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, also referred to as stepwise refinement, or divide and conquer, consists of repeatedly decomposing a problem into smaller problems. In other words:

- Construct a program from smaller pieces or components
  - These smaller pieces are called modules
- Each piece more manageable than the original program

Program Modules in C

- Functions
  - Modules in C
  - Programs combine user-defined functions with library functions
    - C standard library has a wide variety of functions
- Function calls
  - Invoking functions
    - Provide function name and arguments (data)
    - 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 \text{sqrt} which returns the square root of its argument
    - All math functions return data type double
  - Arguments may be constants, variables, or expressions

Available Mathematical functions

<table>
<thead>
<tr>
<th>Function Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int abs(int num)</td>
<td>Returns the absolute value of an integer element.</td>
</tr>
<tr>
<td>double fabs(double num)</td>
<td>Returns the absolute value of a double precision element.</td>
</tr>
<tr>
<td>double pow(double x, double y)</td>
<td>Returns ( x ) raised to the power of ( y ).</td>
</tr>
<tr>
<td>int rand(void)</td>
<td>Returns a random number</td>
</tr>
<tr>
<td>double sin(double angle)</td>
<td>Returns the sine of an angle angle should be in radians.</td>
</tr>
<tr>
<td>double cos(double angle)</td>
<td>Returns the cosine of an angle angle should be in radians.</td>
</tr>
<tr>
<td>double sqrt(double num)</td>
<td>Returns the square root.</td>
</tr>
</tbody>
</table>

Using Library Functions

- Calculate the square root of \((x_1 - x_2)^2 + (y_1 - y_2)^2\)
  - \( a = x_1 - x_2; \)
  - \( b = y_1 - y_2; \)
  - \( c = \text{pow}(a, 2) + \text{pow}(b, 2); \)
  - \( d = \text{sqrt}(d); \)
  - OR just:
    - \( d = \text{sqrt}(\text{pow}(x_1-x_2, 2) + \text{pow}(y_1-y_2, 2)); \)
- What is the value of:
  - \( \text{sqrt}(\text{floor}(\text{fabs}(-16.8))) \)

Functions

- We have already written our own functions and used library functions:
  - \text{main} is a function that must exist in every C program.
  - \text{printf}, \text{scanf} are library functions which we have already used in our programs.
- We need to do two things with functions:
  - Create Functions
  - Call Functions (Function invocation)
Function Definition

A function definition has the following form:

```
return_type function name (formal parameter list) {
    declarations
    statements
}
```

- **return_type** - the type of value returned by the function
- **void** – indicates that the function returns nothing.
- **function name** – any valid identifier
- **formal parameter list** – comma separated list, describes the number and types of the arguments that get passed into the function when its invoked.

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:

  ```c
  n = cube(5);
  ```

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.

```
#include <stdio.h>

void prn_message(void); /* function prototype */

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

void prn_message(void) /* function definition */ {
    printf("A message for you: ");
    printf("Have a nice day!\n");
}
```
#include <stdio.h>

void print_message (int k); /* function prototype */

int main (void)
{
    int n;
    printf("There is a message for you.
\nHow many times do you want to see it? ");
    scanf("%d", &n);
    print_message(n);
    return 0;
}

void print_message (int k) /* function definition */
{
    int i;
    printf("\nHere is the message.\n\n");
    for (i=0; i < k; ++i)
        printf("Have a nice day!\n");
}

/* An example demonstrating local variables */
#include <stdio.h>

void func1 (void);

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

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

The **return** statement

- When a return statement is executed, 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 has one of the following two forms:
  
  ```c
  return;
  return expression;
  ```

Examples

```c
return;
return 77;
return ++a;
return (a+b+c);
```
#include <stdio.h>

int min (int a, int 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 %d. 
", m);
    return 0;
}

int min (int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}

Parameters

- A function can have zero or more parameters.
- In declaration header:
  ```
  int f (int x, double y, char c);
  ```
  the formal parameter list
  (parameter variables and their types are declared here)
- In function calling:
  ```
  value = f(age, score, initial);
  ```
  actual parameter list (cannot tell what their type 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

Invocation and Call-by-Value

- 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 Prototypes

- **Function prototype**
  - Function name
  - Parameters – what the function takes in
  - Return type – data type function returns (default int)

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

- **The function with the prototype**
  - Takes in int
  - Returns int

---

```c
#include <stdio.h>
int compute_sum(int n);

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

```c
int compute_sum(int n)
{
    int sum;
    sum = 0;
    for (; n > 0; --n)
    {
        printf("%d\n", n);
        sum += n;
    }
    return sum;
}
```

---

```c
#include <stdio.h>
int maximum( int, int, int );   /* function prototype */

int main()
{
    int a, b, c;
    printf("Enter three integers: ");
    scanf("%d%d%d", &a, &b, &c);
    printf("Maximum is: %d\n", maximum( a, b, c ) );
    return 0;
}
```

```c
int maximum( int x, int y, int z )
{
    int max = x;
    if ( y > max )
    max = y;
    if ( z > max )
    max = z;
    return max;
}
```

**Input:**
Enter three integers: 22 85 17

**Output:**
Maximum is: 85

---

### Alternative styles for function definition order

```c
#include <stdio.h>
int max(int,int);
int min(int,int);

int main(void)
{...
    min(x,y);
    max(u,v);
    ...
    return max;
}
```

```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);
    ...
}
```

---

**Note:**

- Function prototypes are used to validate functions.
- They are only needed if the function definition comes after use in the program.
- The function with the prototype:
  - Takes in int parameters.
  - Returns an int type return value.
Correct the errors in the following program segments

1. int g (void) {
   printf("Inside function g\n");
   int h(void) {
      printf("Inside function h\n");
   }
}

2. int sum(int x, int y) {
   int result;
   result = x + y;
}

Correct the errors in the following program segments

3. void f (float a); {
   float a;
   printf("%.f", a);
}

4. void product (void) {
   int a, b, c, result;
   printf("Enter 3 integers: ");
   scanf("%d %d %d", &a, &b, &c);
   result = a * b * c;
   printf("Result is %d\n", result);
   return result;
}