Recursion

CS A150 – C++ Programming I

Introduction to Recursion

- A function that "calls itself"
  - Said to be ***recursive***
  - In function definition, call to same function
- C++ allows recursion
  - As do most high-level languages
  - Can be a useful programming technique
  - Has limitations

Recursive Void Functions

- Divide and Conquer
  - Basic design technique
  - Break large task into subtasks
- Subtasks could be smaller versions of the original task
  - When they are → **recursion**

Recursive Void Function (cont.)

- Consider task:
  - Search list for a value
    - Subtask 1: search 1st half of list
    - Subtask 2: search 2nd half of list
  - Subtasks are smaller versions of original task!
  - When this occurs, recursive function can be used
    - Usually results in "elegant" solution

Example 1: Void Function

- Task:
  - Display digits of number vertically, one per line
  - Example call:

```
writeVertical(1234);
```

We would like to produce this output:

```
4
3
2
1
```

Example 1: Void Function (cont.)

- Using a **for** loop

```
int num = 1234;

for (int i = 0; i < 4; ++i)
{
      cout << num % 10 << endl;
      num /= 10;
}
```

- We can transform this **for** loop to a **recursive** function

Example 1: Void Function (cont.)

- Break problem into ***two*** cases
  - **Simple/base case:** `if (n < 10)`
    - Simply write number n to screen
  - **Recursive case:** `if (n >= 10)`
    - two subtasks:

      1- Ready to output last digit

      2 -Re-send all digits except last digit to the function

- Example: argument 1234:
  - 1^{st} subtask ready to display 4
  - 2^{nd} subtask re-sends 123 to the function

Example 1: Void Function (cont.)

```
void iterWriteVert(int num)
{
   int num = 1234;
   for (int i = 0; i < 4; ++i)
    {
      cout << num % 10 << endl;
      num /= 10;
    }
}
```

```
void recurWriteVert(int num)
{
   if (num < 10)
   {
     cout << num << endl;
   }
   else
   {
     cout << num % 10 << endl;
     recurWriteVert (num / 10);
   }
}
```

Example 1: Void Function

```
void recurWriteVert(int num)
{
    if (num < 10)                          //Base case
        cout << num << endl;
    else
    {
        cout << (n%10) << endl;
        recurWriteVert(n/10);              //Recursive step
     }
}
```

- Example call: `writeVertical(1234);`

```
→ cout << 4 << endl;
  recurWriteVert(123);
      → cout << 3 << endl;
        recurWriteVert(12);
            → cout << 2 << endl;
                  recurWriteVert(1);
                    → cout << 1 << endl;
```

Example 2: Void Function

- You can modify the function to get a different output

1
2
3
4

```
void recurWriteVert(int num)
{
    if (num < 10)                        //Base case
        cout << num << endl;
    else
    {
        recurWriteVert(n/10);            //Recursive step
        cout << (n%10) << endl;
    }
}
```

switch

Example 2: Void Function

```
void recurWriteVert(int num)
{
    if (num < 10)                           //Base case
         cout << num << endl;
    else
    {
         recurWriteVert(n/10);              //Recursive step
         cout << (n%10) << endl;
     }
}
```

- Example call: `writeVertical(1234);`

```
     → writeVertical(123);
            → writeVertical(12);
                   → writeVertical(1);
                         cout << 1 << endl;
                     cout << 2 << endl;
              cout << 3 << endl;
      cout << 4 << endl;
```

Recursion: A Closer Look

- Computer tracks recursive calls
  - Stops current function
  - Must know results of new recursive call before proceeding
  - Saves all information needed for current call to be used later
  - Proceeds with evaluation of new recursive call
  - When THAT call is complete, returns to "outer" computation

Infinite Recursion

- Base case MUST eventually be reached
- If it is not → **infinite recursion**
  - Recursive calls never ends!
- Recall **`recurWriteVert()`** example:
  - Base case happened when down to 1-digit number
  - That's when recursion stopped

Example 3: Infinite Recursion

- Consider alternate function definition:

```
void newRecurWriteVert(int n)
{
    newRecurWriteVert(n / 10);
    cout << n % 10 << endl;
}
```

- Seems "reasonable" enough, BUT
  - Missing "base case"!
  - Recursion never stops

Stacks for Recursion

- A **stack**
  - Specialized memory structure
  - Like stack of paper
    - Place new on top
    - Remove when needed from top
  - Called "**last-in/first-out**" (**LIFO**) memory structure
- Recursion uses stacks
  - Each recursive call placed on stack
  - When one completes, last call is removed from stack

Stack Overflow

- Size of **stack** is limited
  - Memory is **finite**
- Long chain of recursive calls continually adds to stack
  - All are added before base case causes removals
- If stack attempts to grow beyond limit:
  - **Stack overflow error**
- Infinite recursion *always* causes this

Recursion Versus Iteration

- Recursion not always "necessary"
- Not even allowed in some languages
- Any task accomplished with recursion can also be done without it
  - Non-recursive (called **iterative**) → using loops
- Recursive:
  - Runs slower, uses more storage
  - Elegant solution; less coding

PROJECT 1

- Write an iterative void function, **`iterDrawStars()`**, that has one parameter, a positive integer indicating the number of stars, and writes out the number of asterisks (*) to the screen all in one line
- Write the corresponding recursive function, **`recurDrawStar()`**

- Example:

```
iterDrawStars(6);    // function call
******               // output
recurDrawStars(6);   // function call
******               // output
```

Returning a Value

- Recursion is ***not*** limited to **void** functions
- Can **return value** of any type
- Same technique:
  - Base case
  - Recursive call

EXAMPLE 4: POWER

- Recall predefined function `pow()`:

```
result = power(2.0,3.0);
```

  - Returns 2 raised to the power of 3 (8.0)
  - Takes two arguments of type **`double`**
  - Returns a value of type **`double`**

Example 4: Power (cont.)

- Find x to the y (assume y is an integer >= 0)
- What is the base case?

$y = 0 \rightarrow x^0 = 1$

- How do we find the recursion function?

16	$\rightarrow$	$2^4 = 2^3 * 2$
8		$2^3 = 2^2 * 2$
4		$2^2 = 2^1 * 2$
2		$2^1 = 2^0 * 2$
1		$2^0 = 1$ (base case)

- So, we can re-write our previous definition:

```
power(n, exp) = 1  (base case)          if exp = 0
power(n, exp) = power(n, exp -1) * n    if n > 0
```

Function Definition - power()

```
int power(int x, int n)
{
        if (n > 0)
                return (power(x, n-1) * x);
        else
                return (1);
}
```

- Example calls:
  - power(2, 0);
  - power(2, 1);
  - power(2,3);

Recursive Design Check: power()

- Check `power()` against 3 properties:
  1. No infinite recursion:
     - 2nd argument decreases by 1 each call
     - Eventually must get to base case of 1
  2. Stopping case returns correct value:
     - `power(x,0)` is base case
     - Returns 1, which is correct for x^0
  3. Recursive calls correct:
     - For `n>1`, `power(x,n)` returns `power(x,n-1)*x`
     - Plug in values → check if correct

EXAMPLE: FACTORIAL

- Recall

```
4! = 4 * 3 * 2 * 1
0! = 1
```

- How can we find a recursive definition?

```
fact(n) = 1          if n = 0   (base case)
fact(n) = ?          if n > 0
```

Example: Factorial (cont.)

- Finding the recursive portion of the function

```
fact(n) = ?                  if n > 0
```

- Reason on a couple of examples:

```
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
```

- Look for a pattern:

```
fact(n) = n * fact(n - 1)
```

- Final result:

```
fact(n) = 1                  if n = 0 (base case)
fact(n) = n * fact(n-1)      if n > 0
```

Function Definition: factorial

```
int factorial(int n)
{
        if(n == 0)
                return 1;

        return (n * factorial(n - 1));
}
```

- Example call:
  - factorial (0);
  - factorial (1);
  - factorial (4);

Project 2

- Write a recursive function **`squares`** that has one integer parameter **`n`** and returns the sum of the square of the numbers 1 to **`n`**.

- Example: **`squares(3)`** returns 14, because

  $1^2 + 2^2 + 3^2$ is 14

28

Recursion (end)