Introduction to C++

Recursion

Topic #5
CS162 - Topic #5

• Lecture: Recursion
  – The Nature of Recursion
  – Tracing a Recursive Function
  – Work through Examples of Recursion
  – Problem solving with recursion
CS162 - Recursion

- Recursion is repetition (by self-reference)
  - it is caused when a function calls/invokes itself.
  - Such a process will repeat forever unless terminated by some control structure.
CS162 - Recursion

• So far, we have learned about control structures that allow C++ to iterate a set of statements a number of times.

• In addition to iteration, C++ can repeat an action by having a function call itself.
  – This is called recursion. In some cases it is more suitable than iteration.
While recursion is very powerful and will allow us to at times simply solve complex problems, it should not be used if iteration can be used to solve the problem in a maintainable way (i.e., if it isn’t too difficult to solve using iteration). So, think about the problem. Can loops do the trick instead of recursion?
CS162 - Recursion

- Why select iteration versus recursion?
  - Efficiency!
  - Every time we call a function a stack frame is pushed onto the program stack and a jump is made to the corresponding function
  - This is done in addition to evaluating a control structure (such as the conditional expression for an if statement) to determine when to stop the recursive calls.
  - With iteration all we need is to check the control structure (such as the conditional expression for the while, do-while, or for)
CS162 - Recursion

• Let's look at a very simple example;
  – in this case we can see that by using recursion we can make some difficult problems very trivial...
  – many of these problems would be very difficult to solve if you only were able to use iteration.
  – *trace through the following problem in class...showing how the stack frame works*
• What is the purpose of the following?

```c
void strange(void);
int main()
{
    cout << "Please enter a string" << endl;
    strange();
    cout << endl;
    return 0;
}

void strange(void) {
    char ch;
    cin.get(ch);
    if (!cin.eof() && ch != '\n') {
        strange();
        cout << ch;
    }
}
```
CS162 - Recursion

• This program writes the reverse of what was entered at the keyboard, no matter how many characters were entered!
  • Try to write an equally simple program just using the iterative statements we know about; it would be difficult to make it behave the same without limiting the number of characters that can be entered or using up a lot of memory with a huge array of characters!
  • Notice, with recursion, we didn't have to even use an array!!
CS162 - Recursion

• What happens to this “power” if we had swapped the cout statement with the recursive call in the previous example?
  • It would have simply read and echoed what was typed in.
  • Recursion would be overkill; iteration should be used instead.
CS162 - Recursion

- When a recursive call is encountered, execution of the current function is temporarily stopped.
- This is because the result of the recursive call must be known before it can proceed.
- So, it saves all of the information it needs in order to continue executing that function later (i.e., all current values of all local variables and the location where it stopped).
- Then, when the recursive call is completed, the computer returns and completes execution of the function.
CS162 - Recursion

- In order for your recursive calls to be useful, they must be designed so that your program will ultimately terminate.
- As with iteration or looping, there is danger of creating a recursive function that is an infinite loop!
- We need to be careful to prevent infinite repetition.
- Therefore, when designing a recursive function
  - one of the first steps should be to determine what the stopping condition should be
CS162 - Recursion

- The best way to do this is to use
  - an if statement to determine if a recursive call should be made -- depending on the value of some conditional expression.
- Eventually, every recursive set of calls should reach a point that does not require recursion (i.e., this will stop recursion).
- Recursion should not be used if it makes your algorithm harder to understand or if it results in excessive demands on storage or execution time.
Therefore, there are three requirements when using recursion:

• Every recursive function must contain a control structure that prevents further recursion when a certain state is reached.
• That state must be able to be reached each time you run the program.
• When that state is reached, the function must have completed its computation and (if the function returns a value) return the appropriate value for each recursive call.

Don’t forget to have the function “use” the returned value...if there is one!
CS162 - Recursion

• In class, walk through the following:

```c
int factorial(int n)
{
    if (n < 2)
        return 1;
    else
        return (n * factorial(n-1));
}
```
CS162 - Recursion

• In class, walk through the following:

```c
int factorial(int n)
{
    if (n < 2)
        return 1;
    else
        return (n * factorial(n-1));
}
```

• Compare and contrast with the iterative version (in the lecture notes). Which is better? Why?
CS162 - Recursion

• If you request nesting or recursion that goes beyond what your system can handle...you will get an error when you try to execute your program...such as "stack overflow".

• This simply means that you've tried to make too many function calls - recursively.

• If you get this error, one clue would be to look to see if you have infinite recursion.
  – This situation will cause you to exceed the size of your stack -- no matter how large your stack is!
CS162 - Examples of Recursion

• Two meaningful examples of recursion are the
  – towers of hanoi problem
  – binary search

• Let’s discuss each of these in class and examine:
  – the process they go thru
  – see how recursion helps solve the problem
  – look at the implementation details (of the binary search)
  – discuss the benefits and drawbacks of recursion for these algorithms
CS162 - Using Recursion

• Today we will walk through examples solving problems with recursion

• To get used to this process
  – we will select simple problems that in reality should be solved using iteration and not recursion
  – but, it should give you an understanding of how to design using recursion
  – which we will need to understand for CS163
First, let’s display the contents of a linear linked list, recursively

– obviously this is **should** be done iteratively!
– but, as an exercise determine what the stopping condition should be first:
  • when the head pointer is NULL
  – what should be done when this condition is reached? return
  – what should be done otherwise? display and call the function recursively
• If we were to do this iteratively:
  ```
  void display(node * head) {
    while (head) {
      cout << head->data->title << endl;
      head = head->next;
    }
  }
  ```

• Why is it ok in this case to change head?
• Look at the stopping condition
  – with recursion we will replace the while with an if....and replace the traversal with a function call
• If we were to do this recursively:

```cpp
void display(node * head) {
    if (head) {
        cout << head->data->title << endl;
        display(head->next);
    }
}
```

• Now, change this to display the list backwards (recursively)

• Discuss the code you’d need to do THAT recursively....
Next, let’s insert at the end of a linear linked list, recursively

– again this is **should** be done iteratively!
– but, as an exercise determine what the stopping condition should be first:
  • when the head pointer is NULL
– what should be done when this condition is reached? **allocate memory and save the data**
– what should be done otherwise? call the function recursively **with the next ptr**
If we were to do this iteratively:

```c
void append(node * & head, const video & d) {
    if (!head) {
        head = new node;
        head->data = ••• //save the data
        head->next = NULL;
    } else {
        node * current = head;
        while (current->next) {
            current = current->next;
        }
        current->next = new node;
        current = current->next;
        current->data = ••• //save the data
        current->next = NULL;
    }
}
```
If we were to do this recursively:

```cpp
void append(node * & head, const video & d) {
    if (!head) {
        head = new node;
        head->data = ••• //save the data
        head->next = NULL;
    } else
        append(head->next, d);
}
```

Notice this is much shorter (but less efficient)

Notice the stopping condition (!head)

Examine how the pass by reference can be used to implicitly connect up the nodes

Walk thru an example of invoking this function
This can also be done recursively by using the returned value (rather than call by reference):

```cpp
node * append(node * head, const video & d) {
    if (!head) {
        head = new node;
        head->data = ... //save the data
        head->next = NULL;
    } else
        head->next = append(head->next, d);
    return head;
}
```

Notice the function call must use the returned value

Here, we are explicitly connecting up the nodes

Walk thru an example of invoking this function
Next, let’s remove an item from a linear linked list, recursively
  – again this is should be done iteratively!
  – but, as an exercise determine what the stopping condition should be first:
    • when the head pointer is NULL
    • when a match (the item to be removed) is found
  – what should be done when this condition is reached? deallocate memory
  – what should be done otherwise? call the function recursively with the next ptr
If we were to do this recursively:

```c
int remove(node * & head, const video & d) {
    if (!head) return 0; //match not found!
    if (strcmp(head->data->title, d->title)==0) {
        delete [] head->data->title;
        delete head->data;
        delete head;
        head = NULL;
        return 1;
    } return remove(head->next, d);
}
```

Does this reconnect the nodes?

How does it handle the special cases of a) empty list, b) deleting the first item, c) deleting elsewhere
CS162 - More Examples

• Now in class, let’s design and implement the following **recursively**
  – count the number of items in a linear linked list
  – delete all nodes in a linear linked list
• **Why would recursion not** be the proper solution for push, pop, enqueue, dequeue?
What is the output for the following program fragment? called: f(5)

```cpp
int f(int n) {
    cout <<n <<endl;
    if (n == 0) return 4;
    else if (n == 1) return 2;
    else if (n == 2) return 3;
    n=f(n-2) * f(n-4);
    cout <<n <<endl;
    return n;
}
```
What is the output of the following program or write INFINITE if there are indefinite recursive calls?

called:
cout << watch(-7)

int watch(int n) {
    if (n > 0)
        return n;
    cout << n << endl;
    return watch(n+2)*2;
}