How to replace recursive functions using stack and while-loop to avoid the stack-overflow

Woong Gyu La, 1 Jul 2015 MIT

This article explains 10 rules (steps) for replacing the recursive functions using stack and while-loop to avoid the stack-overflow.

Download the recursive simulation samples - 12.4 KB

Table of contents

Introduction

There are cases where we prefer to use recursive functions such as sort (Merge Sort) or tree operations (heapify up / heapify down). However, if the recursive function goes too deep in some environments, such as in Visual C++ code, an unwanted result might occur such as a stack-overflow. Many professional developers probably already know how to replace recursive functions to avoid stack-overflow problems in advance by replacing with iterative function or using stack (heap stack) and while-loop (*recursive simulation function*). However I thought it would be a great idea to share simple and general methods (or guidelines) to replace the recursive functions using stack (heap stack) and while-loop to avoid the stack-overflow to help novice developers.

Purpose of the Simulated function

If you are using recursive function, since you don't have control on call stack and the stack is limited, the stack-overflow/heap-corruption might occur when the recursive call's depth gets too deep. The purpose of simulated function is moving the call stack to stack in heap, so the you can have more control on memory and process flow, and avoid the stack-overflow. It will be much better if it is replaced by iterative function, however in order to do that, it takes time and experience to handle every recursive function in proper way, so this article is a simple guide to help the novice developers to avoid the stack-overflow by using the recursive function, when they are not ready yet to handle everything in proper way.

Pros and Cons of Recursive and Simulated function

Recursive function

- Pros
  - Very intuitive about the algorithm
    - See the examples in RecursiveToLoopSamples.zip.
- Cons
  - May occur "Stack-overflow," or "Heap corruption"
    - Try to run `IsEvenNumber` function (Recursive) and `IsEvenNumberLoop` function (simulated) of "*MutualRecursion.h*" in RecursiveToLoopSamples.zip with "10000" as its parameter input.

Hide Copy Code

```
#include "MutualRecursion.h"

bool result = IsEvenNumberLoop(10000); // returns successfully
bool result2 = IsEvenNumber(10000);     // stack-overflow error occurs
```

Some people say that "(In order to fix the stack-overflow occurred by recursive function,) increase the MAX value of the stack to avoid the stack-overflow." However this is just temporary bandage, since if the recursive call gets deeper and deeper, the stack-overflow will most likely reappear.

Simulated function

- Pros
  - Can avoid "Stack-overflow," or "Heap corruption" errors.
  - More control on process flow and memory.
- Cons
  - Not very intuitive about the algorithm.
  - Hard to maintain the code.

10 Rules (steps) for replacing the recursive function with stack and while-loop

First rule

1. Define a new struct called "`Snapshot`". The purpose of this data structure is to hold any data used in the recursive structure, along with any state information.
2. Things to include in the "`Snapshot`" structure.
   a. the function argument that changes when the recursive function calls itself **However,** when the function argument is a reference, it does not need to be included in the `Snapshot` struct. Thus, in the following example, argument `n` should be included in the struct but argument `retVal` should not.
      - `void SomeFunc(int n, int &retVal);`
   b. the "`Stage`" variable (usually an `int` to switch into the correct process divisions)
      - Read "Sixth rule" for detail.
   c. local variables that will be used after returning from the function call (can happen during binary recursion or nested recursion)

Hide Copy Code

```
// Recursive Function "First rule" example
int SomeFunc(int n, int &retIdx)
{
   ...
   if(n>0)
   {
      int test = SomeFunc(n-1, retIdx);
      test--;
      ...
      return test;
   }
   ...
   return 0;
}
```

Hide Copy Code

```
// Conversion to Iterative Function
```

```
int SomeFuncLoop(int n, int &retIdx)
{
    // (First rule)
    struct SnapShotStruct {
       int n;        // - parameter input
       int test;     // - local variable that will be used
                     //     after returning from the function call
                     // - retIdx can be ignored since it is a reference.
       int stage;    // - Since there is process needed to be done
                     //     after recursive call. (Sixth rule)
    };
    ...
}
```

Second rule

1. Create a local variable at the top of the function. This value will represent the role of the return function in the recursive function.
   a. in the iterative function, it is more like a temporary return value holder for each recursive call within the recursive function, since a C++ function can only have one return type.
   b. if the recursive function's return type is void, then you can simply ignore creating this variable.
   c. If there is any default return value then initialize this local variable with default value returning.

Hide Copy Code

```
// Recursive Function "Second rule" example
int SomeFunc(int n, int &retIdx)
{
   ...
   if(n>0)
   {
      int test = SomeFunc(n-1, retIdx);
      test--;
      ...
      return test;
   }
   ...
   return 0;
}
```

Hide Copy Code

```
// Conversion to Iterative Function
int SomeFuncLoop(int n, int &retIdx)
{
     // (First rule)
    struct SnapShotStruct {
       int n;        // - parameter input
       int test;     // - local variable that will be used
                     //     after returning from the function call
                     // - retIdx can be ignored since it is a reference.
       int stage;    // - Since there is process needed to be done
                     //     after recursive call. (Sixth rule)
    };

    // (Second rule)
    int retVal = 0;  // initialize with default returning value
```

```
    ...
    // (Second rule)
    return retVal;
}
```

Third rule

1. Make a stack with the "Snapshot" struct type.

Hide Copy Code

```
// Recursive Function "Third rule" example

// Conversion to Iterative Function
int SomeFuncLoop(int n, int &retIdx)
{
     // (First rule)
    struct SnapShotStruct {
       int n;        // - parameter input
       int test;     // - local variable that will be used
                     //     after returning from the function call
                     // - retIdx can be ignored since it is a reference.
       int stage;    // - Since there is process needed to be done
                     //     after recursive call. (Sixth rule)
    };

    // (Second rule)
    int retVal = 0;  // initialize with default returning value

    // (Third rule)
    stack<SnapShotStruct> snapshotStack;
    ...
    // (Second rule)
    return retVal;
}
```

Fourth rule

1. Create a new "Snapshot" instance, and initialize with parameters that are input into the iterative function and the start "Stage" number.
2. Push the Snapshot instance into the empty stack.

Hide Shrink ▲ Copy Code

```
// Recursive Function "Fourth rule" example

// Conversion to Iterative Function
int SomeFuncLoop(int n, int &retIdx)
{
     // (First rule)
    struct SnapShotStruct {
       int n;        // - parameter input
       int test;     // - local variable that will be used
                     //     after returning from the function call
                     // - retIdx can be ignored since it is a reference.
       int stage;    // - Since there is process needed to be done
                     //     after recursive call. (Sixth rule)
    };

    // (Second rule)
    int retVal = 0;  // initialize with default returning value
```

```
    // (Third rule)
    stack<SnapShotStruct> snapshotStack;

    // (Fourth rule)
    SnapShotStruct currentSnapshot;
    currentSnapshot.n= n;          // set the value as parameter value
    currentSnapshot.test=0;        // set the value as default value
    currentSnapshot.stage=0;       // set the value as initial stage

    snapshotStack.push(currentSnapshot);

    ...
    // (Second rule)
    return retVal;
}
```

Fifth rule

1. Make a `while` loop which continues to loop while the stack is **not** empty.
2. At each iteration of the `while` loop, pop a `Snapshot` object from the stack

Hide Shrink Copy Code

```
// Recursive Function "Fifth rule" example

// Conversion to Iterative Function
int SomeFuncLoop(int n, int &retIdx)
{
     // (First rule)
    struct SnapShotStruct {
       int n;        // - parameter input
       int test;     // - local variable that will be used
                     //     after returning from the function call
                     // - retIdx can be ignored since it is a reference.
       int stage;    // - Since there is process needed to be done
                     //     after recursive call. (Sixth rule)
    };
    // (Second rule)
    int retVal = 0;  // initialize with default returning value
    // (Third rule)
    stack<SnapShotStruct> snapshotStack;
    // (Fourth rule)
    SnapShotStruct currentSnapshot;
    currentSnapshot.n= n;          // set the value as parameter value
    currentSnapshot.test=0;        // set the value as default value
    currentSnapshot.stage=0;       // set the value as initial stage
    snapshotStack.push(currentSnapshot);
    // (Fifth rule)
    while(!snapshotStack.empty())
    {
       currentSnapshot=snapshotStack.top();
       snapshotStack.pop();
       ...
    }
    // (Second rule)
    return retVal;
}
```

Sixth rule

1. Split the stages into two (for the case where there is only a single recursive call in the recursive function). The first case represents the code in the recursive function that is processed before the next recursive call is made, and the second case represents the code that is processed when the next recursive call returns (and when a return value is possibly collected or accumulated before the function returns it).
2. In the situation where there are two recursive calls within a function, there must be three stages:
   a. ** (Stage 1 --> recursive call --> (returned from first recursive call) Stage 2 (recursive call within stage 1)--> (return from second recursive call) Stage 3
3. In the situation where there are three different recursive calls then there must be at least 4 stages.
4. And so on.

Hide Copy Code

```
// Recursive Function "Sixth rule" example
int SomeFunc(int n, int &retIdx)
{
   ...
   if(n>0)
   {
      int test = SomeFunc(n-1, retIdx);
      test--;
      ...
      return test;
   }
   ...
   return 0;
}
```

Hide Shrink Copy Code

```
// Conversion to Iterative Function
int SomeFuncLoop(int n, int &retIdx)
{
     // (First rule)
    struct SnapShotStruct {
       int n;        // - parameter input
       int test;     // - local variable that will be used
                     //     after returning from the function call
                     // - retIdx can be ignored since it is a reference.
       int stage;    // - Since there is process needed to be done
                     //     after recursive call. (Sixth rule)
    };
    // (Second rule)
    int retVal = 0;  // initialize with default returning value
    // (Third rule)
    stack<SnapShotStruct> snapshotStack;
    // (Fourth rule)
    SnapShotStruct currentSnapshot;
    currentSnapshot.n= n;          // set the value as parameter value
    currentSnapshot.test=0;        // set the value as default value
    currentSnapshot.stage=0;       // set the value as initial stage
    snapshotStack.push(currentSnapshot);
    // (Fifth rule)
    while(!snapshotStack.empty())
    {
       currentSnapshot=snapshotStack.top();
       snapshotStack.pop();
       // (Sixth rule)
```

```
        switch( currentSnapshot.stage)
        {
        case 0:
           ...      // before ( SomeFunc(n-1, retIdx); )
           break;
        case 1:
           ...      // after ( SomeFunc(n-1, retIdx); )
           break;
        }
    }
    // (Second rule)
    return retVal;
}
```

Seventh rule

1. Switch into the process division according to the " Stage " variable
2. Do the relevant process

Hide Copy Code

```
// Recursive Function "Seventh rule" example
int SomeFunc(int n, int &retIdx)
{
   ...
   if(n>0)
   {
      int test = SomeFunc(n-1, retIdx);
      test--;
      ...
      return test;
   }
   ...
   return 0;
}
```

Hide Shrink Copy Code

```
// Conversion to Iterative Function
int SomeFuncLoop(int n, int &retIdx)
{
     // (First rule)
    struct SnapShotStruct {
       int n;        // - parameter input
       int test;     // - local variable that will be used
                     //     after returning from the function call
                     // - retIdx can be ignored since it is a reference.
       int stage;    // - Since there is process needed to be done
                     //     after recursive call. (Sixth rule)
    };

    // (Second rule)
    int retVal = 0;  // initialize with default returning value

    // (Third rule)
    stack<SnapShotStruct> snapshotStack;

    // (Fourth rule)
    SnapShotStruct currentSnapshot;
    currentSnapshot.n= n;          // set the value as parameter value
    currentSnapshot.test=0;        // set the value as default value
    currentSnapshot.stage=0;       // set the value as initial stage
```

```
    snapshotStack.push(currentSnapshot);

    // (Fifth rule)
    while(!snapshotStack.empty())
    {
       currentSnapshot=snapshotStack.top();
       snapshotStack.pop();

       // (Sixth rule)
       switch( currentSnapshot.stage)
       {
       case 0:
          // (Seventh rule)
          if( currentSnapshot.n>0 )
          {
             ...
          }
          ...
          break;
       case 1:
          // (Seventh rule)
          currentSnapshot.test = retVal;
          currentSnapshot.test--;
          ...
          break;
       }
    }
    // (Second rule)
    return retVal;
}
```

Eighth rule

1. If there is a return type for the recursive function, store the result of the loop iteration in a local variable (such as retVal).
2. This local variable will contain the final result of the recursive function when the `while` loop exits.

Hide Copy Code

```
// Recursive Function "Eighth rule" example
int SomeFunc(int n, int &retIdx)
{
   ...
   if(n>0)
   {
      int test = SomeFunc(n-1, retIdx);
      test--;
      ...
      return test;
   }
   ...
   return 0;
}
```

Hide Shrink Copy Code

```
// Conversion to Iterative Function
int SomeFuncLoop(int n, int &retIdx)
{
     // (First rule)
```

```
    struct SnapShotStruct {
       int n;        // - parameter input
       int test;     // - local variable that will be used
                     //     after returning from the function call
                     // - retIdx can be ignored since it is a reference.
       int stage;    // - Since there is process needed to be done
                     //     after recursive call. (Sixth rule)
    };
    // (Second rule)
    int retVal = 0;  // initialize with default returning value
    // (Third rule)
    stack<SnapShotStruct> snapshotStack;
    // (Fourth rule)
    SnapShotStruct currentSnapshot;
    currentSnapshot.n= n;          // set the value as parameter value
    currentSnapshot.test=0;        // set the value as default value
    currentSnapshot.stage=0;       // set the value as initial stage
    snapshotStack.push(currentSnapshot);
    // (Fifth rule)
    while(!snapshotStack.empty())
    {
       currentSnapshot=snapshotStack.top();
       snapshotStack.pop();
       // (Sixth rule)
       switch( currentSnapshot.stage)
       {
       case 0:
          // (Seventh rule)
          if( currentSnapshot.n>0 )
          {
             ...
          }
          ...
          // (Eighth rule)
          retVal = 0 ;
          ...
          break;
       case 1:
          // (Seventh rule)
          currentSnapshot.test = retVal;
          currentSnapshot.test--;
          ...
          // (Eighth rule)
          retVal = currentSnapshot.test;
          ...
          break;
       }
    }
    // (Second rule)
    return retVal;
}
```

Ninth rule

1. In a case where there are "`return`" keywords within the recursive function, the "`return`" keywords should be converted to "`continue`" within the "`while`" loop.

- a. In a case where the recursive function returns a value, then as stated in "Eighth rule," store the return value in the local variable (such as `retVal`), and then "`continue`"
  b. Most of the time, "Ninth rule" is optional, but it helps avoid logic error.

Hide Copy Code

```
// Recursive Function "Ninth rule" example
int SomeFunc(int n, int &retIdx)
{
   ...
   if(n>0)
   {
      int test = SomeFunc(n-1, retIdx);
      test--;
      ...
      return test;
   }
   ...
   return 0;
}
```

Hide Shrink Copy Code

```
// Conversion to Iterative Function
int SomeFuncLoop(int n, int &retIdx)
{
     // (First rule)
    struct SnapShotStruct {
       int n;        // - parameter input
       int test;     // - local variable that will be used
                     //     after returning from the function call
                     // - retIdx can be ignored since it is a reference.
       int stage;    // - Since there is process needed to be done
                     //     after recursive call. (Sixth rule)
    };
    // (Second rule)
    int retVal = 0;  // initialize with default returning value
    // (Third rule)
    stack<SnapShotStruct> snapshotStack;
    // (Fourth rule)
    SnapShotStruct currentSnapshot;
    currentSnapshot.n= n;          // set the value as parameter value
    currentSnapshot.test=0;        // set the value as default value
    currentSnapshot.stage=0;       // set the value as initial stage
    snapshotStack.push(currentSnapshot);
    // (Fifth rule)
    while(!snapshotStack.empty())
    {
       currentSnapshot=snapshotStack.top();
       snapshotStack.pop();
       // (Sixth rule)
       switch( currentSnapshot.stage)
       {
       case 0:
          // (Seventh rule)
          if( currentSnapshot.n>0 )
          {
             ...
          }
          ...
          // (Eighth rule)
          retVal = 0 ;

          // (Ninth rule)
          continue;
          break;
       case 1:
          // (Seventh rule)
          currentSnapshot.test = retVal;
```

```
         currentSnapshot.test--;
         ...
         // (Eighth rule)
         retVal = currentSnapshot.test;

         // (Ninth rule)
         continue;
         break;
      }
   }
   // (Second rule)
   return retVal;
}
```

Tenth rule (and the last...)

1. To convert the recursive call within the recursive function, in iterative function, make a new "Snapshot" object, initialize the new "Snapshot" object stage, set its member variables according to recursive call parameters, and push into the stack, and "continue"
2. If there is process to be done after the recursive call, change the stage variable of current "Snapshot" object to the relevant stage, and push into the stack before pushing the new "Snapshot" object into the stack.

Hide Copy Code

```
// Recursive Function "Tenth rule" example
int SomeFunc(int n, int &retIdx)
{
   ...
   if(n>0)
   {
      int test = SomeFunc(n-1, retIdx);
      test--;
      ...
      return test;
   }
   ...
   return 0;
}
```

Hide Shrink Copy Code

```
// Conversion to Iterative Function
int SomeFuncLoop(int n, int &retIdx)
{
     // (First rule)
    struct SnapShotStruct {
       int n;        // - parameter input
       int test;     // - local variable that will be used
                     //     after returning from the function call
                     // - retIdx can be ignored since it is a reference.
       int stage;    // - Since there is process needed to be done
                     //     after recursive call. (Sixth rule)
    };
    // (Second rule)
    int retVal = 0;  // initialize with default returning value
    // (Third rule)
    stack<SnapShotStruct> snapshotStack;
    // (Fourth rule)
    SnapShotStruct currentSnapshot;
    currentSnapshot.n= n;          // set the value as parameter value
```

```
    currentSnapshot.test=0;        // set the value as default value
    currentSnapshot.stage=0;       // set the value as initial stage
    snapshotStack.push(currentSnapshot);
    // (Fifth rule)
    while(!snapshotStack.empty())
    {
       currentSnapshot=snapshotStack.top();
       snapshotStack.pop();
       // (Sixth rule)
       switch( currentSnapshot.stage)
       {
       case 0:
          // (Seventh rule)
          if( currentSnapshot.n>0 )
          {
             // (Tenth rule)
             currentSnapshot.stage = 1;            // - current snapshot need to process after
                                                   //     returning from the recursive call
             snapshotStack.push(currentSnapshot);  // - this MUST pushed into stack before
                                                   //     new snapshot!
             // Create a new snapshot for calling itself
             SnapShotStruct newSnapshot;
             newSnapshot.n= currentSnapshot.n-1;   // - give parameter as parameter given
                                                   //     when calling itself
                                                   //     ( SomeFunc(n-1, retIdx) )
             newSnapshot.test=0;                   // - set the value as initial value
             newSnapshot.stage=0;                  // - since it will start from the
                                                   //     beginning of the function,
                                                   //     give the initial stage
             snapshotStack.push(newSnapshot);
             continue;
          }
          ...
          // (Eighth rule)
          retVal = 0 ;

          // (Ninth rule)
          continue;
          break;
       case 1:
          // (Seventh rule)
          currentSnapshot.test = retVal;
          currentSnapshot.test--;
          ...
          // (Eighth rule)
          retVal = currentSnapshot.test;
          // (Ninth rule)
          continue;
          break;
       }
    }
    // (Second rule)
    return retVal;
}
```

Simple Examples by types of recursion

- Please download RecursiveToLoopSamples.zip
- Unzip the file.
- Open the project with Visual Studio.

  - This project has been developed with Visual Studio 2008
- Sample project contains
  - Linear Recursion Example
  - Binary Recursion Example
  - Tail Recursion Example
  - Mutual Recursion Example
  - Nested Recursion Example

More Practical Example Sources

The below sources contain both a recursive version and a simulated version, where the simulated version has been derived using the above methodology.

- epQuickSort.h
- epMergeSort.h
- epKAryHeap.h
- epPatriciaTree.h

Why do the sources contain both the simulated version and the recursive version?

If you look at the source, you can easily notice the simulated versions look much more complex than the recursive versions. For those who don't know what the function does, it will be much harder to figure out what the function with the loop actually does. So I prefer to keep both versions, so people can easily test out simple inputs and outputs with the recursive version, and for huge operations, use simulated version to avoid stack overflow.

Conclusion

My belief is that when writing C/C++ or Java code, the recursive functions MUST be used with care to avoid the stack-overflow error. However as you can see from the examples, in many cases, the recursive functions are easy to understand, and easy to write with the downside of "if the recursive function call's depth goes too deep, it leads to stack-overflow error". So conversion from recursive function to simulated function is not for increasing readability nor increasing algorithmic performance, but it is simple way of evading the crashes or undefined behaviors/errors. As I stated above, I prefer to keep both recursive version and simulated version in my code, so I can use the recursive version for readability and maintenance of the code, and the simulated version for running and testing the code. It will be your choice how to write your code as long as you know the pros and cons for the choice, you are making.

Reference

- http://www.dreamincode.net/forums/topic/51296-types-of-recursion/
- EpLibrary 2.0

History

- 07.02.2015:- Broken link fixed
- 09.06.2013:- Typo fixed (Thanks to lovewubo)
- 08.22.2013:- Re-distributed under MIT License from GPL v3
- 08.10.2012: - Table of contents updated
- 08.04.2012: - Moved the article's subsection to "Howto"
- 07.23.2012: - Minor fixes on the article
- 07.13.2012: - Table of contents modified
    - Sections removed
    - Moved the article to Beginner section
    - Changed the wording
- 07.13.2012: - Table of contents added.
    - Titles modified.
    - New sections added.
        - Difference between Recursive and Iterative function
        - Pros and Cons of Recursive and Iterative approach
- 07.12.2012: - Sample bugs fixed.
    - Article re-organized.
    - Ninth and Tenth rule added.
    - Examples for each rule added.
- 07.11.2012: - Submitted the article.

License

Share

About the Author

Woong Gyu La

Software Developer

Korea (Republic Of)

Woong Gyu La had been working as a software developer for over 8 years.
His personal interests are improving his personal projects,

EpLibrary (Visual C++ Utility Library)
https://github.com/juhgiyo/EpLibrary[^]

EpOraLibrary (Oracle OCI Wrapper Library for Visual C++)
https://github.com/juhgiyo/EpOraLibrary[^]

EpServerEngine (Visual C++ WinSock Server/Client Engine)
https://github.com/juhgiyo/EpServerEngine[^]

And other projects can be found at
https://github.com/juhgiyo?tab=repositories[^]

Finally, my other articles can be found at
http://www.codeproject.com/Articles/juhgiyo#articles[^]

You can contact me at juhgiyo@gmail.com[^]

Comments and Discussions

You must Sign In to use this message board.

Search Comments Go

Profile popups Spacing Relaxed Layout Thread View Per page 50 Update

First Prev Next

initial input params when invoking FibNum2() in the example	xjs.xjtu	22-Aug-15 17:44	2
Good article	KP Lee	3-Jul-15 12:42	1
My vote of 5	Robert J. Good	3-Jul-15 9:34	1
My vote of 5	Member 10799665	3-Jul-15 9:32	1
Great Explanation...	Sachin Pachari	20-Nov-14 20:12	1
Beautiful Article!	Member 16789	18-Nov-14 20:19	1
recursive vs iterative	Member 11175294	26-Oct-14 3:30	4
Example within a loop	dudz1978	14-Apr-14 3:25	1
Best Authoritative	jediYL	27-Sep-13 10:21	1
My vote of 3	Jeroen Walter	6-Sep-13 3:28	2
is there a typo in step 10?	lovewubo	5-Sep-13 22:23	2
My vote of 5	pasztorpisti	26-Aug-13 10:29	1
My five star rating	Marco Giacinti	24-Aug-13 11:25	1
conversion of recursive to loop	Member 9914471	15-Mar-13 6:25	2
conversion of recursive to loop	madhurisamudrala	14-Mar-13 8:13	2
My vote of 5	Philippe Chessa	12-Nov-12 22:49	2
My vote of 5	Michael Haephrati מיכאל האפרתי	30-Oct-12 12:05	2
My vote of 4	Edward Keningham	11-Aug-12 3:24	2
My vote of 5	Mihai MOGA	8-Aug-12 5:27	2
My vote of 4	Marco Sorich	23-Jul-12 3:06	2
My vote of 5	gndnet	18-Jul-12 22:27	2
My vote of 5	Paul H Kim	15-Jul-12 19:52	2
My vote of 5	vivekvijay	14-Jul-12 0:46	2
First Rule: Refactoring = Define sufficient tests first before refactoring	Andreas Gieriet	13-Jul-12 23:31	2

Call stack vs stack<SnapShotStruct>??	**Matt T Heffron**	13-Jul-12 7:35	2
My vote of 3	**WPFanatic**	13-Jul-12 5:37	5
Difference between	**DaveAuld**	12-Jul-12 19:45	2
My vote of 5	**umlcat**	12-Jul-12 8:31	2

Last Visit: 31-Dec-99 18:00 Last Update: 15-Sep-15 23:15 Refresh 1

General News Suggestion Question Bug Answer Joke Rant Admin