CPS1011 Programming Principles (in C) Assignment 23/24 Documentation

Miguel Baldacchino

Table of Contents

References - page 52

Task 1 Algorithm implementation (Secant and Newton-Raphson Methods)

Introduction to Task:

This section of documentation outlines the Secant and Newton-Raphson Methodologies and Implementation as part of this assignment. Algorithms were designed in such a way to solve single-valued polynomial functions with degree of most 5, defined via user input. This section will delve through all my file organisation, functions, source code and assumptions and validation required.

Task Definition:

(a) Research and implement the Secant root-finding method. It is an iterative algorithm which computes the successive root approximation x_n as follows:$x_n = x_{n-1} - f(x_{n-1})(\frac{x_{n-1}-x_{n-2}}{f(x_{n-1})-f(x_{n-2})})$.

[15 marks]

(b) Research and implement the Newton-Raphson root-finding method. It is also an iterative algorithm which computes the successive root approximation x_n as follows:$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$.

[15 marks]

For both the above tasks, (a) and (b), you are required to implement each algorithm in its own C function, and only for polynomial single-valued functions $f(x)$, where x is a scalar and $f(x)$ is at most degree 5. The polynomial function must be passed as an argument in each case, parsed from user input and transformed into a form that each C function can process.

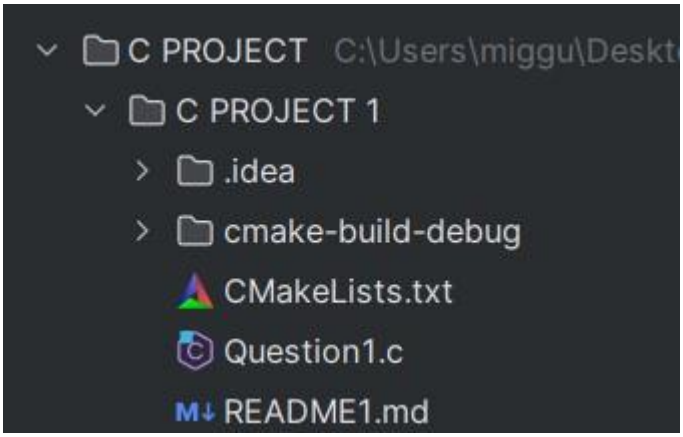
(c) Write a program that presents the end-user with a command-line menu, repeatedly asks the user to execute the above functions, and displays their output accordingly, or else to quit. Proper validation of user input is required.

[15 marks]

File Organization:

The structure of this Task is as follows:

C PROJECT -> C PROJECT 1



Question1.c tackles all the requirements of Task 1, including all Newton-Raphson, Secant Methods and Menu.

A CMakeLists.txt contains all the code necessary for Question1.c to be runnable.

A README1.md to briefly explain the code.

Functions Overview:

User Interaction Functions:

1. clearBuffer():
   Clearing the buffer, this ensures a clean state for future user inputs.

2. *getCoefficients():
   Takes user input for polynomial coefficients and stores them in a global pointer variable ‘coeff’

3. displayPolynomial():
   Displays entered polynomial to user for confirmation

4. displayDerivative():
   Displays calculated derivative of entered polynomial

5. getInitialGuess():
   Takes user input for the initial guess of a particular order in the algorithm

6. getTolerance():
   Takes user input for the tolerance required by user in the algorithm

7. getMaxIterations():
   Takes user input for the maximum number of iterations required by user in the algorithm

Polynomial Functions:

1. polynomial(x):
   Computes the value of the polynomial at a given point

2. derivative(x):
   Computes the value of the derivative of the polynomial at a given point **Root-Finding**

Methods and Implementation:

1. newtonRaphson():
   Implements Newton-Raphson root-finding method

2. secantMethod():
   Implements the Secant root-finding method

Menu and Main Functionality:

1. showMenu():
   Displays the root-finding methods menu

2. main():
   Main program execution, handles user input and method selection

Source Code Explanation:

Polynomial Input and Display:

To create the polynomial, the coefficients for said polynomials is to be defined by user, which is where *getCoefficients() comes into play. It prompts the user to input the coefficients of a polynomial of degree at most 5 and performs necessary input validation. The entered coefficients are stored in a dynamically allocated array (coeff), which is a global variable.

```c
double *getCoefficients() { // gets coefficients from user
    coeff = (double *) malloc( Size: 6 * sizeof(double)); // Memory allocated for coefficients

    if (coeff == NULL) { // checks successful memory allocation
        printf( format: "Error: Memory allocation has failed\n");
        return NULL; // indicates memory allocation failure
    }

    while (1) {
        printf( format: "\nAx^5 + Bx^4 + Cx^3 + D"
               "x^2 + Ex + F\n");

        printf( format: "Enter coefficients of polynomial (A B C D E F)\n");
        if (scanf( format: "%lf %lf %lf %lf %lf %lf",
                   &coeff[5], &coeff[4], &coeff[3], &coeff[2], &coeff[1], &coeff[0]) == 6) {

            if (coeff[0] == 0 && coeff[1] == 0 && coeff[2] == 0 && coeff[3] == 0 && coeff[4] == 0 && coeff[5] == 0) {
                printf( format: "All coefficients are zero. Root is trivially 0.\n");
                free( Memory: coeff);
                return NULL;
            }

            displayPolynomial();

            break;

        } else {
            printf( format: "Error: Invalid input. Please provide coefficients in the correct format.");
            clearBuffer();
        }
    }

    return coeff;
}
```

Breakdown:

Function Signature:

```c
double *getCoefficients()
```

The function returns a pointer to an array of double-precision floating-point numbers, ie. the coefficients of the polynomial

Memory Allocation

```c
coeff = (double *) malloc( Size: 6 * sizeof(double));
```

Allocates memory to store six double-precision floating-point numbers, corresponding to coefficients of polynomial (of degree most 5, hence the need of 6 coeff).

```c
if (coeff == NULL) { // checks successful memory allocation
    printf( format: "Error: Memory allocation has failed\n");
    return NULL; // indicates memory allocation failure
}
```

Checks successful memory allocation, with return NULL if allocation fails

User Input Loop

```
while (1) {
```

This initiates an indefinite loop for continuous user input until the valid coefficients required by function are to be provided

User prompt

```
printf( format: "\nAx^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F\n");
printf( format: "Enter coefficients of polynomial (A B C D E F)\n");
```

Displays a prompt for user to enter coefficients in the format `A B C D E F`

Input Validation

```
if (scanf( format: "%lf %lf %lf %lf %lf %lf",
          &coeff[5], &coeff[4], &coeff[3], &coeff[2], &coeff[1], &coeff[0]) == 6) {
```

Scanf is used to read the six values, and if input is successfully parsed into 6 values, loop breaks and coefficients are displayed via displayPolynomial() call.

Zero Coefficient Check

```
if (coeff[0] == 0 && coeff[1] == 0 && coeff[2] == 0 && coeff[3] == 0 && coeff[4] == 0 && coeff[5] == 0) {
    printf( format: "All coefficients are zero. Root is trivially 0.\n");
    free( Memory: coeff);
    return NULL;
}
```

Checks if all coefficients are zero, which if the case returns `NULL` indicating trivial zero root, but not before freeing previously allocated memory in coeff.

Error Handling

```
else {
  printf( format: "Error: Invalid input. Please provide coefficients in the correct format.");
  clearBuffer();
```

If invalid input is entered, an error message is displayed and buffer is cleared for the next attempt (due to the previously mentioned while loop)

clearBuffer(), displayPolynomial() and displayDerivative()

```c
void clearBuffer() { // clears buffer
    while (getchar() != '\n');
}

void displayPolynomial() { // Displays entered polynomial
    printf( format: "Your polynomial is:\n %.3fx^5 + %.3fx^4 + %.3fx^3 + %.3fx^2 + %.3fx + %.3f\n",
           coeff[5], coeff[4], coeff[3], coeff[2], coeff[1], coeff[0]);
}

void displayDerivative() { // Displays entered derivative of polynomial0
    printf( format: "\nThe derivative is:\n %.3fx^4 + %.3fx^3 + %.3fx^2 + %.3fx + %.3f\n",
           5 * coeff[5], 4 * coeff[4], 3 * coeff[3], 2 * coeff[2], coeff[1]);
}
```

For the clearBuffer(), a while loop is used to read characters until a \n character is found, effectively clearing any remaining characters in input buffer

For the displayPolynomial(), a printf is used to display the polynomial in the format `Ax^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F` and coefficients fetched from global variable coeff

For the displayDerivative, a similar approach is used. Derivative is displayed in format `Ax^4 + Bx^3 + Cx^2 + Dx + E` and coeff derived from global variable coeff.

polynomial(x) and derivative(x):

```c
double polynomial(double x) { // Creates polynomial
    return (coeff[5] * pow(x,  Y: 5)) + (coeff[4] * pow(x,  Y: 4)) + (coeff[3] * pow(x,  Y: 3)) + (coeff[2] * pow(x,  Y: 2)) +
           (coeff[1] * x) + (coeff[0]);

}

double derivative(double x) { // Creates derivative of polynomial
    return (5 * coeff[5] * pow(x,  Y: 4)) + (4 * coeff[4] * pow(x,  Y: 3)) + (3 * coeff[3] * pow(x,  Y: 2)) + (2 * coeff[2] * x) +
           (coeff[1]);

}
```

For the polynomial(x), all this is doing is computing the value of the polynomial given point x, utilising global variable coeff and using the same format as displayPolynomial().

For the derivative(x), similarly to polynomial(x), its computing value of derivative given point x using global variable coeff with same format as displayDerivative() getInitialGuess()

```c
double getInitialGuess(int order) {
    double guess;

    printf( format: "\nEnter the initial guess for x%d:\n", order);
    while (scanf( format: "%lf", &guess) != 1) {
        clearBuffer();
        printf( format: "Error: Invalid input. Please enter a valid numerical value.\n");
        printf( format: "Enter the initial guess for x%d:\n", order);
    }
    clearBuffer();

    return guess;
}
```

This function is used in both root-finding methods. This particular function takes order as parameter, declares variable guess and prints to user for initial guess. A while loop loops through if the initial guess was not valid, and keeps prompting for input until it is entered successfully. Buffer is cleared each time within the loop and another time after it exits, then returns a guess. getTolerance():

```c
double getTolerance() {
    double tolerance;

    printf( format: "\nEnter the tolerable error for convergence (e.g., 0.00001):\n");
    while (scanf( format: "%lf", &tolerance) != 1 || tolerance <= 0) {
        clearBuffer();
        printf( format: "Error: Invalid input. Please enter a valid positive numerical value.\n");
        printf( format: "Enter the tolerable error for convergence:\n");
    }
    clearBuffer();

    return tolerance;
}
```

Similarly to the previous function for initial guess, this prompts user for input of tolerance, and if tolerance is not valid (eg. less than zero), it prompts the user again for input whilst cleaning the buffer with each while loop. When exited, the buffer is cleared again and tolerance is returned.
getMaxIterations():

```c
int getMaxIterations() {
    int maxItr;

    printf( format: "\nEnter the maximum number of iterations:\n");
    while (scanf( format: "%d", &maxItr) != 1 || maxItr <= 0) {
        clearBuffer();
        printf( format: "Error: Invalid input. Please enter a valid positive integer value.\n");
        printf( format: "Enter the maximum number of iterations:\n");
    }
    clearBuffer();

    return maxItr;
}
```

Again, similarly to the previous two functions, maximum iterations is requested via user input, if it does not meet the while loop parameters (e.g. equal to or less than zero), user is prompted again for input and the

buffer is cleared with each cycle of loop. When exited, the buffer is cleared one more time and maxItr is returned.

Root-Finding Methods Implementation Newton-Raphson Method:

```c
double newtonRaphson() { // Newton-Raphson method
    clearBuffer();
    printf( format: "Newton-Raphson Method\n");
    printf( format: "_____________________\n");
    double x0, x1, tolerance; // x0 -> Initial Guess | x1 -> Calculated Root
    int i = 1, maxItr;

    coeff = getCoefficients(); // Get coefficients from user

    if (coeff == NULL) {
        printf( format: "Error: Coefficients allocation failed\n");
        return 0;
    }

    clearBuffer();

    displayDerivative();

    x0 = getInitialGuess( order: 0);
    tolerance = getTolerance();
    maxItr = getMaxIterations();

    if (maxItr < 1) {
        printf( format: "\nError: Iterations must be larger than 0\n");
        return 0;
    }
    do {
        if (derivative( x: x0) == 0) {
            printf( format: "Error: Mathematical Error, Denominator equal to zero");
            break;
        }



        x1 = x0 - (polynomial( x: x0) / derivative( x: x0));
        printf( format: "At iteration %d: x = %f\n", i, x1);
        // Check if the absolute difference is less than the tolerance
        if (fabs( X: polynomial( x: x1)) < tolerance) {
            printf( format: "\nConverged to the desired tolerance.\n");
            break;
        }
        x0 = x1;
        i++;
    } while (i <= maxItr);

    if (i <= maxItr && derivative( x: x0) != 0) {
        printf( format: "\nRoot found successfully.\n");
    } else {
        printf( format: "\nRoot not found within the specified tolerance or maximum iterations.\n");
    }
    free( Memory: coeff);
    return x1;
}
```

The method starts off with clearing the buffer, for safe measure. A display for the method is displayed using printf ‘s and a number of variables are declared, namely x0 being our Initial
Guess and x1 being our Calculated Ro

```
coeff = getCoefficients(); // Get coefficients from user

if (coeff == NULL) {
    printf( format: "Error: Coefficients allocation failed\n");
    return 0;
}

clearBuffer();
```

The previously explained getCoefficients() function is called and stored in a variable coeff. Validation is set into place to make sure a value is returned to coeff, otherwise an error is displayed and the function is stopped.

Buffer is cleared once again.

Derivative is displayed using the previously mentioned displayDerivative().

```
x0 = getInitialGuess( order: 0);
tolerance = getTolerance();
maxItr = getMaxIterations();
```

Functions for the Initial Guess, Tolerance and Max Iterations are all being called and stored into variables. These 3 items have been made into their own functions for better readability of the code, since it is later used in the Secant Method as well.

```
if (maxItr < 1) {
    printf( format: "\nError: Iterations must be larger than 0\n");
    return 0;
}
```

An error for Iterations less than zero is displayed and function is returned.

A do-while loop is initiated to perform the various iterations until convergence/number of iterations is reached.

```
if (derivative( x: x0) == 0) {
    printf( format: "Error: Mathematical Error, Denominator equal to zero");
    break;
}
```

Validation for denominators equal to zero in place, returning error and breaking do-while.

```
x1 = x0 - (polynomial( x: x0) / derivative( x: x0));
printf( format: "At iteration %d: x = %f\n", i, x1);
```

Newton-Rahson root approximation x1 is computed via the Newton-Raphson Formula stated in the Task Question, and current iteration is printed accompanied with calculated root approximation at given iteration.

```
// Check if the absolute difference is less than the tolerance
if (fabs( X: polynomial( x: x1)) < tolerance) {
    printf( format: "\nConverged to the desired tolerance.\n");
    break;
}
```

Convergence check is executed, breaking the do-while if absolute difference between polynomials at x1 is less than specified tolerance.

Variables are updated and counter i is incremented.

The do-while keeps executing until something inside breaks it, or if the counter reaches the maximum number of iterations.

```
if (i <= maxItr && derivative( x: x0) != 0) {
    printf( format: "\nRoot found successfully.\n");
} else {
    printf( format: "\nRoot not found within the specified tolerance or maximum iterations.\n");
}
free( Memory: coeff);
return x1;
```

If the counter reaches maximum iterations and the derivative was not equal to zero, a success message is printed, otherwise not found. Memory is cleared and x1 is returned.

Secant Method:

```c
double secantMethod () { // Secant Method
    clearBuffer();
    printf( format: "Secant Method\n");
    printf( format: "_____\n");
    double x0, x1, x2, tolerance;
    int i = 1, maxItr;

    coeff = getCoefficients(); // Get coefficients from user

    if (coeff == NULL) {
        printf( format: "Error: Coefficients allocation failed\n");
        return 0;
    }

    clearBuffer();

    x0 = getInitialGuess( order: 0);
    x1 = getInitialGuess( order: 1);

    tolerance = getTolerance();
    maxItr = getMaxIterations();

    if (maxItr < 1) {
    do {
        if (polynomial( x: x1) - polynomial( x: x0) == 0) {
            printf( format: "Error: Mathematical Error, Denominator equal to zero");
            break;
        }

        // Check if the absolute difference is less than the tolerance
        if (fabs( X: polynomial( x: x2)) < tolerance) {
            printf( format: "\nConverged to the desired tolerance.\n");
            break;
        }

        x2 = x1 - (polynomial( x: x1) * ( (x1 - x0) / (polynomial( x: x1) - polynomial( x: x0)) ) );
        printf( format: "At iteration %d: x = %f\n", i, x2);

        x0 = x1;
        x1 = x2;  // Update x0 and x1 after checking convergence

        i++;
    } while (i <= maxItr && polynomial( x: x1) - polynomial( x: x0) != 0);

    return x2;
}
```

Similarly to the Newton-Raphson method, buffer is cleared, display for function is displayed and the required variables are initialised, namely x0 and x1 being our initial guesses and x2 being our root approximation

```
coeff = getCoefficients(); // Get coefficients from user

if (coeff == NULL) {
    printf( format: "Error: Coefficients allocation failed\n");
    return 0;
}

clearBuffer();
```

The previously explained getCoefficients() function is called and stored in a variable coeff. Validation is set into place to make sure a value is returned to coeff, otherwise an error is displayed and the function is stopped.

Buffer is cleared once again.

```
x0 = getInitialGuess( order: 0);
x1 = getInitialGuess( order: 1);
```

,

Initial Guesses are stored into variables via its designated function and order parameters

```
tolerance = getTolerance();
maxItr = getMaxIterations();
```

Tolerance and MaxIterations are also stored into the previously initialised variables via their designated functions

```
if (maxItr < 1) {
    printf( format: "Error: Iterations must be larger than 0\n");
    return 0;
}
```

Validation for Iterations is checked again if its smaller to or equal to zero

A do-while loop is initialised until convergence or maximum iterations is reached.

```
do {
    if (polynomial( x: x1) - polynomial( x: x0) == 0) {
        printf( format: "Error: Mathematical Error, Denominator equal to zero");
        break;
    }
```

Denominator is checked to ensure it is not zero, if the case an error is printed and do-while is terminated via break.

```
// Check if the absolute difference is less than the tolerance
if (fabs( X: polynomial( x: x2)) < tolerance) {
    printf( format: "\nConverged to the desired tolerance.\n");
    break;
}
```

Tolerance is checked for further validation to ensure absolute value of polynomial with parameter x2 is less than our tolerance. If the case, a print explaining convergence is displayed and do-while is terminated via break.

```
x2 = x1 - (polynomial( x: x1) * ( (x1 - x0) / (polynomial( x: x1) - polynomial( x: x0)) ) );
printf( format: "At iteration %d: x = %f\n", i, x2);
```

Secant Root-Finding Method is performed utilising formula previously mentioned within the tasks questions, and storing it in its calculated root variable x2. Iteration number is printed accompanied with its calculated root at said iteration.

Variables are changed as necessary incase of another loop, and counter i is incremented.

Once the do-while loop is exited, the value of x2 is returned.

Menu and Main Functionality Menu:

```
void showMenu() {
    printf( format: "\nRoot-Finding Methods Menu\n");
    printf( format: "1. Newton-Raphson Method\n");
    printf( format: "2. Secant Method\n");
    printf( format: "3. Exit\n");
    printf( format: "Enter your choice (1, 2, or 3):\n");
}
```

The showMenu() function is used to simplify and improve overall code readability within my main function. It simply prints the required items for the menu, including the title of menu, the methods ie. Newton-Raphson and Secant, and an option to exit, accompanied with instructions on what is expected as input.

Main:

```c
int main() {
    int choice;

    while (1) {
        showMenu();
        if (scanf( format: "%d", &choice) != 1) {
            printf( format: "Invalid input. Please enter a number.\n");
            clearBuffer();  // Clear the input buffer
            continue;  // Skip the rest of the loop and ask for input again
        }
        switch (choice) {
            case 1 : newtonRaphson(); break;
            case 2 : secantMethod(); break;
            case 3 :
                printf( format: "Exiting");
                return 0;
            default:
                printf( format: "Wrong input\n");
                break;
        }
    }
}
```

The main function is where everything comes together. An indefinite while loop is used to keep prompting the menu to the user. showMenu() is called and validation for input is caught via the if statement, in which the buffer is cleared right after a print explaining the invalid input.

Once int choice has been successfully stored with the user's valid choice, a switch statement with parameter choice is used, with the various cases matching up with our previously stated menu. The Newton-Raphson function is called for 1, the Secant function for 2, and a print for exiting and a return 0 for case 3. If any other input is entered, a default takes care of printing that it's not a valid input and shows the menu again.

Menu can only be exited via the case 3 exit.

Assumptions and Testing

Assumptions:

Various assumptions were made in the creation of this task.

1. Polynomial Degree

   Program assumes that the user will provide coefficients for a polynomial of at most degree 5. The program only takes input of the first 5 valid inputs entered when obtaining coefficients, and the rest are cleared using clearBuffer().

2. Memory Allocation Success

Program assumes that memory allocation for all items are successful, which is why if it does fail, validation was embedded to display an error message and function terminating.

3. Valid numeric Input

   Program assumes that the user will enter the valid numeric input, which is why validation was ensured to handle such cases and prompt the user again until valid input is obtained.

4. Convergence

   The convergence utilised is assumed for well-behaved function within specified tolerance and iterations. If a function is of complex behaviour, the item may not converge.

Testing:

Various methods of testing were utilised to ensure a robust and efficient algorithm.

Menu Testing:

```
Root-Finding Methods Menu
1. Newton-Raphson Method
2. Secant Method
3. Exit
Enter your choice (1, 2, or 3):
```

Upon running the program via the CMakeLists.txt, the user is prompted with the menu.

```
Enter your choice (1, 2, or 3):
7
Wrong input

Root-Finding Methods Menu
1. Newton-Raphson Method
2. Secant Method
3. Exit
Enter your choice (1, 2, or 3):
```

If wrong input is entered, the default in our switch takes care of displaying a wrong input message and displays the menu again.

```
Invalid input. Please enter a number.

Root-Finding Methods Menu
1. Newton-Raphson Method
2. Secant Method
3. Exit
Enter your choice (1, 2, or 3):
```

If input of data type not belonging to an integer is entered (e.g. char), the user is prompted with an error explaining such and the menu is displayed again.

Newton-Raphson Method Testing:

```
Root-Finding Methods Menu
1. Newton-Raphson Method
2. Secant Method
3. Exit
Enter your choice (1, 2, or 3):
1
Newton-Raphson Method
---------------------

Ax^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F
Enter coefficients of polynomial (A B C D E F)
```

If the correct number corresponding to the newton-raphson method is entered, in this case 1, we are taken to its function.

The Function displays its specified printf ‘s and asks for user input regarding coefficients.

```
Enter coefficients of polynomial (A B C D E F)
g
Error: Invalid input. Please provide coefficients in the correct format.
Ax^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F
Enter coefficients of polynomial (A B C D E F)
```

If input which isn't an integer is entered, the user is prompted with the following error message and asked for coefficient input once again.

```
Ax^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F
Enter coefficients of polynomial (A B C D E F)
0 0 0 0 0 0
All coefficients are zero. Root is trivially 0.
Error: Coefficients allocation failed
```

If coefficients are all entered as zero, two errors are displayed to the user explaining the reasonings and also the coefficient allocation failure due to such invalid input. We are taken back to the main menu after this page.

```
Enter coefficients of polynomial (A B C D E F)
1 2 3 4 5 6
Your polynomial is:
 1.000x^5 + 2.000x^4 + 3.000x^3 + 4.000x^2 + 5.000x + 6.000

The derivative is:
 5.000x^4 + 8.000x^3 + 9.000x^2 + 8.000x + 5.000

Enter the initial guess for x0:
```

If 6 valid inputs are entered, we are shown our entered polynomial and its calculated derivative is displayed also.

Inputs for coefficient can also be entered as such:

```
Enter coefficients of polynomial (A B C D E F)
1
2
3 4
5 6
Your polynomial is:
 1.000x^5 + 2.000x^4 + 3.000x^3 + 4.000x^2 + 5.000x + 6.000

The derivative is:
 5.000x^4 + 8.000x^3 + 9.000x^2 + 8.000x + 5.000

Enter the initial guess for x0:
```

Now we are prompted for initial guesses.

```
Enter the initial guess for x0:
a
Error: Invalid input. Please enter a valid numerical value.
Enter the initial guess for x0:
```

If the value entered is not of desired type, and this is prompted via an error message to the user.
Initial guess is to be entered again.

```
Enter the initial guess for x0:
2

Enter the tolerable error for convergence (e.g., 0.00001):
```

If valid input is entered, we move on to the tolerance.

```
Enter the initial guess for x0:
1 2

Enter the tolerable error for convergence (e.g., 0.00001):
```

Also note that for all these functions, if any extra values are entered by mistake, the clearBuffer() takes care of ignoring them and they are not stored anywhere either.

```
Enter the tolerable error for convergence (e.g., 0.00001):
a
Error: Invalid input. Please enter a valid positive numerical value.
Enter the tolerable error for convergence:
```

If the value entered is not of desired type, and this is prompted via an error message to the user. Convergence is to be entered again.

```
Enter the tolerable error for convergence (e.g., 0.00001):
-1
Error: Invalid input. Please enter a valid positive numerical value.
Enter the tolerable error for convergence:
```

If an input of a negative value is entered, the user is prompted with the same error message as before stating its reasoning. Value to be entered again.

```
Enter the tolerable error for convergence:
0.000001

Enter the maximum number of iterations:
```

If valid input is accepted, we move on to the maximum number of iterations

```
Enter the maximum number of iterations:
a
Error: Invalid input. Please enter a valid positive integer value.
Enter the maximum number of iterations:
```

If the value entered is not of desired type, and this is prompted via an error message to the user.
Maximum number of iterations is to be entered again.

```
Enter the maximum number of iterations:
0
Error: Invalid input. Please enter a valid positive integer value.
Enter the maximum number of iterations:
```

If a value less than or equal to zero is entered, the same error message is displayed to the user and the value is to be entered one more time.

Once this is successfully entered, we are given all iterations and root approximations until one of the previously mentioned parameters is reached in our do-while.

E.g. Using various sources online, we can test out a newton-raphson calculation on a specific function on our algorithm and then verify it with such sources.

f(x) = x^3+3*x^2+12*x+8 with initial guess = -5

Tolerance = 0.0001

The answer must be equal to approximately -0.7790

```
Ax^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F
Enter coefficients of polynomial (A B C D E F)
0 0 1 3 12 8
Your polynomial is:
 0.000x^5 + 0.000x^4 + 1.000x^3 + 3.000x^2 + 12.000x + 8.000

The derivative is:
 0.000x^4 + 0.000x^3 + 3.000x^2 + 6.000x + 12.000

Enter the initial guess for x0:
-5
```

```
Enter the tolerable error for convergence (e.g., 0.00001):
0.0001

Enter the maximum number of iterations:
10
At iteration 1: x = -3.210526
At iteration 2: x = -1.828561
At iteration 3: x = -0.922025
At iteration 4: x = -0.778122
At iteration 5: x = -0.778977
```

As previously stated, the root should have been 0.7790, which is the same as my root when rounded up to the nearest 4th decimal place.

Secant Method Testing:

For this method, most functions such as getCoefficients(), clearBuffer(), polynomial(x), displayPolynomial() and much more were used, the same as in the previous Newton Raphson Function.

Hence most testing done previously also accounts for this one, which is why it will not be included.

```
Enter your choice (1, 2, or 3):
2
Secant Method
---------------------

Ax^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F
Enter coefficients of polynomial (A B C D E F)
```

Running the Secant method prompts you with this display.

```
Ax^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F
Enter coefficients of polynomial (A B C D E F)
1 2 3 4 5 6
Your polynomial is:
 1.000x^5 + 2.000x^4 + 3.000x^3 + 4.000x^2 + 5.000x + 6.000

Enter the initial guess for x0:
```

The same test verifications in previous Newton-Raphson, as previously stated, also account for all these inputs. Hence memory allocation, trivial cases, input boundaries and input validation which were previously mentioned will be skipped.

```
Enter the initial guess for x0:
1

Enter the initial guess for x1:
2
```

Entered initial guesses

```
Enter the tolerable error for convergence (e.g., 0.00001):
0.0001

Enter the maximum number of iterations:
5
```

Maximum iterations and tolerance, after which secant method runs and root is found.
E.g. Using various sources online, we can test out a secant calculation on a specific function on our algorithm

and then verify it with such sources. f(x) = x^3+3*x^2+12*x+8 with initial guess = -5, 5

Tolerance = 0.0001

The answer must be equal to approximately -0.7790

```
At iteration 1: x = -2.243243
At iteration 2: x = -1.856641
At iteration 3: x = -1.019151
At iteration 4: x = -0.796358
At iteration 5: x = -0.778790
At iteration 6: x = -0.778978

Converged to the desired tolerance.
```

Inputting the values stated above gives us the same result if rounded up to the nearest 4th decimal place.

Both methods returned the same calculated root, which further proves its legitimacy.

```
Root-Finding Methods Menu
1. Newton-Raphson Method
2. Secant Method
3. Exit
Enter your choice (1, 2, or 3):
3
Exiting
Process finished with exit code 0
```

Inputting value 3 successfully terminates the running of the program.

Task 2 A Generic Set Library

In this section of the documentation, a generic set library implementation is to be tackled, made to accept various element types. The task is divided into 3 parts.

Task Definition:

2. **A Generic Set library.** (Total-55 marks)

In computer science, *sets* are typically used as a simple collection of objects, called elements, that contain no duplicates. In this task, you are to implement a library that provides an implementation for a *Generic Set*, meaning *it can be instantiated for multiple types*, but with each instance being only concerned with a single element type per set instance. The result is a new data type called `GenSet_t` supporting the following operations:

- `initSet()` - Creates a new set for a specific element type and allocates associated memory resources.
- `deinitSet()` - Destroys an existing set and relinquishes associated memory resources.
- `addToSet()` - Adds a non-duplicate element to the set.
- `displaySet()` - Outputs all elements of a set in no particular order to the standard output.
- `unionSet()` - Returns a newly created set, the union of two input sets.
- `intersectSet()` - Returns a newly created set, the intersection of two input sets.
- `diffSet()` - Returns a newly created set, the difference between two input sets.
- `countSet()` - Returns the element count, or cardinality, of an input set.
- `isSubsetSet()` - Returns whether the first set is a subset of the second.
- `isEmptySet()` - Returns whether a set is empty.

All operations above refer to the usual set operations. Functions that take more than one set as input must ensure that *all input sets are instantiated for objects of the same type*. Set elements are to be stored in dynamic memory (on the heap).

Tasks: -

(a) Implement a simplified version of `GenSet_t` that is instantiated with either elements of integer type or of fixed-sized 64-character strings. Both options need to be provided. Test all functionality.

[20 marks]

(b) Extend the implementation in 2(a) so that there are no limitations on the supported element types this time. Additionally, support for the following additional operation is required:

`export()` - Exports the Generic Set elements to a text file, with each line displaying a textual representation of each element in no particular order.

Test the additional functionality.

[20 marks]

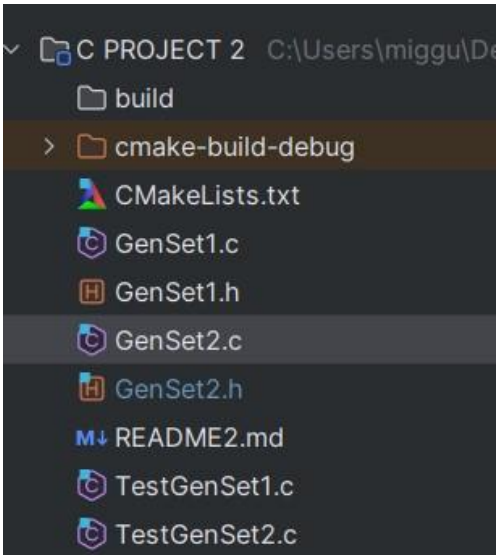
(c) Compile the full implementation as a shared library and provide a proper Abstract Data Type interface. The test application should link with the shared library. In case of difficulties with task 2(b), you can compile the library from the simpler version developed in 2(a), with a maximum of 10 marks.

[15 marks]

File Organization File

structure:

C PROJECT -> C PROJECT 2



Question 1 -> GenSet1.c, GenSet1.h, TestGenSet1.c
Question 2 -> GenSet2.c, GenSet2.h, TestGenSet2.c

Functions Overview

a. GenSet for Integers and Strings

initSet() -> Initialize a set for a specific element type deinitSet() -> Destroy an existing set and release associated memory addToSet() -> Add a non-duplicate element to set displaySet()-> Output all elements of set unionSet() -> Return newly created set, union of 2 sets intersectSet() -> Return newly created set, intersection of 2 sets diffSet() -> Return newly created set, difference of 2 sets countSet() -> return element count of set isSubsetSet() -> Return Set A subset Set B isEmpty() -> Return whether set empty

b. Extended Generic Genset

The same functions as before + export() -> Export Generic Set elements to txt file

Source Code Explanation

a. Simplified GenSet:

GenSet1.h

```h
#ifndef GENSET_H
#define GENSET_H

#include ...

#define SUCCESS 0
#define NULL_POINTER_ERROR 1
#define ELEMENT_EXISTS_ERROR 2
#define DEALLOCATED_ERROR 3

typedef struct {
    size_t elementSize;
    size_t length;
    void **elements;
    bool deallocation;  // flag for deinitSetInt(), shows if been deallocated
    bool isInt; // flag for type, is integer or isnt
} GenSet_t;

GenSet_t *initSet(size_t elementSize); // return NULL if allocation failure (set/elements)
int deinitSet(GenSet_t *set); // Follows Error Codes
bool isEmptySet(GenSet_t *set); // true: dealloc/empty, false: not empty
bool isSubsetSet(GenSet_t *set1, GenSet_t *set2); // NULL: dealloc/NULL sets, diff set types, true/false
int addToSet(GenSet_t *set, const void *element); // Follows Error Codes
int displaySet(GenSet_t *set); // Follows Error Codes
int countSet(GenSet_t *set); // Follows Error Codes
GenSet_t *unionSet(GenSet_t *set1, GenSet_t *set2); // NULL: dealloc/NULL sets, diff set types
GenSet_t *intersectSet(GenSet_t *set1, GenSet_t *set2); // NULL: dealloc/NULL sets, diff set types
GenSet_t *diffSet(GenSet_t *set1, GenSet_t *set2); // NULL: dealloc/NULL sets, diff set types

#endif // GENSET_H
```

This header file gives a short overview of all the functionalities within this system. It also provides a list of error codes which use some of the functions. Each error code corresponds to a common error that might arise in said functions. This helps users better tackle error returns. The functions that do not use such error code systems have been left with comments explaining what each return could mean.

A struct for our own Abstract Data Type was made tackling many items, such as set's length, type and elements present, amongst others.

Parameters vary depending on the function at hand. The initSet() for example takes parameter elementSize which is used to determine whether the set created will be of type int, or of fixed sized 64-character string.

GenSet1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include "GenSet1.h"

bool ElementInSet(const void *element, GenSet_t *set) { // determines if element in set
    for (int i = 0; i < set->length; i++) {
        if (set->elementSize == sizeof(int) && memcmp( Buf1: set->elements[i], Buf2: element, Size: set->elementSize) == 0) {
            return true;
        } else if (set->elementSize == sizeof(char[64]) && strcmp((const char *)set->elements[i], (const char *)element) == 0) {
            return true;
        }
    }
    return false;
}
```

Here we will dive into our source file. The required #include 's are included on the top, and a function ElementInSet() was created to promote readability, reduce code complexity and reduce repetition of code.

It iterates through a set's length (set taken from parameter) and only checks if an element is in set if its type is either of type int or of char[64] (denoting a 64-character string of fixed size). If an element is in the set, it immediately returns true, otherwise false.

```
GenSet_t *initSet(size_t elementSize) { // returns -> NULL: Allocation failure (structure or elements), newSet
    if (elementSize != sizeof(int) && elementSize != sizeof(char[64])) { // check for type, only int and char[64]
        return NULL; // Invalid element size
    }

    GenSet_t *newSet = malloc( Size: sizeof(GenSet_t));  // Dynamic memory ie. heap used

    if (newSet == NULL) {
        return NULL;
    }

    newSet->length = 0; // Initially an empty set
    newSet->elements = malloc( Size: sizeof(void *) * 1); // Space allocated for a single value
    newSet->elementSize = elementSize; // data type assign

    if (newSet->elements == NULL) {
        // allocation failure for elements
        deinitSet( set: newSet);
        return NULL;
    }

    newSet->deallocation = false; // flag for deininSet(), used as validation/check in other functions
    if (newSet->elementSize == sizeof(int)) { // flag to see if set is of type int/char[64]
        newSet->isInt = true; // type int
    }
    else if (newSet->elementSize == sizeof(char[64])) {
        newSet->isInt = false; // type char[64]
    }
    return newSet;
}
```

This is the initSet(), responsible for initialising a set. It first makes sure the parameter entered for data type matches with either int or char[64], otherwise returning NULL. New set is initialised and allocated dynamic memory, and a NULL check is made to ensure it has been successfully allocated memory. Length etc are given their respective initial values, and elements are NULL checked to ensure dynamic memory allocation. Flags are set to their respective values and newSet is returned.

Returns are described in comments.

```
int deinitSet(GenSet_t *set) { // Follows Error Codes
    if (set == NULL) {
        return NULL_POINTER_ERROR; // Avoid dereferencing a null pointer
    }

    for (int i = 0; i < set->length; i++) {
        free( Memory: set->elements[i]);
    }
// free every dynamically stored memory
    free( Memory: set->elements);
    set->deallocation = true;
    free( Memory: set);
    return SUCCESS;
}
```

This function takes care of deinitialising a set. First performs a NULL check to avoid dereferencing a null pointer, then iterates through each element in the set and frees it, followed by setting the deallocation flag and freeing the set entirely.

This function follows the ERROR CODES.

```
bool isEmptySet(GenSet_t *set) { // true: dealloc/empty, false: not empty
    if (set->length == 0 || set->deallocation) {
        // set empty/has been deallocated
        return true;
    } else {
        //set not empty
        return false;
    }
}
```

This function simply returns whether a function is empty or not, via its length. If a set has been deallocated, it also returns empty.

Returns described in comments.

From here on out, NULL and deallocation checks are common in all functions deemed necessary, so they will not be mentioned again.
Similarly, all returns are described in the functions comments up top, and will denote whether it follows the previously stated ERROR CODES or if it follows its own (due it not being of type int).

```
bool isSubsetSet(GenSet_t *set1, GenSet_t *set2) { // NULL: dealloc/NULL sets, diff set types, true/false
    if (set1->deallocation || set2->deallocation) {
        // dealloc check
        return false;
    }
    if (set1 == NULL || set2 == NULL) {
        // NULL check
        return false;
    }

    if (set1->isInt != set2->isInt) {
        // type check
        return false;
    }

    for (int i = 0; i < set1->length; i++) {
        if (!ElementInSet( element: set1->elements[i],  set: set2)) {
            return false;
        }
    }

    return true;
}
```

This function simply returns true if set1 is a subset of set2. This is done by iterating through set1 ‘s length and if any element of set1 isn't in set2, it immediately returns false, otherwise true.

Here we get our first instance of a type check. For functions with more than 1 set as a parameter, we are to ensure that both sets are of the same type. This check will be present in all functions which are as such and will not be mentioned again for the sake of reducing repetition within this documentation.

```c
int addToSet(GenSet_t *set, const void *element) { // Follows Error Codes
    if (set->deallocation) { // dealloc
        return DEALLOCATED_ERROR;
    }
    if (set == NULL) { // null
        return NULL_POINTER_ERROR;
    }


    if (set->isInt && !ElementInSet(element, set)) { // adds if int
        set->elements = realloc( Memory: set->elements,  NewSize: sizeof(void *) * (set->length + 1));
        set->elements[set->length] = malloc( Size: set->elementSize);

        if (set->elements[set->length] == NULL) {
            deinitSet(set);
            return NULL_POINTER_ERROR;
        }

        memcpy( Dst: set->elements[set->length],  Src: element,  Size: set->elementSize); // add element to set
        set->length += 1;
        return SUCCESS;
    } else if (set->elementSize == sizeof(char[64]) && !ElementInSet(element, set)) { // adds if 64-char string
        set->elements = realloc( Memory: set->elements,  NewSize: sizeof(void *) * (set->length + 1));
        set->elements[set->length] = malloc( Size: set->elementSize);

        if (set->elements[set->length] == NULL) {
            deinitSet(set);
            return NULL_POINTER_ERROR;
        }

        strcpy( Dest: (char *)set->elements[set->length],  Source: (const char *)element); // add element to set
        set->length += 1; // increment length
        return SUCCESS;
    }

    return ELEMENT_EXISTS_ERROR;
}
```

This function handles adding of elements to a specified set. It only adds elements if the type of set is of type int or char[64], ensuring validation across sets.

Both options offer a similar implementation, with the integer reallocating memory to the structs elements, ensuring a NULL check in which case it deinitiates set and uses memcpy to copy value at memory element to our desired destination, with the specified size. Length is incremented and function returns.

Similar goes to that of char[64]. It does the exact same with the slight change of using strcpy instead of memcpy when entering the element in the set, since it is of type string.

```c
int displaySet(GenSet_t *set) {  // Follows Error Codes
    if (set->deallocation) {
        return DEALLOCATED_ERROR;
    }
    if (set == NULL) {
        return NULL_POINTER_ERROR;
    }

    printf( format: "{");

    for (int i = 0; i < set->length; i++) {
        // prints all elements of set, only if set is of type int/char[64]
        if (set->elementSize == sizeof(int)) {
            printf( format: "%d", *((const int *)set->elements[i]));
        } else if (set->elementSize == sizeof(char[64])) {
            printf( format: "%s", (const char *)set->elements[i]);
        }

        if (i < set->length - 1) {
            printf( format: ", ");
        }
    }

    printf( format: "}\n");

    return SUCCESS;
}
```

This function displays set elements to the user. It iterates through set length, and only displays elements if the set is of type integer or char[64]. Each has their respective printf statements due to different format specifiers. A comma is placed between elements up until the last element is reached where it isn't.

```c
int countSet(GenSet_t *set) {  // Follows Error Codes
    if (set->deallocation) {
        return DEALLOCATED_ERROR;
    }
    if (set == NULL) {
        return NULL_POINTER_ERROR;
    }
    // returns the length of the set ie. no of elements in set
    return set->length;
}
```

This function is quite simple and straightforward. It simply returns the number of elements within a set, accessed through the set structure's length.

```c
GenSet_t *unionSet(GenSet_t *set1, GenSet_t *set2) {  // NULL: dealloc/NULL sets, diff set types
    if (set1->deallocation || set2->deallocation) {
        return NULL;
    }
    if (set1 == NULL || set2 == NULL) {
        return NULL;
    }

    if (set1->isInt != set2->isInt) {
        return NULL;
    } // dealloc, mismatch type and null checks
    // creates new set to store union
    GenSet_t *newSet = initSet(set1->elementSize);
    // add elements of set1
    for (int i = 0; i < set1->length; i++) {
        addToSet( set: newSet,  element: set1->elements[i]);
    }
    // add elements of set3
    for (int i = 0; i < set2->length; i++) {
        addToSet( set: newSet,  element: set2->elements[i]);
    } // adds all elem of set 1 and 2

    return newSet;
}
```

This function tackles the union between two sets. It initiates a new set and performs the necessary operations to create set1 union set2. It iterates through each of the set elements and adds each one to the new set, once this is finished it is returned.

```c
GenSet_t *intersectSet(GenSet_t *set1, GenSet_t *set2) {  // NULL: dealloc/NULL sets, diff set types
    if (set1->deallocation || set2->deallocation) {
        return NULL;
    }
    if (set1 == NULL || set2 == NULL) {
        return NULL;
    }

    if (set1->elementSize != set2->elementSize) {
        return NULL;
    } // dealloc, mismatch type and null checks

    // creates new set to store intersection
    GenSet_t *newSet = initSet(set1->elementSize);

    for (int i = 0; i < set1->length; i++) {
        if (ElementInSet( element: set1->elements[i],  set: set2)) {
            addToSet( set: newSet,  element: set1->elements[i]);
        }
    } // adds elem of set1 to set if in set2

    return newSet;
}
```

This function handles the intersection of two sets. It initiates a new set,and iterates through set1's elements, adding the ones which are also in set 2 in the new set, and returning it.

```c
GenSet_t *diffSet(GenSet_t *set1, GenSet_t *set2) {  // NULL: dealloc/NULL sets, diff set types
    if (set1->deallocation || set2->deallocation) {
        return NULL;
    }
    if (set1 == NULL || set2 == NULL) {
        return NULL;
    }

    if (set1->elementSize != set2->elementSize) {
        return NULL;
    } // dealloc, mismatch type and null checks

    // creates new set to store diff
    GenSet_t *newSet = initSet(set1->elementSize);

    for (int i = 0; i < set1->length; i++) {
        if (!ElementInSet( element: set1->elements[i],  set: set2)) {
            addToSet( set: newSet,  element: set1->elements[i]);
        }
    } // adds elem of set1 to set if not in set 2

    return newSet;
}
```

This is our last function, and is responsible for the difference of two sets. It initiates a newset, iterates through set 1's elements and adds the ones which aren't in set2 in the new set, afterwards returning it.

b. Extension of GenSet

GenSet2.h

```c
#ifndef GENSET_H
#define GENSET_H

#include ...

typedef struct GenSet GenSet;

GenSet* initSet(size_t elementSize, int (*compare)(const void*, const void*), void (*display)(const void*),
                void (*filewrite)(const void*, FILE*));

void deinitSet(GenSet* set);
bool isEmptySet(GenSet* set);

void addToSet(GenSet* set, const void* element);
void displaySet(const GenSet* set);
size_t countSet(const GenSet* set);
bool isSubsetSet(const GenSet* setA, const GenSet* setB);
GenSet* unionSet(const GenSet* setA, const GenSet* setB);
GenSet* intersectSet(const GenSet* setA, const GenSet* setB);
GenSet* diffSet(const GenSet* setA, const GenSet* setB);
void exportSetToFile(const GenSet* set, const char* filename);
```

This header file quickly gives an overview of all the main functions within this set of code. The functions seen here were explained briefly in the previous section.

Each parameter taken corresponds with the functions used. Function pointers were utilised as well due to the generic data type capability required by the question, which entailed as such. Alot of parameters were given the `const` data type ensuring that functions can not modify the parameter itself and provides more compatibility with values, which is usual in such a generic set implementation.

GenSet2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>


#include "GenSet2.h"

// generic set data structure, where user-defined data types can be used.
// displaying of elements func to be supplied by user, since data type is user-defined
// comparing of elements func to be supplied by user, since data type is user-defined

// use of function pointers needed for these two, for library flexibility

struct GenSet{
    void** elements; // a pointer to array of void pointers, stores addresses of elements in set.
    // elements is a pointer to a pointer void, hence the 'void**', (pointer to a pointer)
    // the ** is used for flexibility w handling diff data types
    size_t length; // size_t -> bytes
    size_t elementSize;
    bool deallocation; // flag for deinitSet()

    int (*compare)(const void*, const void*);
    void (*display)(const void*);
    void (*filewrite)(const void*, FILE*);

};
```

The struct for GenSet is as follows:

- ➔ void** elements is a pointer to an array of void pointers, storing addresses of elements in set
- ➔ Size_t length is the size of set in bytes
- ➔ Size_t elementSize is the size of element in bytes
- ➔ Int (*compare)(const void*, const void*) is a function pointer for generic element comparison
- ➔ Void (*display)(const void*) is a function pointer for generic element display
- ➔ Void (*filewrite)(const void*, FILE*) is a function pointer for generic element file writing

```c
void* getMemoryLocation(const GenSet* set, size_t index) {
    return (char*)set->elements + index * set->elementSize;
    // pointer to memory of i-th element, better explained in personal notes
    // char* is used to turn set->elements into bytes
    // char* for better pointer arithmetic
    // (char*)set->elements -> casted pointer   set->elementSize * set->length -> offset eg. sizeof(int) * set->length

}
```

This function simply returns a pointer to memory of the i-th element, useful in many of the required functions. Helps with readability of code, and further explanation is found in snippet's comments.

```
GenSet* initSet(size_t elementSize, int (*compare)(const void*, const void*), void (*display)(const void*),
                void (*filewrite)(const void*, FILE*)) {
    GenSet* newSet = (GenSet*)malloc( Size: sizeof(GenSet));

    if (newSet == NULL) {
        // memory allocation failure
        return NULL;
    }

    newSet->elements = NULL;
    newSet->length = 0;
    newSet->elementSize = elementSize;
    newSet->deallocation = false;
    newSet->compare = compare;
    newSet->display = display;
    newSet->filewrite = filewrite;

    return newSet;
}
```

The initSet takes on the Abstract Data Type of GenSet (previously stated). It takes such parameters and does the following:

Allocates memory using malloc for the new set, checks whether memory allocation failed and if so returns NULL, and initiates various fields within the GenSet structure.

This function has function pointers as parameters which take in user-defined functions for display, comparison and file writing. Since the implementation is that of a generic set data structure, including use of user defined data types, these function pointers tell the program how to handle these data types in such cases.

```
void deinitSet(GenSet* set) {
    if (set == NULL) {
        return;
    }
    set->deallocation = true;

    free( Memory: set->elements);
    free( Memory: set);
}
```

deinitSet() is responsible for deallocating memory from the parameter set.

It includes a NULL check and returns if true, sets a deallocation flag to true (used in future functions as validation) and frees all memory.

```
bool isElementInSet(const GenSet* set, const void* element) {
    for (size_t i = 0; i < set->length; i++) {
        if (set->compare(element, getMemoryLocation(set, index: i)) == 0) {
            // checks to see if a-b = 0 essentially

            return true; // element found
        }
    }
    return false;
}
```

This function is used to provide better readability and reduce repetition of code. It simply iterates through a set and returns true if its found (using previously mentioned getMemoryLocation()) and otherwise false

```
void addToSet(GenSet* set, const void* element) {
    // error handling to be added later

    if (isElementInSet(set, element)) {
        return;
    }

    // calculating new elements memory location ie. adding offset to caster pointer
    void* elementMemoryLocation = malloc( Size: set->elementSize);
    if (!elementMemoryLocation) {
        // handle memory allocation failure
        return;
    }

    void* newElements = realloc( Memory: set->elements,  NewSize: set->elementSize * (set->length + 1)); // as in 2a
    if (newElements == NULL) {
        // memory allocation failure
        free( Memory: elementMemoryLocation);
        return;
    }
    set->elements = newElements;
    memcpy( Dst: elementMemoryLocation,  Src: element,  Size: set->elementSize);

    void* setElementMemoryLocation = getMemoryLocation(set,  index: set->length);
    memcpy( Dst: setElementMemoryLocation,  Src: elementMemoryLocation,  Size: set->elementSize);

    set->length = set->length + 1;

    // qsort(set->elements, set->length, set->elementSize, set->compare);
    free( Memory: elementMemoryLocation);
}
```

This function adds an element to a set. First it checks if an element is in set, and if so returns.

Then it allocates memory and this memory is checked, if not allocated it is returned. Memory is then reallocated for set's elements but not before a NULL check, in which case returns. Element is copied into the newly allocated memory using memcpy and set's length is updated. At the end, elementMemoryLocation is freed as it is no longer of need.

Most functions will all start having NULL and deallocation checks, in which case they return.
Henceforth they will not be mentioned again.

```c
void displaySet(const GenSet* set) {
    if (set == NULL || set->deallocation) {
        return;
    }

    printf( format: "Set elements: { ");

    for (size_t i = 0; i < set->length; i++) {
        // using function pointer
        set->display(getMemoryLocation(set,  index: i));
        if (i < set->length - 1) {
            printf( format: ", ");
        }
    }
    printf( format: " }\n");
}
```

This function is responsible for printing the content of a specified set. It iterates through the sets length and uses the display attribute of the GenSet structure which previously had a function pointer allocated with it in the initSet() function. This function pointer prints the elements and adds a , after each element given its not the last element of the set.

```c
size_t countSet(const GenSet* set) {
    if (set == NULL || set->deallocation) {
        return 0;
    }
    return set->length;
}
```

Function simply returns the set's length in bytes

```c
bool isSubsetSet(const GenSet* setA, const GenSet* setB) {
    if (setA == NULL || setB == NULL || setA->deallocation || setB->deallocation) {
        return false;
    }
    for (size_t i = 0; i < setA->length; i++) {
        if (!isElementInSet( set: setB,  element: getMemoryLocation( set: setA,  index: i))) {
            return false;
        }
    }
    // all elements of a in b, hence a subset b
    return true;
}
```

This function iterates through setA's length and for each iteration, if element is not in setB it returns false, otherwise true. NULL and deallocation checks for all sets as well.
The next couple of functions will also include NULL and deallocation checks, but for all sets provided, such as seen here.

The next couple of functions are also to ensure both functions are of the same type.

```
GenSet* unionSet(const GenSet* setA, const GenSet* setB) {
    if (setA == NULL || setB == NULL || setA->deallocation || setB->deallocation) {
        return false;
    }
    if (setA->elementSize != setB->elementSize || setA->compare != setB->compare || setA->display != setB->display) {
        // sets not of same type
        return NULL;
    }

    GenSet* unionSet = initSet(setA->elementSize, setA->compare, setA->display, setA->filewrite);

    // add of setA
    for (size_t i = 0; i < setA->length; i++) {
        addToSet( set: unionSet,  element: getMemoryLocation( set: setA,  index: i));
    }

    // add of setB
    for (size_t i = 0; i < setB->length; i++) {
        addToSet( set: unionSet,  element: getMemoryLocation( set: setB,  index: i));
    }
    return unionSet;
}
```

This function initiates a new unionSet, iterates through both setA and setB's length and adds every element of both, afterwards returning it.

```
GenSet* intersectSet(const GenSet* setA, const GenSet* setB) {
    if (setA == NULL || setB == NULL || setA->deallocation || setB->deallocation) {
        return false;
    }
    if (setA->elementSize != setB->elementSize || setA->compare != setB->compare || setA->display != setB->display) {
        // sets not of same type
        return NULL;
    }

    GenSet* intersectionSet = initSet(setA->elementSize, setA->compare, setA->display, setA->filewrite);

    for (size_t i = 0; i < setA->length; i++) {
        if (isElementInSet( set: setB,  element: getMemoryLocation( set: setA,  index: i))){
            addToSet( set: intersectionSet,  element: getMemoryLocation( set: setA,  index: i));
        }
    }
    return intersectionSet;
}
```

This function initiates a new intersectSet and iterates through setA's length. All elements in B get added to this new set, and is returned.

```c
GenSet* diffSet(const GenSet* setA, const GenSet* setB) {
    if (setA == NULL || setB == NULL || setA->deallocation || setB->deallocation) {
        return false;
    }
    if (setA->elementSize != setB->elementSize || setA->compare != setB->compare || setA->display != setB->display) {
        // sets not of same type
        return NULL;
    }

    GenSet* diffSet = initSet(setA->elementSize, setA->compare, setA->display, setA->filewrite);

    for (size_t i = 0; i < setA->length; i++) {
        if (!isElementInSet( set: setB,  element: getMemoryLocation( set: setA,  index: i))) {
            addToSet( set: diffSet,  element: getMemoryLocation( set: setA,  index: i));
        }
    }
    return diffSet;
}
```

diffSet() function first initiates a set diffSet, iterates through setA's length and if element is not in setB get added, and is later returned.

```c
void export(const GenSet* set, const char* filename) {
    if (set == NULL || set->deallocation) {
        return;
    }

    FILE* file = fopen(filename,  Mode: "w");
    if (file == NULL) {
        return;
    }

    for (size_t i = 0; i < set->length; i++) {
        set->filewrite(getMemoryLocation(set,  index: i), filename);
    }
    fclose(file);
}
```

This function opens a file in writing mode, and if this fails it returns early. It iterates through the sets length and writes to the file using the function pointer assigned to the filewrite attribute of the GenSet structure. Once this is done, it closes the file.

c. Shared Library and ADT interface

The previously shown header and c files were compiled into a shared library with a proper ADT interface.

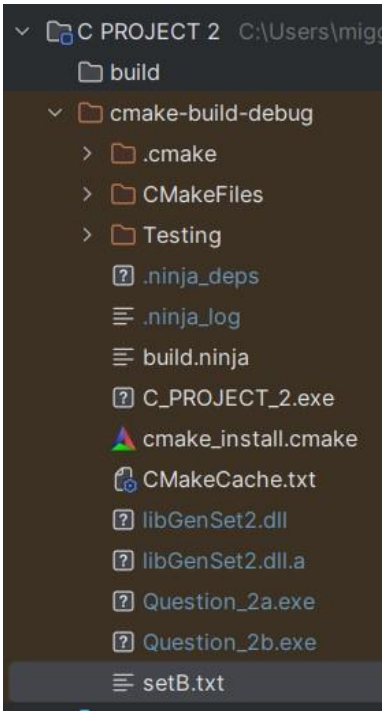
```c
#ifndef GENSET_H
#define GENSET_H

#include ...

typedef struct GenSet GenSet;

GenSet* initSet(size_t elementSize, int (*compare)(const void*, const void*), void (*display)(const void*
                void (*filewrite)(const void*, FILE*));

void deinitSet(GenSet* set);
bool isEmptySet(GenSet* set);

void addToSet(GenSet* set, const void* element);
void displaySet(const GenSet* set);
size_t countSet(const GenSet* set);
bool isSubsetSet(const GenSet* setA, const GenSet* setB);
GenSet* unionSet(const GenSet* setA, const GenSet* setB);
GenSet* intersectSet(const GenSet* setA, const GenSet* setB);
GenSet* diffSet(const GenSet* setA, const GenSet* setB);
void export(const GenSet* set, const char* filename);
```

This is the Abstract Data Type interface, giving a brief overview of all the functions available to use within the library.

```
cmake_minimum_required(VERSION 3.26)
project(C_PROJECT_2 C)

set(CMAKE_C_STANDARD 11)

add_executable(Question_2a
        TestGenSet1.c
        GenSet1.c
        )

add_library(GenSet2 SHARED GenSet2.c GenSet2.h)

add_executable(Question_2b
        TestGenSet2.c
        )

target_link_libraries(Question_2b GenSet2)
```

This is the CMakeLists.txt file, including the add_library set as SHARED and the target_link_libraries, effectively compiling everything into a shared library.

The respective .dll and object files are found within the cmake-build-debug folder.



Assumptions and Testing

Assumptions:

Question 1

It is assumed that only integers and fixed sized 64-character strings are permitted within these functions and their respective functionality.

It uses dynamic memory allocation, presuming the system has adequate sufficient memory for said allocation.

Users are expected to handle memory themselves via deinitSet(), and deinitalising any memory they allocated previously.

Function parameters are assumed to be entered correctly, and for mismatched parameter types to be avoided, in which case the proper checks and validation are in place to handle any and all occurrences of such.

Question 2

Users are expected to enter the sets of the same type into operations which require 2 sets as parameters, which could lead to problems if this is not met. Hence why validation has been embedded with many functions to ensure proper data types within operations.

Users must provide a valid element display function during initialization, displaying content of a single element. If function is not functional, this would be a problem that cannot be validated.

Users must also provide a valid element comparison function, and it must adhere to a specified format with a specified return value format.

Users are required to provide an element export function during initialization also which is responsible for writing elements into a text file. Failure in making a proper export function can lead to discrepancies in the library's overall functionality, as goes for the previous 2 assumptions.

Question 3

Assumption about compatibility with users operating system, as the files for the shared library is a .dll file, which is not the same as what is required for other systems (.so for Linux/Mac)

Users are expected to link test application with the compiled library

Users are assumed to interact with the libraries ADT interface and to not manipulate its internal structures

Testing:

Question 1

We will firstly begin by testing out the integer side of things.

```c
int main() {
    //INT
    printf( format: "INTEGER:\n");
    GenSet_t *integerSet1 = initSet( elementSize: sizeof(int)); // initializing sets
    GenSet_t *integerSet2 = initSet( elementSize: sizeof(int)); // w type int

    int elem1 = 1;
    int elem2 = 2; int elem3 = 3;
    int elem4 = 4; int elem5 = 5; // declaring int variables

    addToSet( set: integerSet1,  element: &elem1);
    addToSet( set: integerSet1,  element: &elem2);
    addToSet( set: integerSet1,  element: &elem3);
    // adding vars to set
    addToSet( set: integerSet2,  element: &elem3);
    addToSet( set: integerSet2,  element: &elem4);
    addToSet( set: integerSet2,  element: &elem5);

    printf( format: "Set 1: ");
    displaySet( set: integerSet1);
    //display sets
    printf( format: "Set 2: ");
    displaySet( set: integerSet2);
```

```c
  //set operations
  GenSet_t *unionintegerSet = unionSet( set1: integerSet1,  set2: integerSet2);
  printf( format: "Union Set: ");
  displaySet( set: unionintegerSet);

  GenSet_t *intersectintegerSet = intersectSet( set1: integerSet1,  set2: integerSet2);
  printf( format: "Intersection Set: ");
  displaySet( set: intersectintegerSet);

  GenSet_t *diffintegerSet = diffSet( set1: integerSet1,  set2: integerSet2);
  printf( format: "Difference Set: ");
  displaySet( set: diffintegerSet);
```

This set of code, when run, handles any and all operations involving integers only.

```
INTEGER:
Set 1: {1, 2, 3}
Set 2: {3, 4, 5}
Union Set: {1, 2, 3, 4, 5}
Intersection Set: {3}
Difference Set: {1, 2}
```

This is the output when finished, handling all operations correctly.

Now we will tackle the 64-character string side of things

```
// CHAR
printf( format: "\nCHARACTER:\n");
GenSet_t *characterSet1 = initSet( elementSize: sizeof(char[64])); // initializing sets
GenSet_t *characterSet2 = initSet( elementSize: sizeof(char[64])); // w type 64-char string


addToSet( set: characterSet1,  element: "test1");
addToSet( set: characterSet1,  element: "test2");
addToSet( set: characterSet1,  element: "test3");
// adding str to set
addToSet( set: characterSet2,  element: "test3");
addToSet( set: characterSet2,  element: "test4");
addToSet( set: characterSet2,  element: "test5");

printf( format: "Set 1: ");
displaySet( set: characterSet1);
// displaying sets
printf( format: "Set 2: ");
displaySet( set: characterSet2);
```

```
// sset operations
GenSet_t *uniooncharacterSet = unionSet( set1: characterSet1,  set2: characterSet2);
printf( format: "Union Set: ");
displaySet( set: uniooncharacterSet);

GenSet_t *intersectcharacterSet = intersectSet( set1: characterSet1,  set2: characterSet2);
printf( format: "Intersection Set: ");
displaySet( set: intersectcharacterSet);

GenSet_t *diffcharacterSet = diffSet( set1: characterSet1,  set2: characterSet2);
printf( format: "Difference Set: ");
displaySet( set: diffcharacterSet);
```

This handles any and all functions in regards to 64-character strings.

```
CHARACTER:
Set 1: {test1, test2, test3}
Set 2: {test3, test4, test5}
Union Set: {test1, test2, test3, test4, test5}
Intersection Set: {test3}
Difference Set: {test1, test2}
```

This is the output when finished, handling all operations correctly.

Now we will go to a few different cases to ensure that the code is of robust nature.

```
// CHECKS AND TESTING
if (!(isSubsetSet( set1: integerSet2,  set2: characterSet1))) { printf( format: "Mismatches sets\n"); } // VALIDATION CHECK
if (unionSet( set1: integerSet1,  set2: characterSet2) == NULL) { printf( format: "MISMATCH SETS FOR UNION\n"); } // VALIDA
if (intersectSet( set1: integerSet1,  set2: characterSet2) == NULL) { printf( format: "MISMATCH SETS FOR INTERSECT\n"); } /
if (diffSet( set1: integerSet1,  set2: characterSet2) == NULL) { printf( format: "MISMATCH SETS FOR DIFF\n"); } // VALIDATI
```

This set of code tests all mismatching possible within the system, that is mismatching of data types of sets. This could be a big issue if not resolved and tackled correctly.

```
Mismatches sets
MISMATCH SETS FOR UNION
MISMATCH SETS FOR INTERSECT
MISMATCH SETS FOR DIFF
```

When run, we can see that it stops the operation and prints out that it's a mismatch due to the caught NULL returns.

Now we will also check if we can initiate sets of incorrect type ie. not of our desired int/char[64] type

```
GenSet_t *invalidSet = initSet( elementSize: sizeof(double));
if (invalidSet == NULL) {printf( format: "INVALID SET RETURNED NULL\n");}
```

These lines of code will be the determinant of whether or not it successfully tackles such an obstacle.

```
INVALID SET RETURNED NULL
```

And as required, the program tackles the invalid set error perfectly, not initialising the set and returning NULL which is what caused such a print to be displayed.

Question 2, 3

```
int charComparison(const void* a, const void* b) {
    return (*(char*)a - *(char*)b); // via type-casting
}

void charDisplaying(const void* element) {
    printf( format: "%c", *(char*)element);
}

void CharFileWrite(const void* element, FILE* name) {
    FILE* file = fopen( Filename: name, Mode: "a");
    if (file == NULL) {
        return;
    }

    fprintf( stream: file, format: "%c\n", *(char*)element);

    fclose(file);
}
```

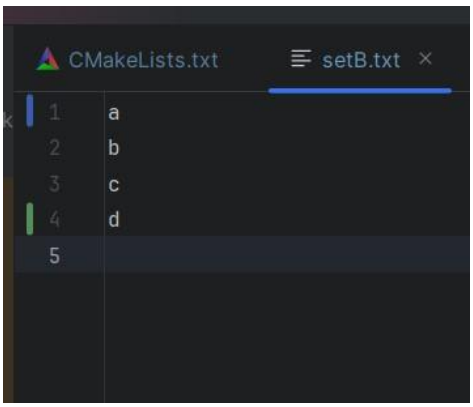
These are examples of the user-defined functions needed. Users are to define something of this format for proper functionality.

```c
int main () {
    GenSet *setA = initSet( elementSize: sizeof(char),  compare: charComparison,  display: charDisplaying,  filewrite: CharFileWrite);
    GenSet *setB = initSet( elementSize: sizeof(char),  compare: charComparison,  display: charDisplaying,  filewrite: CharFileWrite);


    if (isEmptySet( set: setA)) { printf( format: "Set A is empty/deallocated"); }
    else { printf( format: "Set A is not empty"); }
    printf( format: " after initialization\n");
    if (isEmptySet( set: setB)) { printf( format: "Set B is empty/deallocated"); }
    else { printf( format: "Set is B not empty"); }
    printf( format: " after initialization\n");

    char element1 = 'a'; char element4 = 'd';
    char element2 = 'b';
    char element3 = 'c';

    addToSet( set: setA,  element: &element1); addToSet( set: setB,  element: &element4);
    addToSet( set: setA,  element: &element2); addToSet( set: setB,  element: &element2);
    addToSet( set: setA,  element: &element3); addToSet( set: setB,  element: &element3);

    if (isEmptySet( set: setA)) { printf( format: "Set A is empty/deallocated\n"); }
    else { printf( format: "Set A is not empty\n"); }
    if (isEmptySet( set: setB)) { printf( format: "Set B is empty/deallocated\n"); }
    else { printf( format: "Set B is not empty\n"); }

    displaySet( set: setA);
    displaySet( set: setB);

    size_t numberOfElements = countSet( set: setA);
    printf( format: "Number of elements in Set A: %zu\n", numberOfElements);
    if (isSubsetSet(setA, setB)) {printf( format: "A subset B\n"); }
    else { printf( format: "A not subset B\n"); }

    GenSet* setC = unionSet(setA, setB); // all elements of a and b
    displaySet( set: setC);

    GenSet* setD = intersectSet(setA, setB); // all elements of a and b
    displaySet( set: setD);

    GenSet* setE = diffSet(setA, setB); // all elements of a and b
    displaySet( set: setE);

    export( set: setB,  filename: "setB.txt");

    deinitSet( set: setA);
    displaySet( set: setA);

    if (isEmptySet( set: setA)) { printf( format: "Set is empty/deallocated"); }
    else { printf( format: "Set is not empty"); }
    printf( format: " after deinitialization\n");
}
```

This will be our Testing for every function, using char as test input.

```
Set A is empty/deallocated after initialization
Set B is empty/deallocated after initialization
Set A is not empty
Set B is not empty
Set elements: { a, b, c }
Set elements: { d, b, c }
Number of elements in Set A: 3
A not subset B
Set elements: { a, b, c, d }
Set elements: { b, c }
Set elements: { a }
Set is empty/deallocated after deinitialization

Process finished with exit code 0
```

This is the output. Everything works as intended, and a setB.txt has been created with contents of setB written in a text file setB.txt within the projects directory, with each element being written on a new line as required.



Lets now test it out with Integer type functions and input.

All code in main remains the same except for these slight tweaks:

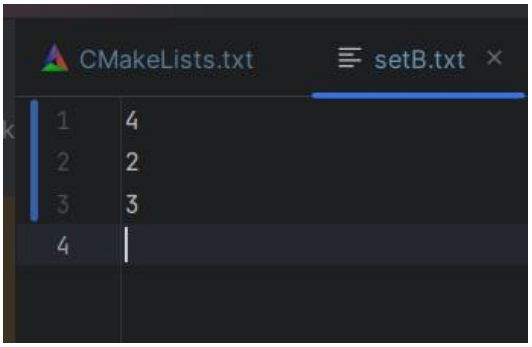
```c
int intComparison(const void* a, const void* b) {
    return (*(int*)a - *(int*)b); // via type-casting
}

void intDisplaying(const void* element) {
    printf( format: "%d ", *(int*)element);
}

void IntFileWrite(const void* element, FILE* name) {
    FILE* file = fopen( Filename: name,  Mode: "a");
    if (file == NULL) {
        return;
    }

    fprintf( stream: file,  format: "%d\n", *(int*)element);

    fclose(file);
}
```

```c
int main () {
    GenSet *setA = initSet( elementSize: sizeof(int),  compare: intComparison,  display: intDisplaying,  filewrite: IntFileWrite);
    GenSet *setB = initSet( elementSize: sizeof(int),  compare: intComparison,  display: intDisplaying,  filewrite: IntFileWrite);


    if (isEmptySet( set: setA)) { printf( format: "Set A is empty/deallocated"); }
    else { printf( format: "Set A is not empty"); }
    printf( format: " after initialization\n");
    if (isEmptySet( set: setB)) { printf( format: "Set B is empty/deallocated"); }
    else { printf( format: "Set is B not empty"); }
    printf( format: " after initialization\n");

    int element1 = 1; int element4 = 4;
    int element2 = 2;
    int element3 = 3;
```

With these changes, the following output occurs:

```
Set A is empty/deallocated after initialization
Set B is empty/deallocated after initialization
Set A is not empty
Set B is not empty
Set elements: { 1 , 2 , 3  }
Set elements: { 4 , 2 , 3  }
Number of elements in Set A: 3
A not subset B
Set elements: { 1 , 2 , 3 , 4  }
Set elements: { 2 , 3  }
Set elements: { 1  }
Set is empty/deallocated after deinitialization

Process finished with exit code 0
```

And setB.txt is updated with the new integer contents:



This library works with all element types, but here I've showcased a few for reference of what the program expects in terms of user-defined functions for the function pointers etc.

Whenever the source/header files are altered and the library is run, our .dll file updates and so do the respective object files, indicating a successful library.

GitLab

https://gitlab.com/miguel.baldacchino.23/cproject/-/tree/main?ref_type=heads

77 Commits, Nov 27th 2023 - Jan 19th 2024

References

https://www.um.edu.mt/vle/course/view.php?id=75772 Ie. VLE
Notes by Mark Vella

https://www.w3schools.com/ Ie.
w3schools

https://www.geeksforgeeks.org/ Ie.
Geeks for Geeks

C Primer Plus (6th edition). Stephen Prata. Addison-Wesley, 2013.

https://stackoverflow.com/
Ie. Stack overflow