Recursion in C

KAPIL K NAGWANSHI
DOCSE NIT-Raipur

Kapil14s@yahoo.co.uk

"Definition of Recursion"

Definition: Functions are recursive if they call themselves. For this to work, the following conditions apply :

- There must be a solveable problem.
- There must be a terminating clause.

Factorials are often cited as a classic example of using recursion.

The factorial f(n) = n * (n-1) * (n-2) * .. 1.

For example factorial 5 (written as 5!) = 5 x 4 x 3 x 2 x 1 = 120. The function below returns the factorial of the parameter n.

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

int value=factorial(6) ;
```

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. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". This makes it sound very similar to a loop because it repeats the same code, and in some ways it *is* similar to looping. On the other hand, recursion makes it easier to express ideas in which the result of the recursive call is necessary to complete the task. Of course, it must be possible for the "process" to sometimes be completed without the recursive call. One simple example is the idea of building a wall that is ten feet high; if I want to build a ten foot high wall, then I will first build a 9 foot high wall, and then add an extra foot of bricks. Conceptually, this is like saying the "build wall" function takes a height and if that height is greater than one, first calls itself to build a lower wall, and then adds one a foot of bricks. A simple example of recursion would be:

```
void recurse()
{
    recurse(); /* Function calls itself */
}

int main()
{
    recurse(); /* Sets off the recursion */
    return 0;
```

```
}
```
This program will not continue forever, however. The computer keeps function calls on a stack and once too many are called without ending, the program will crash. Why not write a program to see how many times the function is called before the program terminates?

```
#include <stdio.h>

void recurse ( int count ) /* Each call gets its own copy of count */
{
    printf( "%d\n", count );
    /* 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 one */
  return 0;
}
```
This simple program will show the number of times the recurse function has been called by initializing each individual function call's count variable one greater than it was previous by passing in count + 1. Keep in mind that it is not a function call restarting itself; it is hundreds of function calls that are each unfinished.

The best way to think of recursion is that each function call is a "process" being carried out by the computer. If we think of a program as being carried out by a group of people who can pass around information about the state of a task and instructions on performing the task, each recursive function call is a bit like each person asking the next person to follow the same set of instructions on some part of the task while the first person waits for the result.

At some point, we're going to run out of people to carry out the instructions, just as our previous recursive functions ran out of space on the stack. There needs to be a way to avoid this! To halt a series of recursive calls, a recursive function will have a condition that controls when the function will finally stop calling itself. The condition where the function will not call itself is termed the base case of the function. Basically, it will usually be 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).

A quick example:

```
void count_to_ten ( int count )
{
    /* we only keep counting if we have a value less than ten
       if ( count < 10 )
       {
           count_to_ten( count + 1 );
       }
}
int main()
{
  count_to_ten ( 0 );
}
```

This program ends when we've counted to ten, or more precisely, when count is no longer less than ten. This is a good base case because it means that if we have an input greater than ten, we'll stop immediately. If we'd chosen to stop when count equalled ten, then if the function were called with the input 11, it would run of of memory before stopping.

Notice that so far, we haven't done anything with the result of a recursive function call. Each call takes place and performs some action that is then ignored by the caller. It is possible to get a value back from the caller, however. It's also possible to take advantage of the side effects of the previous call. In either case, once a function has called itself, it will be ready to go to the next line after the call. It can still perform operations. One function you could write could print out the numbers 123456789987654321. How can you use recursion to write a function to do this? Simply have it keep incrementing a variable passed in, and then output the variable twice: once before the function recurses, and once after.

```
void printnum ( int begin )
{
    printf( "%d", begin );
    if ( begin < 9 )         /* The base case is when begin is no
longer */
    {                           /* less than 9 */
        printnum ( begin + 1 );
    }
    /* display begin again after we've already printed everything from
1 to 9
     * and from 9 to begin + 1 */
    printf( "%d", begin );
}
```

This function works because it will go through and print the numbers begin to 9, and then as each printnum function terminates it will continue printing the value of begin in each function from 9 to begin.

This is, however, just touching on the usefulness of recursion. Here's a little challenge: use recursion to write a program that returns the factorial of any number greater than 0. (Factorial is number*number-1*number-2...*1).

Hint: Your function should recursively find the factorial of the smaller numbers first, i.e., it takes a number, finds the factorial of the previous number, and multiplies the number times that factorial...have fun. :-)

4.3. Recursion and argument passing

So far, we've seen how to give functions a type (how to declare the return value and the type of any arguments the function takes), and how the definition is used to give the body of the function. Next we need to see what the arguments can be used for.

4.3.1. Call by value

The way that C treats arguments to functions is both simple and consistent, with no exceptions to the single rule.

When a function is called, any arguments that are provided by the caller are simply treated as expressions. The value of each expression has the appropriate conversions applied and is then used to initialize the corresponding formal parameter in the called function, which behaves in exactly the same way as any other local variables in the function. It's illustrated here:

```
void called_func(int, float);

main(){
```

```
	called_func(1, 2*3.5);
	exit(EXIT_SUCCESS);
}

void
called_func(int iarg, float farg){
	float tmp;

	tmp = iarg * farg;
}
```
Example 4.6

The arguments to `called_func` in `main` are two expressions, which are evaluated. The value of each expression is used to initialize the parameters `iarg` and `farg` in `called_func`, and the parameters are indistinguishable from the other local variable declared in `called_func`, which is `tmp`.

The initialization of the formal parameters is the last time that any communication occurs between the caller and the called function, except for the return value.

For those who are used to FORTRAN and `var` arguments in Pascal, where a function *can* change the values of its arguments: forget it. You cannot affect the values of a function's actual arguments by anything that you try. Here is an example to show what we mean.

```
#include <stdio.h>
#include <stdlib.h>
main(){
	void changer(int);
	int i;

	i = 5;
	printf("before i=%d\n", i);
	changer(i);
	printf("after i=%d\n", i);
	exit(EXIT_SUCCESS);
}

void
changer(int x){
	while(x){
		printf("changer: x=%d\n", x);
		x--;
	}
}
```
Example 4.7

The result of running that is:

```
before i=5
changer: x=5
changer: x=4
changer: x=3
changer: x=2
changer: x=1
after i=5
```

The function `changer` uses its formal parameter `x` as an ordinary variable—which is exactly what it is. Although the value of `x` is changed, the variable `i` (in main) is unaffected. That is the whole point—the arguments in C are passed into a function by their value only, no changes made by the function are passed back.

4.3.2. Call by reference

It is possible to write functions that take *pointers* as their arguments, giving a form of call by reference.

4.3.3. Recursion

With argument passing safely out of the way we can look at recursion. Recursion is a topic that often provokes lengthy and unenlightening arguments from opposing camps. Some think it is wonderful, and use it at every opportunity; some others take exactly the opposite view. Let's just say that when you need it, you really *do* need it, and since it doesn't cost much to put into a language, as you would expect, C supports recursion.

Every function in C may be called from any other or itself. Each invocation of a function causes a new allocation of the variables declared inside it. In fact, the declarations that we have been using until now have had something missing: the keyword `auto`, meaning ‘ automatically allocated’ .

```
/* Example of auto */
main(){
      auto int var_name;
      .
      .
      .
}
```

The storage for auto variables is automatically allocated and freed on function entry and return. If two functions both declare large automatic arrays, the program will only have to find room for both arrays if both functions are active at the same time. Although auto is a keyword, it is never used in practice because it's the default for internal declarations and is invalid for external ones. If an explicit initial value (see ‘ initialization’) isn't given for an automatic variable, then its value will be unknown when it is declared. In that state, any use of its value will cause undefined behaviour.

The real problem with illustrating recursion is in the selection of examples. Too often, simple examples are used which don't really get much out of recursion. The problems where it really helps are almost always well out of the grasp of a beginner who is having enough trouble trying to sort out the difference between, say, definition and declaration without wanting the extra burden of having to wrap his or her mind around a new concept as well. The chapter on data structures will show examples of recursion where it is a genuinely useful technique.

The following example uses recursive functions to evaluate expressions involving single digit numbers, the operators `*`, `%`, `/`, `+`, `-` and parentheses in the same way that C does. (Stroustrup[1], in his book about C++, uses almost an identical example to illustrate recursion. This happened purely by chance.) The whole expression is evaluated and its value printed when a character not in the ‘ language’ is read. For simplicity no error checking is performed. Extensive use is made of the `ungetc` library function, which allows the last character read by `getchar` to be ‘ unread’ and become once again the next character to be read. Its second argument is one of the things declared in `stdio.h`.

Those of you who understand BNF notation might like to know that the expressions it will understand are described as follows:

```
<primary> ::= digit | (<exp>)
<unary>   ::= <primary> | -<unary> | +<unary>
<mult>    ::= <unary> | <mult> * <unary> |
              <mult> / <unary> | <mult> % <unary>
<exp>     ::= <exp> + <mult> | <exp> - <mult> | <mult>
```

The main places where recursion occurs are in the function `unary_exp`, which calls itself, and at the bottom level where `primary` calls the top level all over again to evaluate parenthesized expressions.

If you don't understand what it does, try running it. Trace its actions by hand on inputs such as

```
1
1+2
1+2 * 3+4
1+--4
1+(2*3)+4
```

That should keep you busy for a while!

```
/*
* Recursive descent parser for simple C expressions.
* Very little error checking.
*/

#include <stdio.h>
#include <stdlib.h>

int expr(void);
int mul_exp(void);
int unary_exp(void);
int primary(void);

main(){
      int val;

      for(;;){
              printf("expression: ");
              val = expr();
              if(getchar() != '\n'){
                      printf("error\n");
                      while(getchar() != '\n')
                              ; /* NULL */
              } else{
                      printf("result is %d\n", val);
              }
      }
      exit(EXIT_SUCCESS);
}

int
expr(void){
      int val, ch_in;

      val = mul_exp();
      for(;;){
```

```
		switch(ch_in = getchar()){
		default:
			ungetc(ch_in,stdin);
			return(val);
		case '+':
			val = val + mul_exp();
			break;
		case '-':
			val = val - mul_exp();
			break;
		}
	}
}

int
mul_exp(void){
	int val, ch_in;

	val = unary_exp();
	for(;;){
		switch(ch_in = getchar()){
		default:
			ungetc(ch_in, stdin);
			return(val);
		case '*':
			val = val * unary_exp();
			break;
		case '/':
			val = val / unary_exp();
			break;
		case '%':
			val = val % unary_exp();
			break;
		}
	}
}

int
unary_exp(void){
	int val, ch_in;

	switch(ch_in = getchar()){
	default:
		ungetc(ch_in, stdin);
		val = primary();
		break;
	case '+':
		val = unary_exp();
		break;
	case '-':
		val = -unary_exp();
		break;
	}
	return(val);
}

int
```

```
primary(void){
	int val, ch_in;

	ch_in = getchar();
	if(ch_in >= '0' && ch_in <= '9'){
		val = ch_in - '0';
		goto out;
	}
	if(ch_in == '('){
		val = expr();
		getchar();	/* skip closing ')' */
		goto out;
	}
	printf("error: primary read %d\n", ch_in);
	exit(EXIT_FAILURE);
out:
	return(val);
}
```

Example 4.8

Linear Recursive

A linear recursive function is a function that only makes a single call to itself each time the function runs (as opposed to one that would call itself multiple times during its execution). The factorial function is a good example of linear recursion.
Another example of a linear recursive function would be one to compute the square root of a number using Newton's method (assume EPSILON to be a very small number close to 0):

Code: C

```c
double my_sqrt(double x, double a)
  {
      double difference = a*x-x;
      if (difference < 0.0) difference = -difference;
      if (difference < EPSILON) return(a);
      else return(my_sqrt(x,(a+x/a)/2.0));
  }
```

Tail recursive

Tail recursion is a form of linear recursion. In tail recursion, the recursive call is the last thing the function does. Often, the value of the recursive call is returned. As such, tail recursive functions can often be easily implemented in an iterative manner; by taking out the recursive call and replacing it with a loop, the same effect can generally be achieved. In fact, a good compiler can recognize tail recursion and convert it to iteration in order to optimize the performance of the code.
A good example of a tail recursive function is a function to compute the GCD, or Greatest Common Denominator, of two numbers:

Code: C

```c
int gcd(int m, int n)
  {
      int r;

      if (m < n) return gcd(n,m);

      r = m%n;
      if (r == 0) return(n);
```

```
    else return(gcd(n,r));
  }
```

Binary Recursive

Some recursive functions don't just have one call to themself, they have two (or more). Functions with two recursive calls are referred to as binary recursive functions.
The mathematical combinations operation is a good example of a function that can quickly be implemented as a binary recursive function. The number of combinations, often represented as nCk where we are choosing n elements out of a set of k elements, can be implemented as follows:

Code: C

```
int choose(int n, int k)
  {
      if (k == 0 || n == k) return(1);
      else return(choose(n-1,k) + choose(n-1,k-1));
  }
```

Exponential recursion

An exponential recursive function is one that, if you were to draw out a representation of all the function calls, would have an exponential number of calls in relation to the size of the data set (exponential meaning if there were n elements, there would be O(an) function calls where a is a positive number).
A good example an exponentially recursive function is a function to compute all the permutations of a data set. Let's write a function to take an array of n integers and print out every permutation of it.

Code: C

```
void print_array(int arr[], int n)
  {
      int i;
      for(i=0; i<n; i) printf("%d ", arr[i]);
      printf("\n");
  }

  void print_permutations(int arr[], int n, int i)
  {
      int j, swap;
      print_array(arr, n);
      for(j=i+1; j<n; j) {
          swap = arr[i]; arr[i] = arr[j]; arr[j] = swap;
          print_permutations(arr, n, i+1);
          swap = arr[i]; arr[i] = arr[j]; arr[j] = swap;
      }
  }
```

To run this function on an array arr of length n, we'd do print_permutations(arr, n, 0) where the 0 tells it to start at the beginning of the array.

Nested Recursion

In nested recursion, one of the arguments to the recursive function is the recursive function itself! These functions tend to grow extremely fast. A good example is the classic mathematical function, " Ackerman's function. It grows very quickly (even for small values of x and y, Ackermann(x,y) is extremely large) and it cannot be computed with only definite iteration (a completely defined for() loop for example); it requires indefinite iteration (recursion, for example).

Ackerman's function

Code: C

```c
int ackerman(int m, int n)
  {
      if (m == 0) return(n+1);
      else if (n == 0) return(ackerman(m-1,1));
      else return(ackerman(m-1,ackerman(m,n-1)));
  }
```

Try computing ackerman(4,2) by hand... have fun!

Mutual Recursion

A recursive function doesn't necessarily need to call itself. Some recursive functions work in pairs or even larger groups. For example, function A calls function B which calls function C which in turn calls function A.
A simple example of mutual recursion is a set of function to determine whether an integer is even or odd. How do we know if a number is even? Well, we know 0 is even. And we also know that if a number n is even, then n - 1 must be odd. How do we know if a number is odd? It's not even!

Code: C

```c
int is_even(unsigned int n)
  {
      if (n==0) return 1;
      else return(is_odd(n-1));
  }

  int is_odd(unsigned int n)
  {
      return (!iseven(n));
  }
```

I told you recursion was powerful! Of course, this is just an illustration. The above situation isn't the best example of when we'd want to use recursion instead of iteration or a closed form solution. A more efficient set of function to determine whether an integer is even or odd would be the following:

Code: C

```c
 int is_even(unsigned int n)
  {
      if (n % 2 == 0) return 1;
      else return 0;
  }

  int is_odd(unsigned int n)
  {
      if (n % 2 != 0) return 1;
      else return 0;
  }
```

/ / / / / /
With Best Wishesh
Kapil Engineer