## Recursive Functions

## 

* A function is recursive if it calls itself.
* Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that base case.
* One or more base cases are used to stop recursion.

If (some case)

Solution case

Else

Recursive call

* A recursive method must eventually be reduced to a stopping case or a base case in order to terminate. When it reaches a stopping case, the method returns a result to its caller. The caller then performs a computation and returns the result to its own caller. This process continues until the result is passed back to the original caller.
* In some cases, using recursion enables you to give a natural, straightforward, simple solution to a problem that would otherwise be difficult to solve.
* Many mathematical functions are defined using recursion.

#### Example 1

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\* Function recur\_func will ultimately cause the program to \**

*\* crash. Each time a function is called, a section of memory is \**

*\* allocated in a part of RAM known as the stack. Eventually \**

*\* these recursive function calls will use up all available \**

*\* stack memory and cause it to overflow. \**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

#include <iostream>

using namespace std;

void recur\_func( );

int main( ){

recur\_func( );

}

void recur\_func( ) {

cout << “hello”;

recur\_func( ); *//Function calling itself*

}



#### Example 2

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\* This is another example of recursion. Function recur will \**

*\* display 123. The function will continue to call itself until \**

*\* it “x” is incremented to 4. Notice that in each call the \**

*\* the value of “x” is incremented by 1. \**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

#include <iostream>

using namespace std;

void recur\_func(int x);

int main()

{

int x=1;

*//This loop will display 123*

while (x<4)

cout << x++;

recur\_func(1); *//Emulates the above loop using recursion*

}

void recur\_func(int x)

{

if (x!=4){ *//Base case*

cout << x;

recur\_func(x+1);

}

}



Example 3

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\* This is another example of recursion. Function fact will \**

*\* calculate the factorial of 4. Notice that the fact function \**

*\* will stop calling itself once the value of “n” is equal to 0. \**

*\* The value of “n” is decremented every time the fact function \**

*\* calls itself. The formula for calculating the factorial of a \**

*\* number is as follows: \**

*\* 0!=1 \**

*\* x!=x\*(x-1)! \**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

#include <iostream>

using namespace std;

int fact(int x);

int main()

{

cout << fact(4);

}

int fact(int x)

{

int result;

if (x == 0) *//Base case*

result=1;

else

result=x\*fact(x-1);

return (result);

}



**Telephone analogy**

#### Example 4

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\* In this example, a recursive method is written for computing \**

*\* a fibonacci number fib(n), given index n. The series can be \**

*\* recursively defined as follows: \**

*\* fib(0)=0 \**

*\* fib(1)=1 \**

*\* fib(x)=fib(x-2) + fib(x-1) \**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

#include <iostream>

using namespace std;

int fib(int x);

int main()

{

cout << fib(3);

getch();

}

int fib (int x)

{

int result;

if (x == 0)

return (0); *//Base case*

else if (x==1)

return (1); *//Base case*

else

return (fib(x-1)+ fib(x-2));

}



**What are the base cases in all the above examples?**

What is the difference between the following?

void reverse(int num){

if (num!=2){

reverse(num+1);

}

cout<<num;

}

**Versus**

void reverse2(){

static int num=0;

if (num!=2){

num++;

reverse2();

}

cout<<num;

}

**Tail versus Non-tail recursion**

A function call is said to be tail recursive if there is nothing to do after the function returns except return its value. Since the current recursive instance is done executing at that point, saving its stack frame is a waste. Specifically, creating a new stack frame on top of the current, finished, frame is a waste.

**Non-tail recursion**

int sum(x){

if (x == 1)

return x;

else

return x + sum(x - 1);

}

If you called sum(5), this is what we

5 + sum(4)

5 + (4 + sum(3))

5 + (4 + (3 + sum(2)))

5 + (4 + (3 + (2 + sum(1))))

5 + (4 + (3 + (2 + 1)))

15

**Tail recursion**

int sum(x, total){

if (x == 0)

return total;

else

return sum(x - 1, total + x);

}

Here's the sequence of events that would occur if you called sum(5, 0)

sum (5, 0)

sum (4, 5)

sum (3, 9)

sum (2, 12)

sum (1, 14)

sum (0, 15)

15

**Iteration**

Tail recursion can be replaced by iteration to remove recursion from the solution as in thenext example.

int sum(x){

int total=0;

for (int i=x; i>0; i--)

total=total+i

}

**Non-tail**

int linkedListLength(node \*p) {

if (p == NULL)

return 0;

else

return linkedListLength(p->next) + 1;

}

Linked list

|  |  |  |  |
| --- | --- | --- | --- |
| 10 1 Next=20 | 20 2 Next=30 | 30 3 Next=40 | 40 4 Next=NULL |

|  |  |
| --- | --- |
| P=10  return linkedListLength (20) + 1; | Return 3+1 4 |
| P=20  return linkedListLength(30) + 1; | Return 2+1 3 |
| P=30  return linkedListLength(40) + 1; | Return 1+1 2 |
| P=40  Return linkedListLength(0) +1 | Return (0+1) 1 |
| 0 | Return 0 |

**tail**

void oddDescending2(int n) {

if (n >= 1) {

cout << n << ’ ’;

oddDescending2(n-2);

}

}

7

|  |  |
| --- | --- |
| cout<< 7  oddDescending(5) |  |
| cout<<5  oddDescending(3) |  |
| cout<<3  OddDescending(1) |  |
| cout<<1  oddDescending(-1) | Nothing happens on the way back |
|  |  |

void oddDscending(int n) {

if (n > 0)

if (n % 2 == 1)

oddDscending2(n);

else

oddDescending2(n-1);

}

void oddDescending2(int n) {

if (n >= 1) {

oddDescending2 (n-2);

cout << n << ’ ’;

}

}

If the number is odd then just call the function. If is even, start one lower to make it odd.

7

|  |  |
| --- | --- |
| N=7  oddDescending(5) | Cout<<7 |
| N=5  oddDescending(3) | Cout<<5 |
| N=3  OddDescending(1) | Cout<<3 |
| N=1  OddDescending(-1) | Cout<<1 |
| N=-1 DONE | It will start displaying on the way back |

void decToBin(int n) {

if (n > 0) {

decToBin(n/2);

cout << n%2;

}

}

8

|  |  |
| --- | --- |
| N=8  decToBin(4) | cout<<8%2 0 |
| N=4  decToBin(2) | cout << 4%2; 0 |
| N=2  decToBin(1) | cout << 2%2; 0 |
| N=1  decToBin(0) | cout << 1%2; 1 |
| Done (Going back has purpose) | Will start displaying on the way back |

void reverse(char \*a){

if (\*a!=0){

reverse(a+1);

cout<<\*a;

}

}

|  |  |  |
| --- | --- | --- |
| H | I | M |
| 10 | 11 | 12 13=NULL |

|  |  |
| --- | --- |
| 10  Reverse(11) | Cout<<H |
| 11  Reverse(12) | Cout<<i |
| 12  Reverse(13) | Cout<<m |
| 13-NULL DONE | Displays on the way back |

void reverse2(){

int ch;

ch=cin.get();

if (ch!='\n')

reverse2();

cout.put(ch);

}

|  |  |
| --- | --- |
| H=cin.get()  Reverse() | Cou<<h |
| I=cin.get()  Reverse() | Cout<<i |
| M=cin.get()  Reverse() | Cout<<m |
| \n=cin.get DONE | Displays on the way back |
| Replace int ch with static int ch | |
| Static just gives you a single variable. This means that we don’t have one for each function. Whatever happens at the end, replaces what it was before. In this case, it will display a bunch of m | |

bool charInStr (char ch, char \*s) {

if (\*s == ’\0’)

return false;

else if (ch == \*s)

return true;

else

return charInStr(ch,++s);

}

A, BAbdul

|  |  |
| --- | --- |
| A, B  Return charInStr(A,a) | Return true |
| A, A  Return true DONE |  |
| Nothing happens on the way up  What stops this is when it is the end of the string or a match is found |  |

int freqInStr(char ch, char \*s) {

if (\*s == ’\0’)

return 0;

else if (ch == \*s)

return freqInStr(ch,++s) + 1;

else

return freqInStr(ch,++s);

}

a, baad

|  |  |
| --- | --- |
| a, b  return freqInStr(a, a) | Return 2 |
| a, a  return(freqInStr(a, a) + 1 | Return 1+ 1 =2 |
| a, a  return freqInStr(a,d) + 1 | Return 0+1 =1 |
| a, d  return freqInStr(a, NULL) | Return 0 |
| Return 0 |  |
| Something does happen on the way back |  |