Recursion

Lecture 7

- In this lecture we study ***recursion***.
- Recursion is a programming technique in which procedures and functions ***call themselves***.
- We'll also be looking at the ***stack*** — a structure maintained by each program at run time. This will help us to understand recursive procedure call.

The Value of Recursion

- Recursion can be used to replace loops.
- Recursively defined data structures, like lists, are very well-suited to processing by recursive procedures and functions
- A recursive procedure is mathematically more elegant than one using loops.
- Sometimes procedures that would be tricky to write using a loop are straightforward using recursion.

A Recursive Factorial Function

- Here's a function that computes the factorial of a number N without using a loop.
- It checks whether N is smaller than 3. If so, the function just returns N.
- Otherwise, it computes the factorial of N – 1 and multiplies it by N.

```
int Factorial (int Number) {
    if (Number <= 2)
        return Number;
    else
        return (Number*Factorial(Number-1));
}
```


Recursive Evaluation

- This box shows the pattern of function calls used to evaluate Factorial(5) recursively.

```
To evaluate Factorial(5)
evaluate 5 * Factorial(4)
        To evaluate Factorial(4)
        evaluate 4 * Factorial(3)
                To evaluate Factorial(3)
                evaluate 3 * Factorial(2)
                        Factorial(2) is 2
                        Return 2
                Evaluate 3 * 2
                Return 6
        Evaluate 4 * 6
        Return 24
Evaluate 5 * 24
Return 120
```

Another Recursive Procedure

- The function below prints out the number in a list of numbers. Unlike the version of **printAll** in Lecture 5, slide 16, it doesn't use a loop.
- This is how it works.
  - It prints out the first number in the list.
  - Then it calls itself to print out the rest of the list.
  - Each time it calls itself, the list is a bit shorter.
  - Eventually we reach an empty list, and the whole process terminates.

```
void printAll (NodePtr hdList) {
    if (hdList != NULL) {
        cout << getFromList(hdList) << endl;
        printAll(hdList);
    }
}
```


The Stack

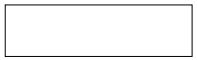
- To understand how recursion works at run time, we need to understand what happens when a function is called.
- Whenever a function is called, a block of memory is allocated to it in a run-time structure called the ***stack***.
- This block of memory will contain
  - the function's ***local variables***,
  - local copies of the function's ***call-by-value parameters***,
  - pointers to its ***call-by-reference parameters***, and
  - a ***return address***, in other words where in the program the function was called from. When the function finishes, the program will continue to execute from that point.

Function Call Example I

- Let's look at an example. Here's the main part of the program on slide 5.17.
- The initial stack is empty. (Or might contain the program's global variables, depending on implementation.)

```
int main()
{
    buildList (4,hdList);
    printAll (hdList);
    getchar();
}
```

Stack before call to **buildList**



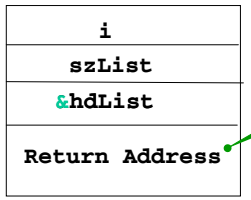
- What happens when the program calls **buildList**?

Function Call Example 2

- The **buildList** function has one local variable, and one call-by-value parameter one call-by-reference parameter.
- The local variable gets space on the stack, along with the value parameter and a pointer to the call-by-reference parameter.

```
void buildList (int      szList,
                NodePtr &hdList) {
    int i;
    for (i = 0; i < szList; i++)
        addToList(getItem(), hdList);
}
```

Stack after call to **buildList**

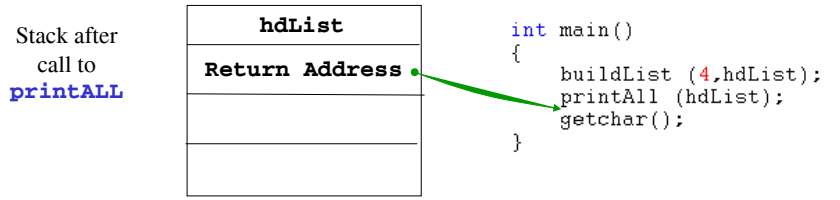



```
int main()
{
    buildList (4,hdList);
    printAll (hdList);
    getchar();
}
```

Recursive Function Calls I

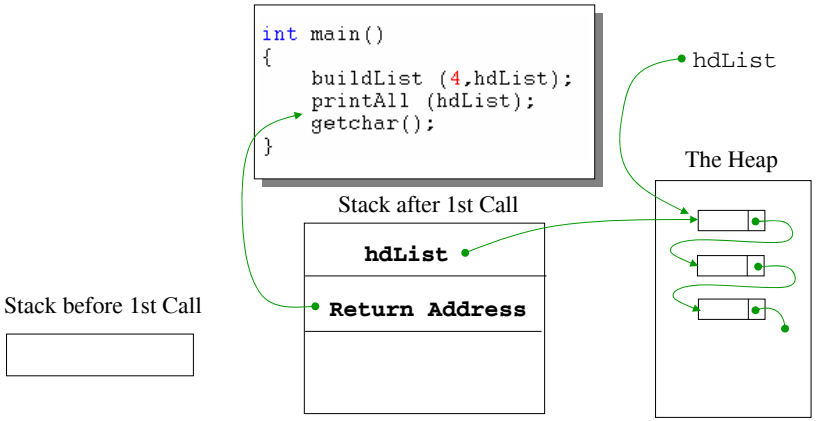
- Now let's look at the recursive calls to **printAll**.
- The **printAll** function has no local variables, but one call-by-value parameter.

```
void printAll (NodePtr hdList) {
    .... body of the function ...
}
```



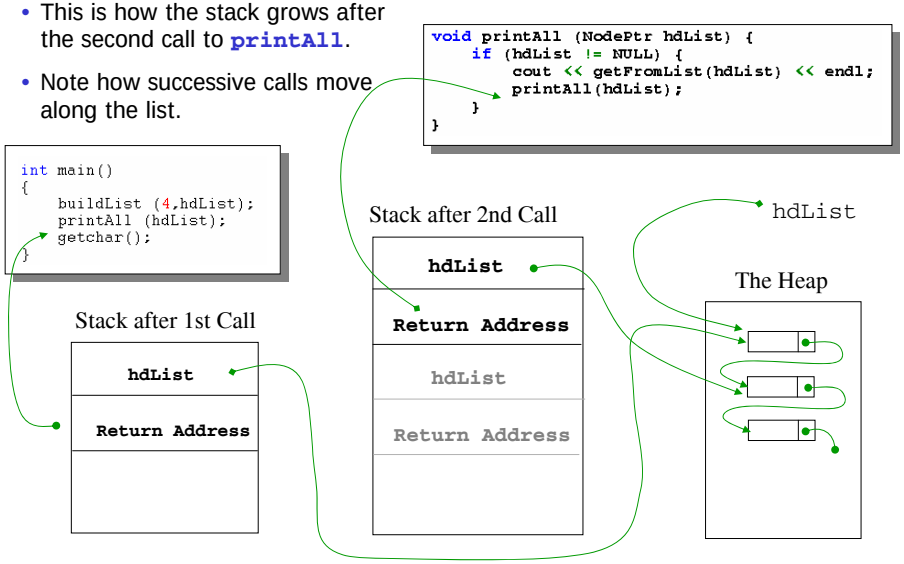
Recursive Function Calls II

- This is how the stack grows after the first call to **printAll**.



Recursive Function Calls III

- This is how the stack grows after the second call to **printAll**.
- Note how successive calls move along the list.



Recursive Function Calls IV

- Eventually, **printAll** is called with the empty list as its argument.
- Then there are no more recursive calls, and control is returned to the previous invocation of **printAll** , which in turn returns control to the one before, and so on until we are back in the main program again.
- The stack shrinks in the opposite order to that in which it grew. The *last* element to be put on the stack is taken off *first*. (A stack is a LIFO structure — Last In First Out.)
- If there are three elements in the list, this is how the execution looks.

1st call to **printAll**
Print 1st element of the list
2nd call to **printAll**
Print 2nd element of the list
3rd call to **printAll**
Print 3rd element of the list
Exit 3rd call to **printAll**
Exit 2nd call to **printAll**
Exit 1st call to **printAll**

Tricks with Recursion

- Recall that when we consider the program of ordered list, the number printed starts from head of the list to the tail of the list.
- It is possible to print the list from tail first with recursion.
- It is identical to the **printAll** program, except that the two lines inside the if statement are the other way around.
- Printing the list out backwards using a loop is much harder.

```
void printAll (NodePtr hdList) {
    if (hdList != NULL) {
        printAll(hdList->next);
        cout << getFromList(hdList) << endl;
    }
}
```


Recursion: Final Remarks

- The trick with recursion is to ensure that each recursive call gets closer to a ***base case***. In most of the examples we've looked at, the base case is the empty list, and the list gets shorter with each successive call.
- Recursion can ***always*** be used instead of a loop. (This is a mathematical fact.) In declarative programming languages, like Prolog, there are no loops. There is only recursion.
- Recursion is elegant and sometimes very handy, but it is marginally less efficient than a loop, because of the overhead associated with maintaining the stack.