Recursive, Trees, Recursion, Expressions, Stacks

Recursion

CSI 310: Lecture 14
But, running \texttt{fact(0)} calculates 0! = 0 is true and returns 1.

720, then computes 720! = 5040 and returns it.

For example, running \texttt{fact(7)} computes 7! = 6, \texttt{fact(6)} which returns

\begin{verbatim}
{ if \( n=0 \) return 1; else return \texttt{fact(n-1)}*n; }
\end{verbatim}

\texttt{int fact(int n)}

\textbf{Example:} When the function runs, calls the same function, either directly or indirectly,  recursion is \underline{re-implementation} \textit{it is worth it.}

\textbf{Definition:} A function is \underline{recursive} means the body of the function sometimes,

re-implemented \textbf{if it is worth it}.

\begin{itemize}
\item[(1)] Understanding, not just programming, data structures and algorithms.
\item[(2)] Powerful problem solving technique.
\item[(3)] Elegant way to write programs; performance can be improved with routine.
\end{itemize}

\textbf{Recursion:}

\begin{itemize}
\item Trees, Recursion, Expressions, Stacks are closely related.
\end{itemize}
{ int CAT6(int n) { steep(1000000000); return -1; }

{ int CAT5(int n) { if(n==1) return 1 else return CAT5(n - 1); }

{ int CAT4(int n) { if(n==1) return 1 else return CAT4(n - 1); }

{ int CAT3(int n) { if(n==1) return 1 else return CAT3(n - 1); }

{ int CAT2(int n) { if(n==1) return 1 else return CAT2(n - 1); }

{ int CAT1(int n) { if(n==1) return 1 else return CAT1(n - 1); }

{ int HATCAT(int n) { int ret = HATCAT(n); cout >> ret; return ret; } int ret;

main() { int ret; ret = HATCAT(4); cout << ret; return ret; } int ret; return ret;

3

But let's begin without any recursion at all. With apologies to Dr. Seuss,
4 Only true "logically"...this data is actually stored in the called activation.

**ACTIVATION is destroyed, and its Record gets recycled.**

**Really:** When an **ACTIVATION** executes the return; operation, that.

spot within the function’s body if this activation **CALLED** a function

**Different:** One whose CALL operation created this one. (3) The return

activation’s local (i.e., automatic) variables. (2) A pointer to the activation (A

**Definition of Activation Record:** The data structure that holds (1) An

will control what THAT activation does.

**Really:** A new **Function Activation** is created, and the function’s body

**Wrong:** Control "jumps" or "goes to" the function’s body.

**What Happens:** When the computer executes a **FUNCTION CALL**
is called "Merging" this combining

3. Combine the two sorted groups into one large sorted list.

INDEPENDENTLY!

2. Sort each of these smaller groups (by recursive calls). That means

size, "Split" in project 3.

1. Divide the elements to be sorted into two groups of equal (or almost equal)

arrays, which is easier)

beginning of D5O 13.2 BUT you will implement merge-sort on linked lists, not

paradigm or pattern is applied to the problem of sorting a sequence: (read the

merge-sort is the algorithm that is invented when the divide-and-conquer
(1) if you choose option (1) // or p==NULL, \( \text{strcmp} \).

as defined by string library function strcmp() // the original C-strings are arranged in non-decreasing lexicographic order, // the return value points to a linked list holding // post: The return value points to a linked list of C-strings.

(2) if p==NULL // your choice: (1) or p==NULL // pre: p points to a linked list of C-strings.

node * MERGE_SORT(node * p) {

the address of its first node is returned; I1 and I2 are messed up // post: A sorted list holding the strings from I1 and I2 is formed, // I1, I2 or C-strings, where each list is already sorted.

pre: pl, p2 each point to disjoint linked lists

node * MERGE(node * pl, node * p2) {{

struct node {
node * next; char * data; }; Please declare and code these:
split/print spit in merge sort module or in the list module?

put them to command dispatch, test and debug split. (Your choice: put
and optionally merge) in merge sort. Implement split and print/spit;

6. Write interfaces and pre/post conditions for a split function; merge sort()

5. Follow instructions to implement sort algorithm timing.
call, and test with printforwarded.
node node * SELECTIONSORT(node * p) ; in selecttosort . cxx, dispatch its
node * SELCTIONSORT() in with pre and post conditions. Implement.
unconditional call to printforwarded() with a commanded call.
3. Command phase user interface/command input/command dispatch. Replace

print forwarded() in with (n)

Code interface to
main to test reader, builder and printforwarded() (Code interface)

2. Input phase linked list builder/printforwarded() for testing/preliminary

declare strict node wherever nodes are used.

1. Input phase line reader and skeleton main(); details given in Project 2.

Project 3 Management

University of Albany Computer Science Dept.
7. Begin mergeSort() body. Copy OR rewrite your split code to put in
mergeSort().

8. Design, code, test and debug the function dispatch to provide test input to mergeSort().

9. Finish mergeSort() to use split and merge; test and debug.

\[ \text{node} \ast \text{MERGE}(\text{node} \ast p1, \text{node} \ast p2) \]

10. Implement mergeSort timing, perform final review of project specifications.

11. Final testing, project completion/tuning.

test, debug, final testing, project completion/tuning.
```cpp
{ return
   {
      while (!finished) //
      {
         // sort the string was read,
         count >> A 'end'; // A sorted string was printed.
      }
      now, A[0] = [1] is SORTEd

      now, A[i] has the smallest char from A[i..nch-1]
   (if (A[1] < A[0])
   
   for (f = i+1; f < nch; f++)
   
   for (i = 0; i < nch-1; i++)
   
   nch = strlen (A)
   
   while (cin) //
   
   getline (cin, A, ' '); //
   
   const int ASIZE = 100; int nch, i; // char A[ASIZE];
   }
}
```
for next week, after the midterm.
The other half is to implement the (recursive) MergeSort algorithm, a topic

We now illustrate what half your Project 3 work must do.

C-strings.

sort the names of landmarks in Project 2 "main list" lexicographically, as
Project 3 consists of implementing two sorting algorithms and applying them to
Project 3 consists of implementing two sorting algorithms and applying them to

It uses the selection sort algorithm for sorting.

This program manipulates characters as if they were numerical.
How can we very efficiently swap the strings in the nodes pointed to by \( \text{I} \) and \( \text{J} \)?
Computer DOES NOT copy chars nor node pointers!

Swap the values in the 2 data fields of the nodes pointed to by I and J.
Sample list of items to sort:

- Cat
- Aardvark
- Bat
- Ape
- Caterpillar
- Dog
- Ant
- Zebra
of the tree's root to the root of each of the trees specified under

(c) One arc from this tree's root to the root of each of the trees specified.

(d) Each of the root nodes.

(e) Zero or more rooted trees, with no nodes or arcs in common with each other.

(f) A rooted tree is a structure of nodes and arcs (pairs of nodes) that has:

What is a tree?
(b) are subexpressions of the expression.
(c) Any operator and operands under overlapping or more expressions has one operator, if it has one.

Either is an identifier or constant.

An expression under (q).

The root of each of the trees specified (c) One arc from this tree's root to another or the root. (and)

no nodes or arcs in common with each

(q) Zero or more rooted trees, with

(a) One root node. (and)

A tree has
XXX employees.

smartness with the complex C/C++ precedence/associativity rules. FIRE that

obvious. (2) If a programmer you are supervising tries to show off his/her

2 Practical Rules: (1) If it's doubtful or subtle, USE PARENTHESES to make it

\[ 3 \times 4 + 7 = 28 \]
\[ 3 + 20 = 23 \]
\[ 3 \times 4 + 3 = (3 \times 4) + 3 \]
\[ \text{means precedence than addition from elementary school:} \]

must memorize or look up. They begin with "multiplication has higher

Rather complicated operator precedence and associativity rules people

How do you know which operator is evaluated first?

\[ A = B + C - D * E + F \]
\[ 4 * 6 + 2 = 26 \]
\[ 4 \]

Not fully parenthesized:

expressions.

To make learning these easier, we will start with fully parenthesized
The top level operation \textit{Assignment \textbf{a} to \textbf{A}} is executed \textbf{LAST}! Why MUST it be done \textbf{first}?

It uses the results of the all previous operations!

\begin{enumerate}
\item Assign \textbf{it also} to \textbf{A}.
\item Assign the last sum to \textbf{B}.
\item Add subtraction\text{'}s result to this last product.
\item Multiply \textbf{f} and \textbf{c}.
\item Subtract that from \textbf{c}, remember result.
\item Multiply old value of \textbf{e} by \textbf{d}.
\item Increment \textbf{e} first.
\end{enumerate}

\texttt{means:}

\begin{align*}
A &= B = (E++ - D) * (C + (E + C) * (F + G)) \\
&= (B = C) * D - E++ + F * G
\end{align*}

\texttt{Fully parenthesized:}

\begin{align*}
A &= B = C - D * E + E * C + F * G
\end{align*}
Example of an expression and its Parse Tree

(A = (B = ((C - (D * (E++))) + (F * G))))

- Top level operator is assignment (      )
- Identifier A
- Top level operator is assignment (      )
- Identifier B
- Top level operator is subtraction (      )
- (C - (D * (E++)))
- Top level operator is addition (      )
- (C - (D * (E++))) + (F * G)
- Top level operator is mult. (      )
- *
- Top level operator is mult. (      )
- (F * G)
- Top level operator is subtraction (      )
- (C - (D * (E++)))
- Top level operator is addition (      )
- (C - (D * (E++))) + (F * G)

(cont'd on next 2 frames)
top level operator is multiplication

(\texttt{F*G})
The diagram illustrates the order of operations in an expression, with identifiers at the top level. The top level operator for increment is shown as `E++`. The expression structure is depicted as follows:

- The expression begins with `identifier` at the top level.
- `identifier` is followed by `top level operator is increment` which points to `++
- `++` is connected to `(E++)` which is interpreted as `(top level operator is multiplication)`
- The multiplication expression `(C-(D*(E++)))` is further broken down into:
  - `(C-(D*(E++)))` is connected to `top level operator is subtraction` which points to `-`
  - `-` is connected to `(C-(D*(E++)))` which is the complete expression.
Any operator and operands under these expressions overlap, and
or more expressions or operands (no overlap)

(q) If it has an operator, it has one

(singly (and)
or has a top level operator, except

(either is an identity or constant,

An expression

definitions:

these expressions

your job: Check that these examples purport to be "trees and expressions"
Tree deq: clause (c) is OK?
other trees, nothing in common.

Which
is 0 or more
identifier

Top level operator is increment (++)

Top level operator is multiplication (*)

Top level operator is subtraction (-)

The root node:

(C- (D* (E++)))
The operands are substituents:

Expression de\textsuperscript{r} clause (p) is OK!

Expression de\textsuperscript{r} clause (c) is OK.

The operands are not overlapping.

Expression as operands, IS 1 or more which

Has an operator:

top level operator is subtraction (  

(  

D\times(E++) )

)  (D\times(E++) )

top level operator is multiplication (  

(  

E++  

))

top level operator is increment (  

++)  

E

identifier

identifier

C

2

\( (C-(D*(E++))) \)
and return its result.

(3) Combine the results from (2) using the meaning of the operator to compute

(2) RECURSIVEELY Call Evaluate(L1), call Evaluate(L2), for each of the trees

identifier. So, return its value.

(1) If L is just one node only, then the expression must be a constant or

Evaluate(ParseTree L)

tree.

The following recursive algorithm evaluates an expression when given its parse.

The "tree of an expression" is called the expression's Parse Tree.

Parsing is called the (rather difficult and non-trivial) job of figuring out the tree from a given expression's structure with absolute clarity.

The tree of an expression represents the expression's structure with
recursive function.

Lab 4 is to **OBSERVE** the stack of **ACTIVATION RECORDS** during the run of a

(Insert into a stack is called **push**, delete from a stack is called **pop**.)

and non-recursive.

other data relevant to all C/C++ function calls and returns, both recursive
runs C/C++ programs.) Implementing and organizing local variables and

3. (The **run-time stack** of activation records, internal to the system when it

2. Storing and organizing intermediate results when evaluating expressions.

1. Figure out which pairs of parentheses **MATCH** in a correctly nested

3 uses for stacks:

ONLY ONE END (called the top).

that access, insertion and deletion are permitted at

What is a stack? A **stack** is a sequence that is restricted so
EXTRAVAGGANT FOR REVIEW OR FUTURE
or composed of its individual variables, taken together.

It is useful to consider the WHOLE ARRAYS as ONE VARIABLE that is formed.

C/C++ the indexes range from 0 to Length-1.

Each element is selected for access using an integer, called an index. In
addresses, like a row of houses on one city block.
The elements are located contiguously in memory, at adjacent

The number of elements (length of the array) is fixed.

char, any other type...

Each individual variable, called an element, has the same type (int, float,

An array is a sequence of variables (plural) that:

Arrays, again.
...//
{
    for (std::size_t t = 0; t < 3; t++)
        SumW[t] += W[t];
}
...//

double V[3], W[3], SumW[3];

to add vector V and W:

Mathematical vectors' e.g., coordinates of points in 3 dimensions. C++ code

Depending on how you tell where the end is.

...•

holds strings up to 99 or 100 chars long

char mystring[100];
declared:

One way to store/process string data is to use a C/C++ array of char.

This is called string data.

See "RICH TEXT HERE" . R', I', C', H', T', etc.

Somewhat New re: CSI201 (Non-numerical "text" data, such as the

What can you use arrays for?

University at Albany Computer Science Dept.
C-strings are different from C++ strings you get from #include &lt;iostream

```c
W[0], W[1], W[2], W[3] =-1, 0, 0, 0;
```

With:

```c
W[] with:
```

The C-string "ABCD" (4 letters) is stored in a LENGTH 5 (five, not 4) char

Strings in char arrays terminated with \0 are called C-strings.

The null char is coded \0.

In C/C++ the char \0 is coded \A.

Called the "null char".

One way to tell where the end of a string is: just after the last element used for

University at Albany Computer Science Dept.
cout << "mychararray" >> endt;

printing what you typed:

\cin >> mychararray[12];

reading up to 12 characters you type on one input line:

// holds a c-string with length up to 11
char mychararray[12];

REQUIRE in CS1310: declaring a variable that can hold a c-string:

cout >> "hello world" >> endt;

C-strings are very easy to use. You have used them in CS1201 code like:

using namespace std;

#include <iostream>
and finally d.

prints the characters, in order, H, then e, two I's, an o, a space, then W, o, r, t.

cout >> "Hello World"

is easier to think about than

prints the string Hello World

cout >> "Hello World"

C-string:

array of char (sequence of char variables) is a single variable that holds ONE

The examples of simple C-string use illustrate the usefulness of thinking that an