Debugging

- It is highly unlikely that you will write code that will work on the first go

- Bugs or errors
  - Syntax
    - Fixable if you learn to read compiler error messages
  - Semantic
    - No easy fix
      - Use print statements to our advantage

Syntax Error

- Use the Syntax Highlighter to your advantage
  - Keywords are emphasized
  - Comments
  - {} matching

- Reading compiler Errors
  - GoTo line number option
  - Learn common syntax errors

//sum positive odd numbers upto n
for (i = 1; i <= n; i = i + 1){
    sum = sum + i;
    printf("%d : %d\n", i, sum);
}

Result for n = 3
1 :1
2 :3
3 :6

Remember to comment out print statement when you are done testing
Common Semantic Problem - I

- The very last thing a function must do is return something (unless void)

- Usually if you have a loop, then a return statement inside it may or may not work correctly
  - This because the code may not be reachable
  - Or you return to early
  - C gives warning but does not complain (compiler error)
    - Unlike Java which results in error if does not find a return statement as last statement.

Example

```c
#include <stdio.h>

int factorial(int n){
    int prod = 1;
    int i;
    for(i = n; i > 0; i--){
        prod = prod * i;
    }
    return prod;
}
```

- Does the program compile?
- If compiles, does it return the expected outcome?

Common Semantic Problem - II

- Variable initialization and comparison errors
- Variable is usually initialized to zero
  - Caveat to accumulate product, the variable is initialized to zero
- In a loop the iterative or loop variable initialization or end-statement test is incorrect
  - E.g. iterate through 10 times
    1. for(i = 0; i < 10; i++)
    2. for(i = 1; i < 10; i++)

Frame for function calls

- Whenever a function is called, some memory is set aside to contain information related to the call
  - Example: parameter values, current statement being executed etc
  - This is called the frame for the function call
    ```c
    int add ( int number1, int number2 ) {
        int sum = number1 + number2 ;
        return sum ;
    }
    ```
  - The frame is discarded when the function returns
    - This implies parameters and local variables (variables declared within method) are discarded
    - Hence, this explains the reasoning for scope rules for variables within functions.
**Call by Value**

- We call a function like this:
  ```
  result = add(3, 5);
  ```

- Actual parameter values are copied into formally defined parameters

```
int add(int number1, int number2) {
    int sum = number1 + number2;
    return sum;
}
```

**Functions calling other Functions**

```
int isLeapYear(int year) {
    if(year %4 == 0 && (year %100 != 0 || year %400 == 0))
        return TRUE;
    else
        return FALSE;
}
```

```void leapYearBet(int start, int end) {
    int x = start;
    printf("Leap years between %d and %d:\n", start, end);
    while(x <= end){
        if(isLeapYear(x)){
            printf("%d\n", x);
        }
        x++;
    }
}"
```

**Stack**

- Suppose a call to function \( m_1 \) is being executed, this causes a frame for the call to exist:
  ```
  e.g. leapYears(2000, 2008)
  ```

- Suppose the function \( m_1 \) while executing calls another function \( m_2 \) e.g. leapYears function calls isLeapYear(x):
  ```
  A frame for \( m_2 \) is created, & now at this point two frames exist
  ```

- If \( m_2 \) calls another method \( m_3 \), then a third frame is created:
  ```
  The frames are destroyed in the order in which they were created:
  ```

- \( m_3 \) is first completed and hence discarded, then \( m_2 \), and finally \( m_1 \)

- Follows first-in-last-out principle and the section of memory that allocates memory during function calls is known as **stack**
Need for recursive functions

Many things we model have recursive structures (where the subparts look like the whole):

- Mathematical formulas
  - Power: \( x^y = x \times x^{y-1} \)
  - Factorial: \( n! = n \times (n-1)! \)
  - Fibonacci series: \( \text{fb}(n) = \text{fb}(n-1) + \text{fb}(n-2) \)
- Fractal geometry
- Languages
  - Programming languages: An expression can contain expressions
  - A statement can contain other statements
  - HTML tags in a web page
  - XML (extensible markup language), for storing semi-structured data
- Hierarchical Entities (often stored as “trees”)
  - A “family ancestral tree”
  - An operating system’s folder/directory hierarchy
- Graphs
  - Social networks (Facebook)
  - Computer networks
  - Airline travel networks

Recursive function

- “Recursive” means “self referencing”
- Function that calls itself is known as *recursive* function
- Each time a function calls itself, a new frame for the method is created
- Frame is discarded when it returns from the call

Writing Recursive Function

- Handle the “base case”
  - Which is when a recursive call should not be made because the process should stop
- If not handled then we call function infinite and at some point we will have stack over flow problem
  - Indicated as “Segmentation Fault”
  - Segmentation fall is runtime/semantic error you get when you access a illegal memory location
- For other cases, make a recursive call
  - Which must make progress towards a goal (towards the base case)

Example of recursive function

```c
/* Recursive function that prints
 * num, num-1, num-2, ... 1.
 * Assumes num is positive.
 */
void printDownTo1(int num){
    if (num == 0) {// base case
        return;
    }
    printf("num is %d\n", num);
    printDownTo1(num - 1);
}
```
Recursive Example: sum numbers

/* A recursive function that returns
* the sum of the numbers from start to end
*/

int sum(int start, int end){
    if (start > end) { // base case
        return 0;
    }
    return start + sum(start + 1, end);
}