MANGALAYATAN UNIVERSITY
Learn Today to Lead Tomorrow
NAAC GRADE A+ Accredited University

BACHELOR OF COMPUTER APPLICATION (BCA)

CSB-1112 (Problem Solving Using C)

Block 3: Advanced C Programming Concepts

Unit 12: Error Handling and Debugging Techniques

Structure	
12.0	Introduction
12.1	Objectives
12.2	Error Handling in 'C'
12.3	Different types of errors exit in 'C'
12.4	The Perror () Function
12.5	The Sterror () Function
12.6	Uses of Ferror Function
12.7	The uses of clearer () Function
12.8	Divide by Zero Errors
12.9	Debugging tools and techniques
12.10	Program Exit Status
12.11	Conclusion
12.12	Questions
12.13	References

12.0 Introduction

This unit discusses as such, C programming does not provide direct support for error handling but being a sytem programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and set an error code errno. It is set as a global variable and indicates an error occurred during any function call. You can find various error codes defined in header file. So a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice to set errno to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program.

This unit will be discussing the error handling system process of language C.

12.1 Objectives

After completing this unit, you will be able to:

- ‘C’ programming error handling is provided with NSError class available in Foundation framework.
- An NSError object encapsulates richer and more extensible error information than is possible using only an error code or error string.
- An NSError object are an error domain (represented by a string)
- A domain-specific error code and a user info dictionary containing application specific information.

12.2 Error Handling in ‘C’

Error handling is a big part of writing software, and when it is done poorly, the software becomes difficult to extend and to maintain. Programming languages like C++ or Java provide an exceptions and destructors that make error handling easier. Such mechanisms are not natively available for C, and literature on good error handling in C is widely scattered over the internet.

Errors are the problems or the faults that occur in the program, which makes the behavior of the program abnormal, and experienced developers can also make these faults. Programming errors are also known as the bugs or faults, and the process of removing these bugs is known as **debugging**.

12.2.1 NSError

C programs use NSError objects to convey information about runtime errors that users need to be informed about. In most cases, a program displays this error information in a dialog or sheet. But it may also interpret the information and either ask the user to attempt to recover from the error or attempt to correct the error on its own.

NSError Object consists of –

Domain: The Error domain can be one of the predefined NSError domains an arbitrary string describing a custom domain and domain must not be nil.

Code: The error code for the error.

User Info: The User Info dictionary for the error and user Info may be nil.

Example:

```
NSString *domain = @"com.MyCompany.MyApplication.ErrorDomain";

NSString *desc = NSLocalizedString(@"Unable to complete the process", @"");

NSDictionary *userInfo = @{ NSLocalizedDescriptionKey : desc };

NSError *error = [NSError errorWithDomain:domain code: 101 userInfo:userInfo];
```

12.2.2 Pattern for Error Handling

A collected knowledge on good error handling in the form of C error-handling patterns and a running example that applies the patterns. The patterns provide good practice design decisions and elaborate on when to apply them and which consequences they bring. For a programmer, these patterns remove the burden of making many fine-grained decisions. Instead, a programmer can rely on the knowledge presented in these patterns and use them as a starting point to write good code.

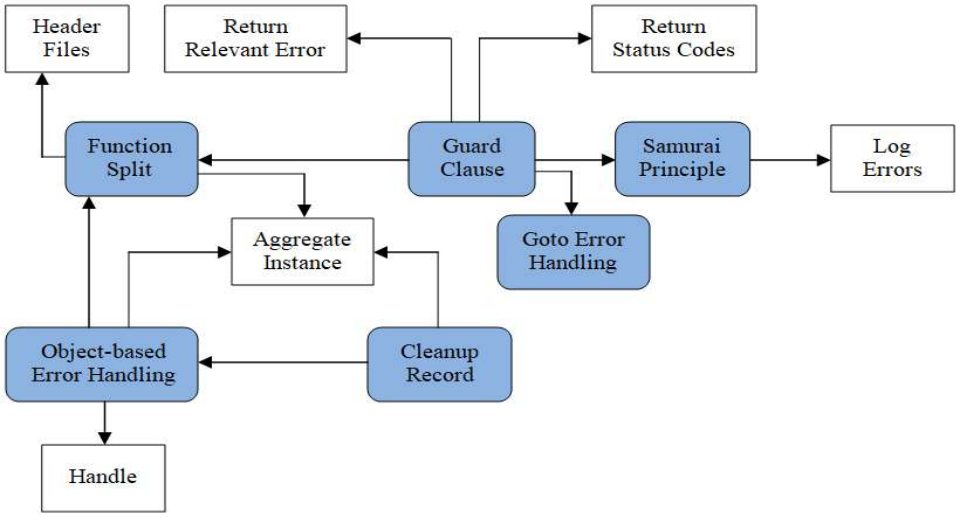

Figure: An overview of the patterns and their relationships

Pattern_Name	Description
Function Split	The function has several responsibilities, which makes the function hard to read and maintain. Therefore, split it up. Take a part of a function that seems useful on its own, create a new function with that, and call that function.
Guard Clause	The function is hard to read and maintain because it mixes pre-condition checks with the main program logic of the function. Therefore, check whether you have mandatory pre-conditions and immediately return from the function if these pre-conditions are not met.
Samurai Principle	When returning error information, you assume that the caller checks for this information. However, the caller can simply omit this check and the error might go unnoticed. Therefore, return from a function victorious or not at all. If there is a situation for which you know that an error cannot be handled, then abort the program.
Goto Error Handling	Code gets difficult to read and maintain if it acquires and cleans up multiple resources at different places within a function. Therefore, have all resource cleanup and error handling at the end of the function. If a resource cannot be acquired, use the goto statement to jump to the resource cleanup code.
Cleanup Record	It is difficult to make a piece of code easy to read and maintain if this code acquires and cleans up multiple resources, particularly if those resources depend on one another. Therefore, call resource acquisition functions as long as they succeed, and store which functions require cleanup. Call the cleanup functions depending on these stored values.
Object-Based Error Handling	Having multiple responsibilities in one function, such as resource acquisition, resource cleanup, and usage of that resource, makes that code difficult to implement, read, maintain, and test. Therefore, put initialization and cleanup into separate functions, similar to the concept of constructors and destructors in object-oriented programming.

12.2.3 Errno Values and Meaning

errno is a global variable indicating the error occurred during any function call and it is defined inside <errno.h> header file. When a function is called in C, a variable named errno is automatically assigned a code (value) which can be used to identify the type of error that has been encountered. Different codes values for errno mean different types of errors.

Error value	Error
1	Operation not permitted
2	No such file or directory
3	No such process
4	Interrupted system call
5	I/O error
6	No such device or address
7	The argument list is too long
8	Exec format error
9	Bad file number
10	No child processes
11	Try again
12	Out of memory
13	Permission denied

Example:

```
 #include <errno.h>
#include <stdio.h>
                int main()
  {
                 // If a file is opened which does not exist,
                // then it will be an error and corresponding
                // errno value will be set
FILE* fp;
                // opening a file which does not exist
                fp = fopen("Error Handling.txt", "r");
                printf("Value of errno: %d\n", errno);
                return 0;
  }
```

12.3 Different types of errors exit in 'C'

These errors are detected either during the time of compilation or execution. Thus, the errors must be removed from the program for the successful execution of the program.

There are mainly five types of errors exist in C programming:

- Syntax error
- Run-time error
- Linker error
- Logical error
- Semantic error

Syntax error:

Syntax errors are also known as the compilation errors as they occurred at the compilation time, or we can say that the syntax errors are thrown by the compilers. These errors are mainly occurred due to the mistakes while typing or do not follow the syntax of the specified programming language. These mistakes are generally made by beginners only because they are new to the language. These errors can be easily debugged or corrected.

The syntax errors are:

- If miss the parenthesis (}) while writing the code.
- Displaying the value of a variable without its declaration.
- If miss the semicolon (;) at the end of the statement.

Example:

```
#include <stdio.h>
int main()
{
   a = 10;
   printf("The value of a is : %d", a);
  return 0;
}
```

The following result:

```
main.c: In function 'main':
main.c:13:5: error: 'a' undeclared (first use in this function)
     a =  10;
     ^
main.c:13:5: note: each undeclared identifier is reported only once for each function it appears in
```

Example:

```
#include <stdio.h>
int main()
{
   a = 2;
   if(.)    // syntax error
   printf("a is greater than 1");
  return 0;
  }
```

The following result:

```
main.c: In function 'main':
main.c:15:6: error: expected expression before '.' token
   if(.)
      ^
```

Run-time error:

Sometimes the errors exist during the execution-time even after the successful compilation known as run-time errors. When the program is running, and it is not able to perform the operation is the main cause of the run-time error. The division by zero is the common example of the run-time error. These errors are very difficult to find, as the compiler does not point to these errors.

```
#include <stdio.h>
int main()
{
   int a=2;
   int b=2/0;
   printf("The value of b is : %d", b);
   return 0;
}
```

The following result:

```
main.c:14:12: warning: division by zero [-Wdiv-by-zero]
Floating point exception


...Program finished with exit code 136
Press ENTER to exit console.
```

Linker error:

Linker errors are mainly generated when the executable file of the program is not created. This can be happened either due to the wrong function prototyping or usage of the wrong header file. For example, the **main.c** file contains the **sub()** function whose declaration and definition is done in some other file such as **func.c**. During the compilation, the compiler finds the **sub()** function in **func.c** file, so it generates two object files, i.e., **main.o** and **func.o**. At the execution time, if the definition of **sub()** function is not found in the **func.o** file, then the linker error will be thrown. The most common linker error that occurs is that we use **Main()** instead of **main()**.

```
#include <stdio.h>
int Main()
{
   int a=78;
   printf("The value of a is : %d", a);
   return 0;
 }
```

The following result:

```
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

Logical error:

The logical error is an error that leads to an undesired output. These errors produce the incorrect output, but they are error-free, known as logical errors. These types of mistakes are mainly done by beginners. The occurrence of these errors mainly depends upon the logical thinking of the developer. If the programmers sound logically good, then there will be fewer chances of these errors.

```
#include <stdio.h>
int main()
{
  int sum=0; // variable initialization
  int k=1;
  for(int i=1;i<=10;i++); // logical error, as we put the semicolon after loop
  {
     sum=sum+k;
     k++;
  }
printf("The  value of sum is %d", sum);
   return 0;
}
```

The following result:

```
The  value of sum is 1

...Program finished with exit code 0
Press ENTER to exit console.
```

Semantic error:

Semantic errors are the errors that occurred when the statements are not understandable by the compiler.

The following can be the cases for the semantic error:

Use of a un-initialized variable.

```
int i;
i=i+2;
```

Type compatibility

```
int b = "Error Handling";
```

Errors in expressions

```
int a, b, c;
a+b = c;
Array index out of bound
int a[10];
a[10] = 34;
#include <stdio.h>
int main()
{
int a,b,c;
a=2;
b=3;
c=1;
a+b=c;          // semantic error
return 0;
}
```

The following result:

```
main.c: In function 'main':
main.c:17:6: error: lvalue required as left operand of assignment
   a+b=c;
      ^
```

12.4 The Perror () Function

The **perror()** function is used to show the error description. It displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.

Syntax

```
void perror(const char *str);
```

Parameters

str: It is a string containing a custom message to be printed before the error message itself.

Example

```
// C implementation to see how perror() function is used to
// print the error messages.
#include <errno.h>
#include <stdio.h>
#include <string.h>
int main()
```

```
{
   FILE* fp;
   // If a file is opened which does not exist,
   // then it will be an error and corresponding
   // errno value will be set
   fp = fopen(" Error Handling.txt ", "r");
   // opening a file which does
   // not exist.
   printf("Value of errno: %d\n ", errno);
   perror("Message from perror");
   return 0;
}
```

The following result:

Value of errno: 2

Message from perror: No such file or directory

12.5 The Sterror () Function

The **strerror()** function is also used to show the error description. This function returns a pointer to the textual representation of the current errno value.

Syntax

```
char *strerror(int errnum);
```

Parameters

errnum: It is the error number (errno).

Example:

```
// C implementation to see how strerror() function is used
// to print the error messages.
#include <errno.h>
#include <stdio.h>
#include <string.h>
 int main()
{
   FILE* fp;
  // If a file is opened which does not exist,
 // then it will be an error and corresponding
// errno value will be set
```

```
    fp = fopen(" Error Handling.txt ", "r");
 // opening a file which does
 // not exist.
    printf("Value of errno: %d\n", errno);
    printf("The error message is : %s\n", strerror(errno));
    return 0;
}
```

The following result:

Value of errno: 2

The error message is: No such file or directory

12.6 Uses of Ferror Function

The **ferror()** function is used to check whether an error occurred during a file operation.

Syntax

int **ferror**(FILE *stream);

Parameters

stream: It is the pointer that points to the FILE for which we want to check the error.

Return Value

It returns a non-zero value if an error occurred, otherwise it returns 0.

Example

```
// C program to demonstrate the ferror() function
#include <stdio.h>
  int main()
{
    // Open the file in read mode
    FILE* file = fopen("nonexistent_file.txt", "r");
    if (file == NULL)
{
        // Print an error message
        // if file opening fails
        perror("Error opening file");
        // Return with non-zero exit status to
        // indicate an error
        return 1;
    }
```

```
  int c;
 // Process the character
 // Add your code here to perform operations on each
 // character read from the file
 while ((c = fgetc(file)) != EOF) {
 }
  if (ferror(file)) {
    // Print an error message if an error occurred
    // during file reading
    printf(
       "An error occurred while reading the file.\n");
 }
 else {
    // Print success message if file reading completed
    // without errors
    printf("File read successfully.\n");
 }
 // Close the file
 fclose(file);
 // Return with zero exit status to indicate successful
 // execution
 return 0;
}
```

The following result:

Error opening file: No such file or directory

12.7 The uses of clearer () Function

The **clearerr()** function is used to clear both end-of-file and error indicators for a file stream.

Syntax

void **clearerr**(FILE *stream);

Parameters

stream: It is the pointer that points to the FILE for which we want to check the error.

Example

```
#include <stdio.h>
  int main()
```

```
{
    FILE* file = fopen("file.txt", "r");
    // Open the file in read mode
    if (file == NULL)
{
    // Print an error message
    // if file opening fails
        perror("Error opening file");
    // Return with non-zero exit status to
    // indicate an error
        return 1;
    }
    // Perform file operations
    // Add your code here to perform operations on the
    // opened file
      if (ferror(file)) {
    // Print an error message
    // if an error occurred
    // during file operations
        printf("An error occurred while performing file "
               "operations.\n");
    }
    // Clear the error indicators for the
    // file stream
    clearerr(file);
  // Continue with file operations
  // Add your code here to continue with further
  // operations on the file
  // Close the file
     fclose(file);
     return 0;
}
```

12.8 Divide by Zero Errors

A common pitfall made by C programmers is not checking if a divisor is zero before a division command. Division by zero leads to undefined behavior, there is no C language construct that can do anything about it. Your best bet is to not divide by zero in the first place, by checking the denominator.

Example:

```
// C program to check and rectify
// divide by zero condition
#include <stdio.h>
#include <stdlib.h>
 void function(int);
 int main()
{
   int x = 0;
   function(x);
   return 0;
}
 void function(int x)
{
   float fx;
   if (x == 0) {
   printf("Division by Zero is not allowed");
   fprintf(stderr, "Division by zero! Exiting...\n");
   exit(EXIT_FAILURE);
   }
   else
{
   fx = 10 / x;
   printf("f(x) is: %.5f", fx);
}
}
```

The following result:

Division by zero! Exiting ….

12.9 Debugging tools and techniques

Debugging is a critical part of the software development process, and in C programming, there are various tools and techniques available to help identify and resolve issues in code. Here are some commonly used debugging tools and techniques:

Debugging Tools

GDB (GNU Debugger):

- GDB is a powerful debugger for C (and C++) that allows you to see what is happening inside your program while it runs or what it was doing when it crashed.
- Key features include setting breakpoints, stepping through code, inspecting variables, and changing variable values at runtime.

Usage Example:

```
gcc -g -o myprogram myprogram.c   # Compile with debugging information
gdb ./myprogram                   # Run GDB on the compiled program
```

Common GDB commands:

```
break main          # Set a breakpoint at the beginning of main function
run                 # Start the program
next                # Step to the next line of code
print variable_name # Print the value of a variable
continue            # Continue running the program until the next breakpoint
```

Valgrind:

- Valgrind is a tool for memory debugging, memory leak detection, and profiling.
- It can detect memory management problems such as memory leaks, invalid memory access, and improper use of uninitialized memory.

Usage Example:

```
valgrind --leak-check=full ./myprogram
```

LLDB:

- LLDB is a debugger that is part of the LLVM project and is similar to GDB. It provides an intuitive interface and powerful features for debugging C programs.

Usage Example:

```
clang -g -o myprogram myprogram.c   # Compile with debugging information
lldb ./myprogram                    # Run LLDB on the compiled program
```

❖ **IDE Debuggers:**

- Integrated Development Environments (IDEs) such as Eclipse, Code::Blocks, and Visual Studio include built-in debuggers that provide graphical interfaces for setting breakpoints, stepping through code, and inspecting variables.

❖ **Debugging Techniques**

1. **Print Statements:**
   - Using printf to output the values of variables and the flow of execution is a simple yet effective way to understand what your program is doing.
   - Insert printf statements at critical points in your code to trace variable values and program execution.

2. **Binary Search Debugging:**
   - If the bug's location is unknown, divide the code into sections and place printf statements or breakpoints halfway through. By checking which section contains the issue, you can iteratively narrow down the problem's location.
3. **Code Review:**
   - Reviewing code manually or with peers can help identify logical errors, improper use of variables, and other issues that might not be caught by automated tools.
4. **Static Code Analysis:**
   - Tools like cppcheck and splint analyze your source code without executing it, identifying potential bugs, undefined behaviors, and coding standard violations.

   Usage Example:

   cppcheck myprogram.c

5. **Unit Testing:**
   - Writing unit tests for individual functions helps ensure they work correctly. Frameworks like CUnit or Unity can automate these tests and catch bugs early in development.

   Example with Unity:

```
#include "unity.h"
void test_function_should_return_42(void) {
    TEST_ASSERT_EQUAL(42, my_function());
}
int main(void) {
    UNITY_BEGIN();
    RUN_TEST(test_function_should_return_42);
    return UNITY_END();
}
```

6. **Memory Sanitizers:**
   - Memory sanitizers, such as AddressSanitizer (ASan), are tools to detect memory errors at runtime, such as buffer overflows and use-after-free errors.

   Usage Example with GCC:

   gcc -fsanitize=address -o myprogram myprogram.c

   ./myprogram

7. **Log Files:**
   - Writing logs to a file can be useful for long-running programs or when running in environments where console output is not feasible.
   - Use logging libraries or simple file I/O to create logs with timestamps and variable values.
8. **Conditional Compilation:**
   - Use preprocessor directives to include or exclude debug code without modifying the main source code.

   Example:

```
#ifdef DEBUG
#define DEBUG_PRINT(x) printf x
#else
#define DEBUG_PRINT(x)
#endif
DEBUG_PRINT(("Variable value: %d\n", var));
```

Example Debugging Session with GDB

Here's an example of a debugging session using GDB to find and fix a segmentation fault:

- **Compile the Program:**

  gcc -g -o myprogram myprogram.c
- **Run GDB:**

  gdb ./myprogram
- **Set a Breakpoint at the Beginning of main:**

  (gdb) break main
- **Run the Program:**

  (gdb) run
- **Step Through the Code:**

  (gdb) next
- **Inspect Variables:**

  (gdb) print variable_name
- **Continue Execution Until the Program Crashes:**

  (gdb) continue
- **Analyze the Cause of the Segmentation Fault:**

  (gdb) backtrace

12.10 Program Exit Status

Exit status is the value returned by the program after its execution is completed which tells the status of the execution of the program.

The C standard specifies two constants: **EXIT_SUCCESS** and **EXIT_FAILURE**, that may be passed to exit() to indicate successful or unsuccessful termination, respectively. These are macros defined in <stdlib.h> header file.

Example:

```
// C implementation which shows the
// use of EXIT_SUCCESS and EXIT_FAILURE.
```

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
 int main()
{
 FILE* fp;
 fp = fopen("filedoesnotexist.txt", "rb");
 if (fp == NULL)
{
 printf("Value of errno: %d\n", errno);
 printf("Error opening the file: %s\n",
 strerror(errno));
 perror("Error printed by perror");
 exit(EXIT_FAILURE);
 printf("I will not be printed\n");
  }
 else
{
 fclose(fp);
  exit(EXIT_SUCCESS);
  printf("I will not be printed\n");
  }
  return 0;
  }
```

The following result:

```
Value of errno: 2
Error opening the file: No such file or directory
Error printed by perror: No such file or directory
```

12.11 Conclusion

This chapter showed you how to perform error handling in C. Function Split tells you to split your functions into smaller parts to make error handling of these parts easier. A Guard Clause for your functions checks pre-conditions of your function and returns immediately if they are not met. This leaves fewer error-handling obligations for the rest of that function. Instead of returning from the function, you could also abort the program, adhering to the Samurai Principle. When it comes to more complex error handling

a ”particularly in combination with acquiring and releasing resources a ”you have several options. Goto Error Handling makes it possible to jump forward in your function to an error-handling section. Instead of jumping, Cleanup Record stores the info, which resources require cleanup, and performs it by the end of the function. A method of resource acquisition that is closer to object-oriented programming is Object-Based Error Handling, which uses separate initialization and cleanup functions similar to the concept of constructors and destructors.

With these error-handling patterns in your repertoire, you now have the skill to write small programs that handle error situations in a way that ensures the code stays maintainable.

12.12 Questions

1. **What is the purpose of error handling in C programming?**

Ans.1 Error handling in C programming is essential to ensure that a program can deal with unexpected situations gracefully without crashing or producing incorrect results. The main purposes of error handling are:

- **Detecting Errors:** Identifying when an error occurs during program execution.
- **Managing Errors:** Handling the error in a controlled manner, such as logging the error, cleaning up resources, or providing feedback to the user.
- **Improving Reliability:** Making the program more robust and less likely to fail unexpectedly.
- **Maintaining Program State:** Ensuring that the program can recover or safely terminate after an error occurs, preserving data integrity and avoiding resource leaks.

2. **Describe the use of the errno variable in C. How is it typically used?**

Ans.2 The **errno** variable in C is used by system calls and some library functions to indicate what error occurred during their execution. It is defined in the **<errno.h>** header file and is set to a positive integer value representing the error code when an error occurs.

Typical Usage:

1. **Include the Header File:**
   #include <errno.h>
2. **Check for Errors:** After calling a function that can set errno, check if the return value indicates an error.
3. **Access the Error Code:** If an error occurred, **errno** will contain the error code. Use **strerror()** or **perror()** to print a human-readable error message.

Example:

```
        #include <stdio.h>
#include <errno.h>
#include <string.h>
int main() {
   FILE *file = fopen("nonexistent.txt", "r");
   if (!file) {
      printf("Error opening file: %s\n", strerror(errno));
      return 1;
   }
   // Use the file
   fclose(file);
```

```
    return 0;
}
```

3. What are setjmp and longjmp used for in C? Provide an example.

Ans.3 **setjmp** and **longjmp** are used for non-local jumps in C, which allow control to jump back to a specific point in the program from a deeply nested function call. This is useful for error handling in complex programs where errors need to be handled at a higher level.

Usage:

a) **setjmp:**
   - Saves the current execution state (stack context) in a **jmp_buf** variable.
   - Returns 0 initially and returns a non-zero value when returning from **longjmp**.

b) **longjmp:**
   - Restores the execution state saved by **setjmp** and jumps back to that point.
   - Takes two arguments: the **jmp_buf** variable and an integer value that **setjmp** will return.

Example:

```
        #include <stdio.h>
#include <setjmp.h>
jmp_buf buf;
void error_function() {
   printf("Error occurred!\n");
   longjmp(buf, 1); // Jump back to the point where setjmp was called
}
int main() {
   if (setjmp(buf)) {
      printf("Returned from a long jump\n");
      return 1; // Handle error
   } else {
      printf("Calling error_function\n");
      error_function(); // This will trigger the long jump
   }
   return 0;
}
```

In this example, setjmp sets a recovery point in main. When error_function calls longjmp, the control returns to the point where setjmp was called, and setjmp returns 1, allowing the program to handle the error.

4. Explain the purpose and usage of assert in C?

Ans.4 The **assert** macro in C is used to perform runtime checks during the development phase. It evaluates an expression and, if the expression is false, it prints an error message and terminates the program. This helps catch logical errors and incorrect assumptions in the code early.

Usage:

a) **Include the Header File:**
   - #include <assert.h>

b) Use assert to Check Assumptions:

- Place **assert** statements at critical points in the code where you want to verify assumptions.
- If the condition is false, the program will print an error message and terminate.

Example:

```
int main() {
   int x = 0;
   assert(x != 0 && "x should not be zero");
   printf("This line will not be executed if x is zero\n");
   return 0;
}
```

5. What is memory leak and how can it be detected and fixed in C?

Ans.5 A memory leak occurs when a program allocates memory dynamically using functions like **malloc** or **calloc** but fails to **free** it using free after the memory is no longer needed. This results in a gradual loss of available memory, which can lead to reduced performance or program crashes.

Detection:

1. **Code Review:**
   - Manually reviewing the code to ensure every allocated memory is freed appropriately.
2. **Valgrind:**
   - A tool to detect memory leaks by running the program and reporting any unfreed memory.
     Example:
     valgrind --leak-check=full ./myprogram

Fixing:

1. **Freeing Memory:**
   - Ensure that every dynamically allocated memory block is freed when it is no longer needed.
     Example:
     int *ptr = (int*)malloc(sizeof(int) * 10);
     // Use the memory
     free(ptr); // Free the memory
2. **Avoiding Dangling Pointers:**
   - After freeing memory, set the pointer to **NULL** to avoid accidental dereference of invalid memory.
     Example:
     free(ptr);
     ptr = NULL;

12.13 References

- Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language (2nd ed.). Prentice Hall.
- King, K. N. (2008). C Programming: A Modern Approach (2nd ed.). W. W. Norton & Company.
- Prata, S. (2014). C Primer Plus (6th ed.). Addison-Wesley Professional.
- Deitel, P., & Deitel, H. (2016). C How to Program (8th ed.). Pearson.

***