KYALO BENEDICT KISWILI
SCT 222-0149/2023
ASSIGNMENT

1.

a) Compiler - This is a program that translates a source code written in a programming language into machine code that the computer can execute.
b) Source Code - This is a text written by a programmer in a human readable format using a programming language that a computer cannot execute directly.
c) Object Code - This is a code that is generated by the compiler when it translates source code written in C into machine code that can be executed by a computer
d) Linker - This is a tool that takes the object code generated by the compiler and combines it with other object codes to create a single executable file that can be run on the computer

2.
Example:

```
#include <stdio.h>

int main() {
   int num1, num2

   printf("Enter the first number: ");
   scanf("%d", &num1);
   printf("Enter the second number: ");
   scanf("%d", &num2);
    sum = num1 + num2;

   printf("The sum is: %d\n", sum);


   return 0;
   }
```

Steps involved in the compilation process

1. Preprocessing: In this stage, the preprocessor reads the source code and performs various tasks such as expanding macros, including header files, and removing comments. The preprocessor outputs a modified version of the source code, which is then fed into the compiler.

2. Compilation: The compiler takes the modified source code generated by the preprocessor and converts it into assembly code. The compiler checks for syntax errors and ensures that the code follows the rules of the C programming language. If there are any errors, the compiler will output error messages and stop the compilation process.

3. Assembly: The assembly code generated by the compiler is then fed into an assembler, which converts it into machine code. The assembler maps the assembly code instructions to their corresponding machine code instructions.

4. Linking: The machine code generated by the assembler is then fed into a linker, which links the machine code to any required libraries and creates an executable file. The linker also resolves any symbolic references in the code, such as function calls, to their corresponding memory addresses.

5. Execution: The executable file generated by the linker can now be run on the computer. When the program is run, the operating system loads the executable file into memory, and the CPU executes the instructions in the program.

3. **Differences between a compiler and interpretor**

1. **Compilation**: A compiler is responsible for converting source code ,the code written by the programmer, into machine code ,the code that the computer can understand and execute. This process is called compilation. An interpreter, on the other hand, does not compile the source code into machine code. Instead, it interprets the source code line by line and executes it directly.
2. **Speed:** Compiled code is generally faster than interpreted code because the compiler optimizes the code for the specific hardware it will be running on. Interpreted code, on the other hand, is slower because it has to translate the code into machine code at runtime.
3. **Portability**: Compiled code is less portable than interpreted code. This is because compiled code is specific to the hardware and operating system it was compiled for. Interpreted code, on the other hand, can be run on any platform that has an interpreter for the programming language.
4. **Memory usage**: Compiled code typically uses less memory than interpreted code because the compiler optimizes the code to use the minimum amount of memory necessary. Interpreted code, on the other hand, can use more memory because it has to store the entire program in memory while it is running.
5. **Debugging**: Compiled code can be more difficult to debug than interpreted code because the compiler hides the details of the machine code from the programmer. Interpreted code, on the other hand, allows the programmer to see the machine code and debug it more easily.
6. **Error handling**: Compiled code can handle errors more gracefully than interpreted code. This is because the compiler can generate error messages and exit codes that the programmer can use

to handle errors. Interpreted code, on the other hand, may not be able to handle errors as gracefully because it does not have the same level of control over the program's execution.
7. **Security**: Compiled code is generally more secure than interpreted code because it is harder to reverse engineer or modify. Interpreted code, on the other hand, can be easier to reverse engineer or modify because it is in plain text.
8. **Development environment:** Compiled code typically requires a more structured development environment than interpreted code. This is because the compiler needs to know the exact syntax and semantics of the programming language in order to generate machine code. Interpreted code, on the other hand, can be developed in a more flexible environment because the interpreter can handle variations in syntax and semantics.

4.**Categories of operators available in C and their operators.**
1. **Arithmetic operators:** These operators are used for performing arithmetic operations such as addition, subtraction, multiplication, and division. Examples include:
    * `+` (addition)
    * `-` (subtraction)
    * `*` (multiplication)
    * `/` (division)
    * `%` (modulus)
2. **Assignment operators**: These operators are used to assign a value to a variable. Examples include:
    * `=` (assignment)
    * `+=` (addition assignment)
    * `-=` (subtraction assignment)
    * `*=` (multiplication assignment)
    * `/=` (division assignment)
    * `%=` (modulus assignment)
3. **Comparison operators**: These operators are used to compare values and return a boolean value (true or false). Examples include:
    * `==` (equal to)
    * `!=` (not equal to)
    * `>` (greater than)
    * `<` (less than)
    * `>=` (greater than or equal to)
    * `<=` (less than or equal to)
4. **Logical operators**: These operators are used to combine multiple comparison operators to create more complex conditions. Examples include:
    * `&&` (logical AND)
    * `||` (logical OR)
    * `!` (logical NOT)

5. **Bitwise operators**: These operators are used to perform operations on individual bits within a binary number. Examples include:
    * `&` (bitwise AND)
    * `|` (bitwise OR)
    * `^` (bitwise XOR)
    * `~` (bitwise NOT)
6. **Membership operators**: These operators are used to test whether a value is a member of a particular set or range. Examples include:
    * `in` (membership operator)
    * `not in` (non-membership operator)
7. **Pointer operators**: These operators are used to work with memory addresses and pointers. Examples include:
    * `&` (address-of operator)
    * `*` (dereference operator)