CS 162C++
Recursion

Recursion is a programming method where a function calls itself directly or calls a second function that calls the first again (indirect recursion).

Requirements

The function must call itself, directly or indirectly. It must have a base case that stops the recursion from continuing indefinitely, and when it calls itself it must do so on a smaller problem. That is, the arguments in successive recursive calls must approach the base case as a limit. Since a recursive program calls itself, implementation requires a stack to save the sequence of calls and returns.

Advantages and Disadvantages

The primary advantage of recursion is that it enables an improved understanding of complex problems and simplifies their coding.

There are two disadvantages: each call uses up memory for the return address and parameters and requires the overhead of a function call. For this reason, an iterative solution is often more efficient both in memory use and in program execution.

Designing a recursive function

Note that anything you can code as recursive, you could also code as iterative. All that is needed is a stack to save the arguments.

Before you start coding a recursive function, determine what your base case(s) are. If it is a Boolean function, you will need two, one for true and one for false. If you are walking down a list of pointers, you terminate when you reach a null at the end. If you are working through an array, you terminate when you reach the end of the array. Then you determine if you will build up the result by passing it as an argument, or implicitly by adding it to the return value as shown in the following two examples:

```
string recursive_reverse1(string message)
{
    int length = static_cast<int>(message.length());
    if ( length == 0 )
        return "";
    return message[length - 1] + recursive_reverse1(message.substr(0, length - 1));
}

string recursive_reverse2(string input, string output)
{
    int length = static_cast<int>(input.length());
    if ( length == 0 )
        return output;

    return recursive_reverse2(input.substr(0,length-1),output+input[length-1]);
}

int main()
{
    string message = "This is a test";
    cout << recursive_reverse1(message) << endl;
    cout << recursive_reverse2(message,"") << endl;
    return 0;
}
```

Example 1 Factorial

A factorial calculation is a good example of a program that can be readily coded as a recursive algorithm. First lets consider an iterative solution: When programming recursively, it is more common to have the argument decrease towards a limit instead of increasing to one, so lets write it this way:

```
#include <iostream>

using namespace std;

int main()
{
    const int LIMIT = 6;
    int result = 1;

    for(int i = LIMIT; i > 0; i--)
    {
        result = result * i;
        cout << "In loop, result is " << result << endl;
    }

    cout << "When done, result is "<< result << endl;

    return 0;
}
```

Now we can replace this with a simple recursive function, we will pass in the current multiplier at each point, pass in a value for our result, and quit when we hit reach zero.

```
#include <iostream>

using namespace std;

int recursive_fact(int start, int result)
{
    if ( start <= 0 )
        return result;
    result *= start;
    cout << "In function, result is " << result << endl;
    return recursive_fact(start-1, result);
}

int main()
{
    const int LIMIT = 6;
    int result = 1;

    result = recursive_fact(LIMIT, result);

    cout << "When done, result is "<< result << endl;

    return 0;
}
```

Example 2 Linear search

A linear search is not normally coded recursively as a for loop does it so efficiently, but I am considering it here as a step before we do a binary search. First, the iterative approach where pass in a value, and compare it to each location in the array in turn, quitting when we find it or when we are done with the array.

```
#include <iostream>

using namespace std;


int main()
{
    const int SIZE = 6;
    int arr[SIZE] = {1, 6, 2, 12, 19, -1};
    bool found = false;

    //int value = 12;
    int value = 7;

    for(int i = 0; i < SIZE; i++)
        if ( value == arr[i] )
        {
            found = true;
            break;
        }
    cout << "The value was" << (found?" ":" not ") << "present" << endl;

    return 0;
}
```

Now we consider the recursive method. The only difference here is that we have to pass in the array also, since it was not defined in the scope of the recursive function. As in the loop, we increment index before the next pass.

```
#include <iostream>

using namespace std;

bool recursive_linear(int arr[], int size, int index, int value)
{
    if ( index == size )
        return false;
    if ( arr[index] == value )
        return true;
    return recursive_linear(arr, size, ++index, value);
}

int main()
{
    const int SIZE = 6;
    int arr[SIZE] = {1, 6, 2, 12, 19, -1};
    bool found = false;

    int value = 12;
    //int value = 7;

    found = recursive_linear(arr, SIZE, 0, value);

    cout << "The value was" << (found?" ":" not ") << "present" << endl;

    return 0;
}
```

Example 3 Binary search

Now lets look at a binary search, this is more naturally coded as a recursive function since you are shrinking the space searched until you find it or nothing is left to look at. First the iterative approach, here we pass in the array, the upper and lower limits of our search, and the value to search for. If we find it, we return true. If the limits cross, we return false;

```
bool iterative_search(int theArray[], int lower, int upper, int value)
{
    bool found = false;
    while ( lower < upper and !found )
    {
        int index = (lower + upper) / 2;
        if ( theArray[index] == value )
            found = true;
        else if ( theArray[index] > value )
            upper = index - 1;
        else
        {
            lower = index + 1;
        }
    }
    return found;
}
```

Now we consider the recursive method. Notice that with this implementation, we pass the same set of arguments.

```
bool recursive_search(int theArray[], int lower, int upper, int value)
{
    if ( lower > upper )
        return false;

    int index = (lower + upper) / 2;

    if ( theArray[index] == value )
        return true;

    if ( theArray[index] > value )
        return recursive_search(theArray, lower, index - 1, value);
    else
        return recursive_search(theArray, index + 1, upper, value);
}
```

And here is the main that calls either function.

```
int main()
{
    const int SIZE = 15;
    int theArray[SIZE] = {1, 3, 5, 7, 9, 12, 14, 16, 18, 20, 22, 30, 33, 40, 44};

    bool found;
    //int value = 25;
    int value = 50;

    found = recursive_search(theArray, 0, SIZE, value);
    //found = interative_search(theArray, 0, SIZE, value);

    cout << "The number " << value << ( found?" was ":" was not " ) << "found." << endl;

    return 0;
}
```