Introduction

Structured Programming is a problem-solving strategy and a programming methodology that includes the following two guidelines:

- The flow of control in a program should be as simple as possible.
- The construction of a program should embody top-down design.
Top-down Design

Top-down design (stepwise refinement, or divide and conquer) consists of repeatedly decomposing a problem into smaller problems.

- A program is constructed from smaller pieces (components, modules)
- Each piece is more manageable than the original program

Functions

- Programs combine user-defined functions with library functions
  - C standard library has a wide variety of functions, e.g. several math functions are included in math.h
- Invoking functions
  - Provide function name and arguments
  - Function performs operations or manipulations
  - Function returns results
- Function call analogy
  - Boss asks worker to complete task
  - Worker gets information, does task, returns result
  - Information hiding: boss does not know details

Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - #include <math.h>
- Format for calling functions
  - FunctionName( argument );
    - if multiple arguments, use comma-separated list
    - \( y = \text{sqrt}( 900.0 ) \);
      - Calls function sqrt, which returns the square root of its argument
- Arguments may be any r-value (constants, variables, or expressions)
Math Library Functions (Example)

- How to code square root of \((x_1 - x_2)^2 + (y_1 - y_2)^2\) by using math.h library functions?

\[
\begin{align*}
  a &= x_1 - x_2; \\
  b &= y_1 - y_2; \\
  c &= \text{pow}(a, 2) + \text{pow}(b, 2); \\
  d &= \text{sqrt}(c);
\end{align*}
\]

Functions

- We have already written/called some functions before.

```c
/* Welcome to BBM 101 */
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

- `main` is a function that must exist in every C program.
- `printf` is a library function which we have already used in our program.

Let's create and call our own functions now!

Function Definition

- Syntax

```c
type name (parameters) {
    variables;
    statements;
}
```

- `name` is the name of the function
- `type` is the type of the returned value by the function
  - `void` means the function returns nothing
  - Functions return `int` value if nothing is specified
- `parameters` specify the types and names of the parameters separated by comma

Function Returning a Value (Example)

- Let's define a function to compute the cube of a number:

```c
int cube ( int num ) {
    int result;
    result = num * num * num;
    return result;
}
```

- This function can be called as:

\[
n = \text{cube}(5);
\]
void Function (Example)

/* function definition */
void print_message(void)
{
    printf("A message for you: ");
    printf("Have a nice day!\n");
}

int main(void)
{
    /* function invocation */
    print_message();
    return 0;
}

Function Invocation

- A program is made up of one or more functions, one of them being main().
- When a program encounters a function, the function is called or invoked.
- After the function does its work, program control is passed back to the calling environment, where program execution continues.

The return Statement

- When a return statement is executed, the execution of the function is terminated and the program control is immediately passed back to the calling environment.
- If an expression follows the keyword return, the value of the expression is returned to the calling environment as well.
- A return statement can be one of the following two forms:
  return;
  return expression;

Examples:
  return;
  return 1.5;
  return result;
  return a+b-c;
  return x <= y ? x : y;

The return Statement (Example)

- Define a function to check if asked year is a leap year
  int IsLeapYear(int year){
    return ( ((year % 4 == 0) && (year % 100 != 0))
    || (year % 400 == 0 ) );
  }

- This function may be called as:
  if (IsLeapYear(2005))
    printf("29 days in February.\n");
  else
    printf("28 days in February.\n");
Example: Find minimum of two integers

```c
#include <stdio.h>
int min(int a, int b){
    if (a < b) return a;
    else return b;
}

int main (void){
    int j, k, m;
    printf("Input two integers: ");
    scanf("%d %d", &j, &k);
    m = min(j, k);
    printf("The minimum is \n\n" , m);
    return 0;
}
```

Input two integers: 11 3
The minimum is 3.

Function Parameters

- A function can have zero or more parameters.
- The *formal parameter list* in declaration header
  ```c
  int f (int x, double y, char c);
  ```
  Parameter variables and their types are declared here.
- The *actual parameter list* in function calling
  ```c
  value = f(age, 100*score, initial);
  ```
  Cannot be told what their types are from here.

Rules for Parameter Lists

- The number of parameters in the actual and formal parameter lists must be *consistent*.
- Parameter association is *positional*: the first actual parameter matches the first formal parameter, the second matches the second, and so on.
- *Actual* parameters and *formal* parameters must be of compatible *data types*.
- *Actual* parameters may be a variable, constant, any expression matching the type of the corresponding formal parameter.

Call-by-Value Invocation

- Each argument is evaluated, and its value is used locally in place of the corresponding formal parameter.
- If a variable is passed to a function, the stored value of that variable in the calling environment will not be changed.
- In C, all calls are call-by-value.
Function Call

- The type of a function-call expression is the same as the type function being called, and its value is the value returned by the function.
- Function calls can be embedded in other function calls.

```c
int sum = 0;
for (; n > 0; --n)
    sum += n;
```

```
t = cubesum(i);
j = cubesum(t);
```

is equivalent to

```
j = cubesum(cubesum(i));
```

Example: Find maximum of three integers

```c
#include <stdio.h>
int maximum(int a, int b, int c){
    int max = a;
    if (b > max)
        max = b;
    if (c > max)
        max = c;
    return max;
}
```

```c
int main (void){
    int j, k, l, m;
    printf("Input three integers: ");
    scanf("%d %d %d", &j, &k, &l);
    printf("\nThe maximum is %d.\n", maximum(j, k, l));
    return 0;
}
```

Function Call (Example)

```c
#include <stdio.h>
int compute_sum (int n){
    int sum = 0;
    for (; n > 0; --n)
        sum += n;
    printf("%d ", n);
    return sum;
}
```

```c
int main (void){
    int n = 3, sum;
    printf("%d ", n);
    sum = compute_sum(n);
    printf("%d ", n);
    printf("%d", sum);
    return 0;
}
```

Function Prototypes

- General form for a function prototype declaration:
  
  ```c
  return_type function_name (parameter-type-list)
  ```

- Used to validate functions
  - Prototype only needed if function definition comes after use in program

- The function with the prototype

  ```c
  int maximum( int, int, int );
  ```

  (Takes in 3 ints, returns an int)
Using Function Prototypes

```c
#include <stdio.h>
int max (int a, int b){
    ...
}
int min (int a, int b){
    ...
}
int main(void){
    min(x,y);
    max(u,v);
    ...
}
int max (int a, int b){
    ...
}
int min (int a, int b){
    ...
}
int main(void){
    min(x,y);
    max(u,v);
    ...
}
```

Block Structure and Variable Scope

```c
#include <stdio.h>
int total, count;
int main(int argc, const char * argv[]){
    total = count = 0;
    
    int count = 0;
    while (1) {
        if (count > 10)
            break;
        total += count;
        count++;
        printf("%d\n", count);
    }
    printf("%d\n", count);
    count++;
    printf("%d\n", count);
    return 0;
}
```

Today

- Functions
  - Definitions
  - Invocation
  - Parameter Lists
  - Return Values
  - Prototypes
- Variable Scopes
  - Block Structure
  - Global and Local Variables
  - Static Variables
- Recursion

External Variables

- Local variables can only be accessed in the function in which they are defined.
- If a variable is defined outside any function at the same level as function definitions, it is available to all the functions defined below in the same source file
  → external variable
- Global variables are external variables defined before any function definition
  - Their scope will be the whole program
Local Variables

```c
#include <stdio.h>

void func1 (void){
    int i = 5;
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
}

int main (void){
    int i = 5;
    printf("%d \n", i);
    func1();
    printf("%d \n", i);
    return 0;
}
```

Static Variables

- A variable is said to be **static** if it is allocated storage at the beginning of the program execution and the storage remains allocated until the program execution terminates.
- External variables are always static.
- Within a block, a variable can be specified to be static by using the keyword `static` before its type declaration:
  ```c
  static type variable-name;
  ```
- Variable declared static can be initialized only with constant expressions (if not, its default value is zero).

Static Variables (Example)

```c
#include <stdio.h>

void incr(void);

int main (void){
    int i;
    void incr(void);
    for (i=0; i<3; i++)
        incr();
    return 0;
}

void incr(void){
    static int static_i = 0;
    printf("static_i = %d\n", static_i);
}
```

Static Variables (Example-Initial Value)

```c
#include <stdio.h>

void put_stars(int n){
    static int static_n;
    int i;
    for (i=0; i<static_n; i++)
        printf(" ");
    for (i=0; i<n; i++)
        printf("*");
    printf("\n");
    static_n += n;
}

int main(void){
    put_stars(3);
    put_stars(2);
    put_stars(3);
    return 0;
}
```
Today

- Functions
  - Definitions
  - Invocation
  - Parameter Lists
  - Return Values
  - Prototypes

- Variable Scopes
  - Block Structure
  - Global and Local Variables
  - Static Variables

- Recursion

Recursion

- **Recursion** is the process whereby a construct operates on itself.

- In C, a function may **directly** or **indirectly** call itself in the course of execution.
  - **direct**: The call to a function occurs inside the function itself
  - **indirect**: A function calls another function, which in turn makes a call to the first one

- Recursion is a programming technique that naturally implements the divide-and-conquer problem solving methodology.

Recursive Function – *How does it Look like?*

Computes the factorial of a nonnegative integer.

```c
int fact(int n)
{
    if (n == 0)
        return (1);
    else
        return (n * fact(n-1));
}
```

The function fact calls itself.

We will see how it really works soon!

The Nature of (direct) Recursion

- One or more simple cases of the problem (called the stopping cases or base case) have a simple non-recursive solution.

- The other cases of the problem can be reduced (using recursion) to problems that are closer to stopping cases.

- Eventually the problem can be reduced to stopping cases only, which are relatively easy to solve.

  **In general:**
  
  if (stopping case)
      solve it
  else
      reduce the problem using recursion
Recursive Example - Palindrome

- A palindrome is a word, phrase, number or other sequence of units that can be read the same way in either direction.
  - e.g., radar, level, rotator, “Step on no pets”, “Ey Edip Adanada pide ye”, “1234321”, etc.

- Recursive Definition:
  - If the string has no elements, then it’s a palindrome.
  - If the string has only one element, then it’s a palindrome.
  - If the elements in the endpoints (the first and last elements) are the same, and the internal letters are a palindrome, then it’s a palindrome.

Four Criteria of a Recursive Solution

- A recursive function calls itself.
  - This action is what makes the solution recursive.
- Each recursive call solves an identical, but a smaller problem.
  - A recursive function solves a problem by solving another problem that is identical in nature but smaller in size.
- A test for the base case enables the recursive calls to stop.
  - There must be a case of the problem (known as base case or stopping case) that is handled differently from the other cases (without recursively calling itself.)
  - In the base case, the recursive calls stop and the problem is solved directly.
- Eventually, one of the smaller problems must be the base case.
  - The manner in which the size of the problem diminishes ensures that the base case is eventually is reached.

Factorial Function – Iterative Definition

\[
\begin{align*}
n! &= n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \quad \text{for any integer } n > 0 \\
0! &= 1
\end{align*}
\]

Iterative Definition in C:

```c
fval = 1;
for (i = n; i >= 1; i--)
    fval = fval * i;
```

Factorial Function – Recursive Definition

- To define \( n! \) recursively, \( n! \) must be defined in terms of the factorial of a smaller number.
- Observation (problem size is reduced):
  \[ n! = n \times (n-1)! \]
- Base case \( \Rightarrow 0! = 1 \)
- We can reach the base case, by subtracting 1 from \( n \) if \( n \) is a positive integer.

**Recursive Definition:**

\[
\begin{align*}
n! &= 1 \quad \text{if } n = 0 \\
n! &= n \times (n-1)! \quad \text{if } n > 0
\end{align*}
\]
Revisiting the Recursive Factorial Function Definition in C

```c
/* Computes the factorial of a nonnegative integer. 
   Precondition: n must be greater than or equal to 0. 
   Postcondition: Returns the factorial of n; n is unchanged. */
int fact(int n)
{
    if (n == 0)
        return (1);
    else
        return (n * fact(n-1));
}
```

This `fact` function satisfies the four criteria of a recursive solution.

How does it Compute?

```c
printf("%d", fact(3));
```

Tracing a Recursive Function

- A stack is used to keep track of function calls.
- Whenever a new function is called
  - For each function call, an activation record (AR) is created on the stack.
  - AR consists of the function's parameters and local variables are pushed onto the stack along with the memory address of the calling statement (return point).
- To trace a recursive function, the box method can be used.
  - The box method is a systematic way to trace the actions of a recursive function.
  - The box method illustrates how compilers implement recursion.
  - Each box in the box method roughly corresponds to an activation record.

The Box Method

- Label each recursive call in the body of the recursive function.
  - These labels help us to keep track of the correct place to which we must return after a function call completes.
  - After each recursive call, we return to the labeled location, and substitute that recursive call with returned value.
The Box Method (cont’d.)

- Every time a function is called, a new box is created to represent its local environment.
- Each box contains:
  - The values of the arguments
  - The function’s local variables
  - A placeholder for the value returned from each recursive call from the current box (label in the previous step).
  - The value of the function itself.

```
 n = 3
 A: fact(n-1) = ?
 return ?
```

The Box Method (cont’d.)

- Draw an arrow from the statement that initiates the recursive process to the first box.
  - Then draw an arrow to a new box created after a recursive call, put a label on that arrow.

```
printf("%d", fact(3));
```

Box Trace of `fact(3)`

- The initial call is made, and the function `fact` begins execution.

```
 n = 3
 A: fact(n-1)=? 
 return ?
```

- At point A, a recursive call is made, and the new invocation of `fact` begins execution.

```
 n = 2
 A: fact(n-1)=? 
 return ?
```

- At point A, a recursive call is made, and the new invocation of `fact` begins execution.
Box Trace of \textit{fact(3)} (cont’d.)

- At point A, a recursive call is made, and the new invocation of \textit{fact} begins execution.
- This is the base case, so this invocation of \textit{fact} completes.
- The function value is returned to the calling box, which continues the execution.
- The current invocation of \textit{fact} completes.

The value 6 is returned to the initial call.

Example: Find the reverse of an input string

*/ reads n characters and prints them in reverse order. */
void reverse(int n){
  char next;  
  if (n == 1) {
    scanf("%c", &next); /* stopping case */
    printf("%c", next);
  } else {
    scanf("%c", &next);
    reverse(n-1);
    printf("%c", next);
  }
  return;
}

int main(){
  printf("Enter a string: ");
  reverse(3);
  printf("\n");
}
Example: Fibonacci Sequence

- It is the sequence of integers:

\[
\begin{align*}
0 & \quad 1 \\
1 & \quad 1 \\
2 & \quad 2 \\
3 & \quad 3 \\
5 & \quad 5 \\
8 & \quad 8 \\
\ldots & \\
\end{align*}
\]

- Each element in this sequence is the sum of the two preceding elements.

- The specification of the terms in the Fibonacci sequence:

\[
f_n = \begin{cases} 
n & \text{if } n \text{ is } 0 \text{ or } 1 \\
& t_{n-1} + t_{n-2} & \text{otherwise}
\end{cases}
\]

Example: Fibonacci Sequence

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

- This is an example of non-linear recursion. Because total number of recursive calls grows exponentially.

- \(fib(n-1)\) expression must be evaluated completely before its value can be added to the expression \(fib(n-2)\) which must also be evaluated completely.

- Recursion tree is useful in tracing the values of variables during non-linear recursion.

Example: Fibonacci Sequence (Iterative Version)

```c
int Fib(int n)
{
    int Prev1, Prev2, Temp, j;
    if (n==0 || n== 1)
        return n;
    else {
        Prev1=0;
        Prev2 = 1;
        for (j=1; j <= n; j++){
            Temp = Prev1 + Prev2;
            Prev2 = Prev1;
            Prev1 = Temp;
        }
        return Prev1;
    }
}
```
Recursion vs. Iteration

- In general, an iterative version of a program will execute more efficiently in terms of time and space than a recursive version. This is because the overhead involved in entering and exiting a function is avoided in iterative version.

- However, a recursive solution can be sometimes the most natural and logical way of solving a problem. Conflict: machine efficiency versus programmer efficiency.

- It is always true that recursion can be replaced with iteration and a stack (and vice versa).