never being popped off the stack!
- Consequently, at some point, the computer will run out of stack memory, stack overflow will result, and the program will crash or terminate.

Stopping a recursive function generally involves using an if statement.
Function redesigned with a termination condition:

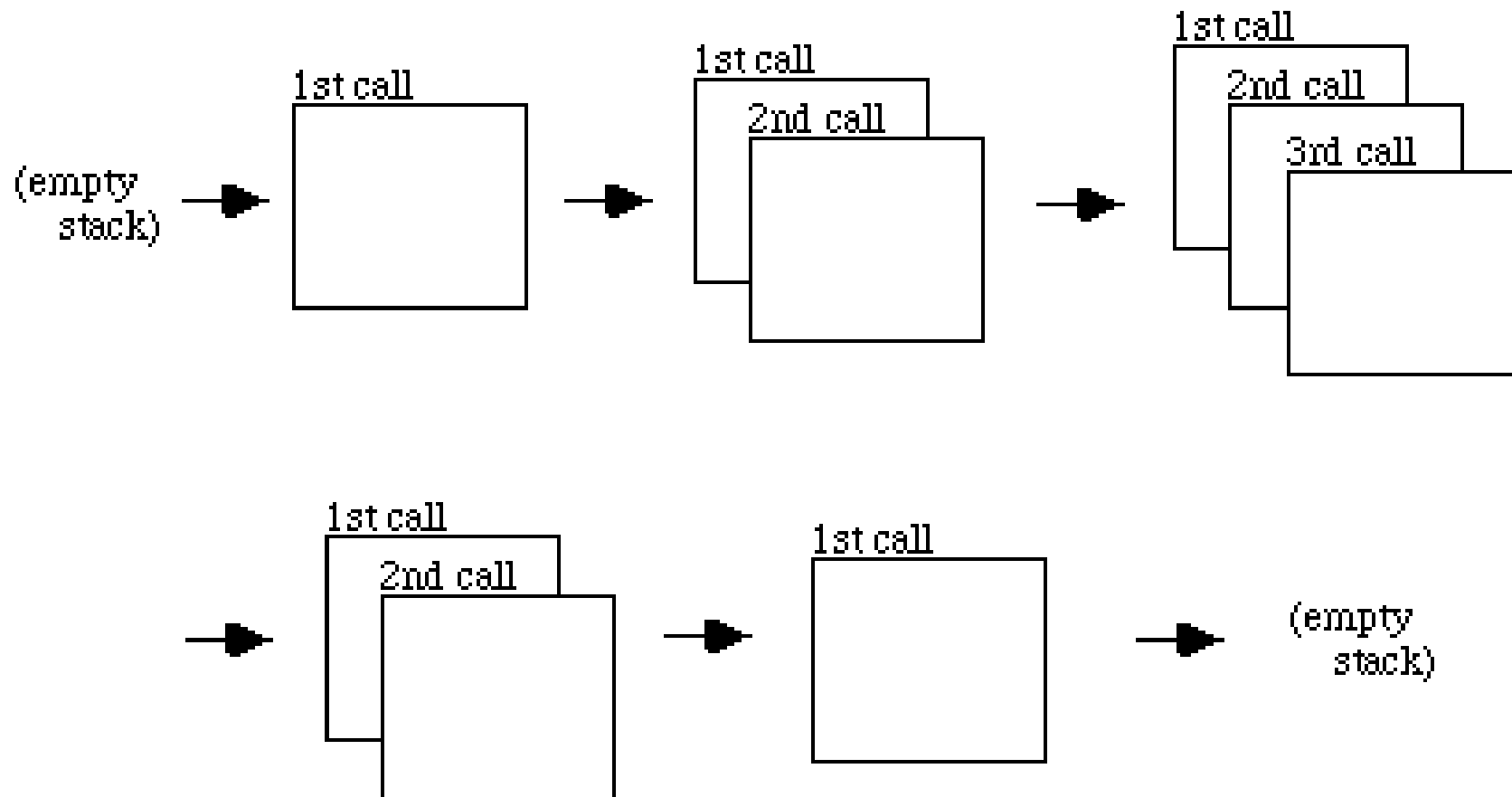
```
void CountDown(int nValue)
{
    using namespace std;
    cout << nValue << endl;

    // termination condition
    if (nValue > 0)
        CountDown(nValue-1);
}
```

```
int main(void)
{
    CountDown(10);
    return 0;
}
```

```
// return the sum of 1 to nValue  
int SumTo(int nValue)  
{  
    if (nValue <=1)  
        return nValue;  
    else  
        return SumTo(nValue - 1) + nValue;  
}
```


- SumTo(5) called, $5 \leq 1$ is false, so we return $\text{SumTo}(4) + 5$.
SumTo(4) called, $4 \leq 1$ is false, so we return $\text{SumTo}(3) + 4$.
SumTo(3) called, $3 \leq 1$ is false, so we return $\text{SumTo}(2) + 3$.
SumTo(2) called, $2 \leq 1$ is false, so we return $\text{SumTo}(1) + 2$.
SumTo(1) called, $1 \leq 1$ is true, so we return 1. This is the termination condition.
- Now we unwind the call stack (popping each function off the call stack as it returns):
SumTo(1) returns 1.
SumTo(2) returns $\text{SumTo}(1) + 2$, which is $1 + 2 = 3$.
SumTo(3) returns $\text{SumTo}(2) + 3$, which is $3 + 3 = 6$.
SumTo(4) returns $\text{SumTo}(3) + 4$, which is $6 + 4 = 10$.
SumTo(5) returns $\text{SumTo}(4) + 5$, which is $10 + 5 = 15$.
- Consequently, SumTo(5) returns 15.



- C++ arranges the memory spaces needed for each function call in a *stack*.
- The memory area for each new call is placed on the top of the stack, and then taken off again when the execution of the call is completed.

- Two approaches for writing repetitive algorithm
 - Loops
 - Recursion

- Recursion is a repetitive process in which function call itself.
- Computer language has to support for recursion

- Iterative function definition

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n == 0 \\ n*(n-1)*(n-2)*\dots*3*2*1 & \text{if } n > 0 \end{cases}$$

- Recursive function definition

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n == 0 \\ n * (\text{Factorial}(n-1)) & \text{if } n > 0 \end{cases}$$

// Factorial of a number using iterative loop

```
#include<iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
int n, result = 1, value;
```

```
cout<<"Enter a positive number: ";
```

```
cin>>value;
```

```
for ( n = 1; n <= value; n++ )
```

```
{
```

```
    result *= n;
```

```
}
```

```
cout << value<< "!= "<< result<<endl;
```

```
cin.ignore();
```

```
cin.get(); // pause
```

```
return 0;
```

```
}
```

```
// factorial calculator using recursive function
```

```
#include <iostream>
```

```
using namespace std;
```

```
int factorial (int n)
```

```
{
```

```
    if (n <= 1)
```

```
        return (1);
```

```
    else
```

```
        return (n * factorial (n-1));
```

```
}
```

```
int main ()
```

```
{
```

```
int number;
```

```
cout << "Please type a number: ";
```

```
cin >> number;
```

```
cout << number << "! = " << factorial (number);
```

```
system("pause");
```

```
return 0;
```

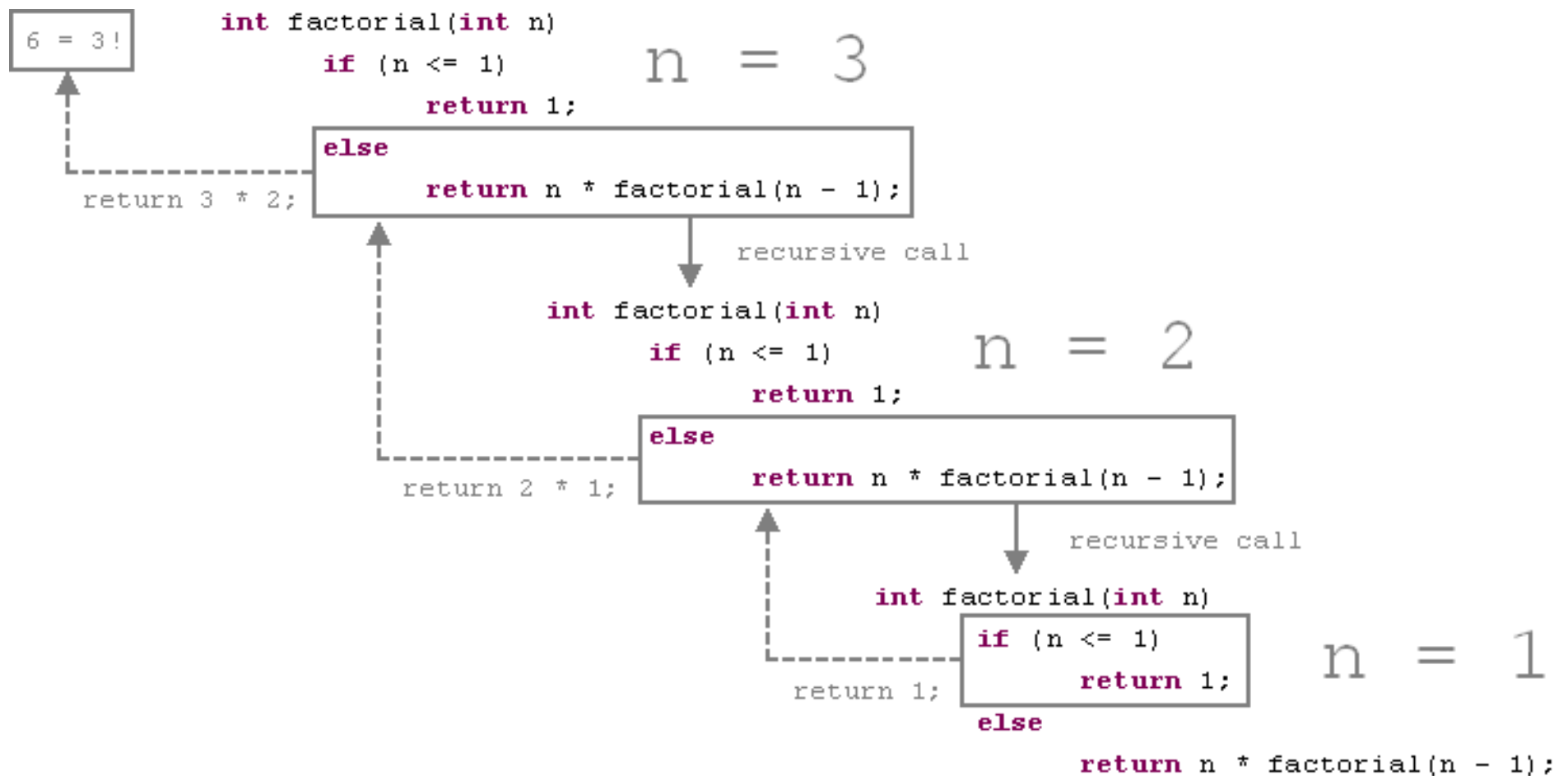
```
}
```



```

int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}

```



Memory used by a program is typically divided into four different areas:

- **The code area:** where the compiled program sits in memory.
- **The globals area:** where global variables are stored.
- **The heap:** where dynamically allocated variables are allocated from.
- **The stack:** where parameters and local variables are allocated from.

- **The stack in action**
- Because parameters and local variables essentially belong to a function, we really only need to consider what happens on the stack when we call a function.
- **Sequence of steps that takes place when a function is called:**
- The address of the instruction beyond the function call is pushed onto the stack. This is how the CPU remembers where to go after the function returns.
- Room is made on the stack for the function's return type. This is just a placeholder for now.
- The CPU jumps to the function's code.
- The current top of the stack is held in a special pointer called the stack frame. Everything added to the stack after this point is considered "local" to the function.
- All function arguments are placed on the stack.
- The instructions inside of the function begin executing.
- Local variables are pushed onto the stack as they are defined.

- When the function terminates, the following steps happen:
- The function's return value is copied into the placeholder that was put on the stack for this purpose.
- Everything after the stack frame pointer is popped off. This destroys all local variables and arguments.
- The return value is popped off the stack and is assigned as the value of the function. If the value of the function isn't assigned to anything, no assignment takes place, and the value is lost.
- The address of the next instruction to execute is popped off the stack, and the CPU resumes execution at that instruction.
- Typically, it is not important to know all the details about how the call stack works.
- However, understanding that functions are effectively pushed on the stack when they are called and popped off when they return gives you the fundamentals needed to understand recursion, as well as some other concepts that are useful when debugging.

- **The Stack**

- The Stack is an special area of memory in which temporary variables are stored. The Stack acts on the LIFO (Last In First Out) principle, which is the same principle involved in, say, the stacking of cardboard boxes one atop the other, where the topmost box, which was the last box stacked (Last In), will be the first to be removed (First Out). Thus, if the values 9,3,2,4 are stored (Pushed) on the Stack, they will be retrieved (Popped) in the order 4,2,3,9.
- In order to understand how recursive functions use the Stack, we will walk through how the second algorithm above works. For your convenience, it is reproduced below.

- ```
if(N==1) return 1
else return N*fact(N-1)
```

Let us assume we want to find the value of 3!, which is  $3 \times 2 \times 1 = 6$ . The first time the function is called, N holds the value 3, so the *else* statement is executed. The function knows the value of N, but not of fact(N-1), so it pushes N (value=3) on the stack, and calls itself for the second time with the value 2. This time round too the *else* statement is executed, and N (value=2) is pushed on the stack as the function calls itself for the third time with the value 1. Now *the* statement is executed as  $n==1$ , so the function returns 1. Since the value of fact(1) is now known, it reverts back to it's second execution by popping the last value (2) from the stack and multiplying it by 1. This operation gives the value of fact(2), so the function reverts to it's first execution by popping the next value (3) from the stack, and multiplying it with fact(2), giving the value 6, which is what the function finally returns.

- From the above example, we see that
- The function runs 3 times, out of which it calls itself 2 times. The number of times that a function calls itself is known as the *recursive depth* of that function.
- Each time the function calls itself, it stores one or more variables on the Stack. Since the Stack holds a limited amount of memory, functions with a high recursive depth may crash because of non-availability of memory. Such a condition is known as a *Stack Overflow*.
- Recursive functions usually have a *terminating condition*. In the above example the function stops calling itself when  $n==1$ . If this condition were not present, the function would keep calling itself with the values 3,2,1,0,-1,-2... and so on for infinity. This condition is known as a *Endless Recursion*.
- All recursive functions go through 2 distinct phases. The first phase, *Winding*, occurs when the function is calling itself and pushing values on the Stack. The second phase *Unwinding*, occurs when the function is popping values from the stack.

- How do you empty a vase containing five flowers?
- Answer: if the vase is not empty, you take out one flower and then you empty a vase containing four flowers.
- How do you empty a vase containing four flowers?
- Answer: if the vase is not empty, you take out one flower and then you empty a vase containing three flowers.
- How do you empty a vase containing three flowers?
- Answer: if the vase is not empty, you take out one flower and then you empty a vase containing two flowers.
- How do you empty a vase containing two flowers?
- Answer: if the vase is not empty, you take out one flower and then you empty a vase containing one flower.
- How do you empty a vase containing one flower?
- Answer: if the vase is not empty, you take out one flower and then you empty a vase containing no flowers.
- How do you empty a vase containing no flowers?
- Answer: if the vase is not empty, you take out one flower but the vase is empty so you're done.
- That's repetitive. Let's generalize it:
- How do you empty a vase containing  $N$  flowers?
- Answer: if the vase is not empty, you take out one flower and then you empty a vase containing  $N-1$  flowers.

```
void emptyVase(int flowersInVase)
{ if(flowersInVase > 0)
 { // take one flower and emptyVase(flowersInVase - 1);
 }
else { // the vase is empty, nothing to do } }
```

- Couldn't we have just done that in a for loop?
- Why yes, recursion can be replaced with iteration, but often recursion is more elegant.

# Recursion

## Drawbacks of recursion:

- has an overhead
- expensive in memory space and processor time
- each call involves processor overhead of call (changing program counter and saving return address)
- compared to this, iteration uses a repetition structure inside the function instead of calling it again and again
- still, recursive method at times gives logically more sound and more readable programs, hence it is used
- during expansion phase, each call requires another copy of function variables

# Default Parameters

## Example:

1. `int volume (int length = 1, int width = 1, int height = 1); /`  
`/ valid`
2. `int volume (int length = 1, int width, int height = 1); //`  
`not allowed`
3. `int volume (int length, int width = 1, int height = 1); //`  
`valid`

## Alternate way for prototypes

1. `int volume (int = 1, int = 1, int = 1);`
2. `int volume (int, int = 1, int = 1);`

## Some calls to the function

- (a) `volume()` //valid for declaration no.1 but not 3
- (b) `volume(5)` //valid for both, width = height = 1
- (c) `volume(5,7)` //valid for both, height = 1



- When we mention a default value for a parameter while declaring the function, it is said to be as default argument.
- In this case, even if we make a call to the function without passing any value for that parameter, the function will take the default value specified.
- By setting default argument, we are also overloading the function. Default arguments also allow us to use the same function in different situations just like function overloading.

# Rules for using Default Arguments

1. Only the last argument must be given default value. **One cannot have a default argument followed by non-default argument.**

```
sum (int x,int y);
```

```
sum (int x,int y=0);
```

```
sum (int x=0,int y); // This is Incorrect
```

2. If you default an argument, then we will have to default all the subsequent arguments after that.

```
sum (int x,int y=0);
```

```
sum (int x,int y=0,int z); // This is incorrect
```

```
sum (int x,int y=10,int z=10); // Correct
```

3. We can give any value a default value to argument, compatible with its datatype.

## Placeholder Arguments

When arguments in a function are declared without any identifier they are called placeholder arguments.

```
void sum (int,int);
```

Such arguments can also be used with default arguments.

```
void sum (int, int=0);
```

# Default values in parameters.

*// default values in functions*

```
#include <iostream>
```

```
using namespace std;
```

```
int divide (int a, int b=2)
```

```
{
```

```
int r;
```

```
r=a/b;
```

```
return (r);
```

```
}
```

```
int main ()
```

```
{
```

```
cout << divide (12);
```

```
cout << endl;
```

```
cout << divide (20,4);
```

```
system("pause");
```

```
return 0;
```

```
}
```

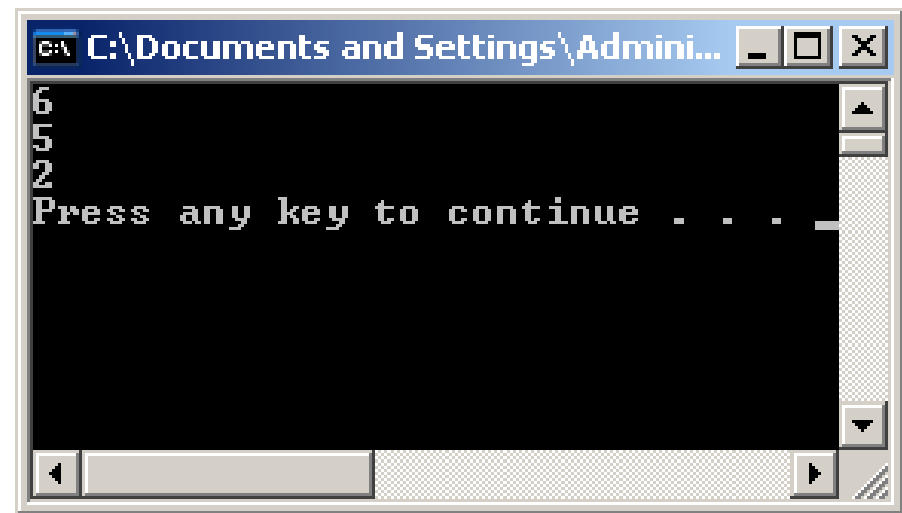
Result is: 6  
5

```
// default values in functions
#include <iostream>

using namespace std;

int divide (int a=4, int b=2)
{
 int r;
 r=a/b;
 return (r);
}

int main ()
{
 cout << divide (12);
 cout << endl;
 cout << divide (20,4);
 cout << endl;
 cout << divide ();
 cout << "\a" << endl;
 system("pause");
 return 0;
}
```



A screenshot of a Windows command prompt window. The title bar reads "C:\Documents and Settings\Admini...". The window has a black background with white text. The output of the program is visible: the number 6 on the first line, the number 5 on the second line, the number 2 on the third line, and the text "Press any key to continue . . ." on the fourth line. The cursor is positioned at the end of the fourth line.

```
// default values in functions
```

```
#include <iostream>
```

```
using namespace std;
```

```
int divide(int = 4, int = 2); // using prototype
```

```
int main ()
```

```
{
```

```
cout << divide (12);
```

```
cout << endl;
```

```
cout << divide (20,4);
```

```
cout << endl;
```

```
cout << divide ();
```

```
cout << "\a" << endl;
```

```
system("pause");
```

```
return 0;
```

```
}
```

```
int divide (int a, int b)
```

```
{
```

```
int r;
```

```
r=a/b;
```

```
return (r);
```

```
}
```

```

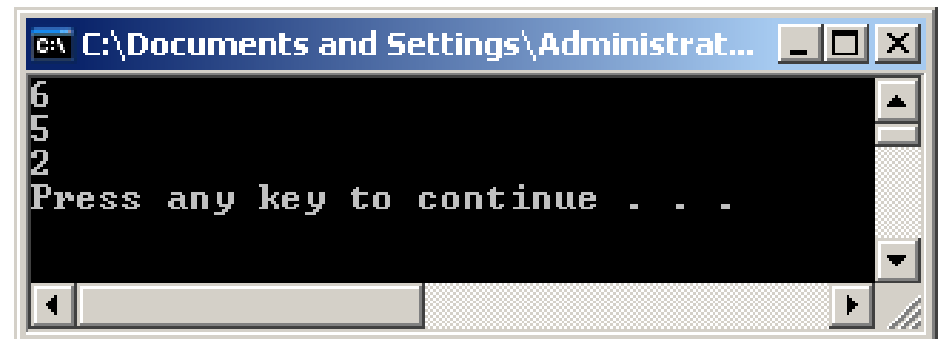
// default values in functions
#include <iostream>
using namespace std;

int divide(int = 4, int = 2);

int main ()
{
 cout << divide (12);
 cout << endl;
 cout << divide (20,4);
 cout << endl;
 cout << divide ();
 cout << "\a" << endl;
 system("pause");
 return 0;
}

int divide (int a, int b)
{
 int r;
 r=a/b;
 return (r);
}

```



```

C:\Documents and Settings\Administrat...
6
5
2
Press any key to continue . . .

```

# Default Parameters

- parameters can have default values
- default values are given in first occurrence of the function – prototype or definition
- a function can be called without giving any value for a default parameter – the default value is used
- default values can be constants, global variables or function calls
- default parameters must be in the end in parameter list of the function
- if a function has many default parameters then its call can omit rightmost parameters but not from in-between



- A default parameter (also called an optional parameter or a default argument) is a function parameter that has a default value provided to it.
- If the user does not supply a value for this parameter, the default value will be used.
- If the user does supply a value for the default parameter, the user-supplied value is used instead of the default value.
- *Rule: If the function has a forward declaration (prototype), put the default parameters there. Otherwise, put them on the function definition (but not on both)*