**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **19-06-2020** | **Name:** | **Dhanya Shetty** |
| **Course:** | **C PROGRAMMING** | **USN:** | **4AL17EC026** |
| **Topic:** | **1.STRUCTURES & UNIONS**  **2.MEMORY MANAGEMENT**  **3.FILES & ERROR HANDLING**  **4.THE PREPROCESSOR** | **Semester & Section:** | **6th A** |
| **Github Repository:** | **Dhanya Shetty\_026** |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| C:\Users\Hp\Desktop\certificates\19june1111.PNG  C:\Users\Hp\Desktop\certificates\19june2222.PNG  **C:\Users\Hp\Desktop\certificates\19june333.PNG**  **C:\Users\Hp\Desktop\certificates\19june4444.PNG**  **C:\Users\Hp\Desktop\certificates\19june5555.PNG**  **C:\Users\Hp\Desktop\certificates\19june6666.PNG**  **C Programming :**  **C** is highly portable and is **used for** scripting system applications which form a major part of Windows, UNIX, and Linux operating system. **C** is a general-purpose **programming language** and can efficiently work on enterprise applications, games, graphics, and applications requiring calculations, etc.  **Benefits of C language over other programming languages :**  [C](https://www.geeksforgeeks.org/c/) is a middle-level programming language developed by Dennis Ritchie during the early 1970s while working at AT&T Bell Labs in the USA. The objective of its development was in the context of the re-design of the UNIX operating system to enable it to be used on multiple computers.  Earlier the language B was now used for improving the UNIX system. Being a high-level language, B allowed much faster production of code than in assembly language. Still, B suffered from drawbacks as it did not understand data-types and did not provide the use of “structures”.  These drawbacks became the driving force for Ritchie for development of a new programming language called C. He kept most of language B’s syntax and added data-types and many other required changes. Eventually, C was developed during 1971-73, containing both high-level functionality and the detailed features required to program an operating system. Hence, many of the UNIX components including UNIX kernel itself were eventually rewritten in C.  **Structures :**  A **struct** in the [C programming language](https://en.wikipedia.org/wiki/C_programming_language) (and many derivatives) is a [composite data type](https://en.wikipedia.org/wiki/Composite_data_type) (or [record](https://en.wikipedia.org/wiki/Record_(computer_science))) declaration that defines a physically grouped list of variables under one name in a block of memory, allowing the different variables to be accessed via a single [pointer](https://en.wikipedia.org/wiki/Pointer_(computer_programming)) or by the struct declared name which returns the same address. The struct data type can contain other data types so is used for mixed-data-type records such as a hard-drive directory entry (file length, name, extension, physical address, etc.), or other mixed-type records (name, address, telephone, balance, etc.).  The C struct directly references a *contiguous block* of physical memory, usually delimited (sized) by word-length boundaries. It corresponds to the similarly named feature available in some [assemblers](https://en.wikipedia.org/wiki/Assembly_language) for Intel processors. Language implementations that could utilize half-word or byte boundaries (giving denser packing, using less memory) were considered advanced in the mid-1980s. Being a block of contiguous memory, each field within a struct is located at a certain fixed offset from the start.  Because the contents of a struct are stored in contiguous memory, the [size of](https://en.wikipedia.org/wiki/Sizeof) operator must be used to get the number of bytes needed to store a particular type of struct, just as it can be used for [primitives](https://en.wikipedia.org/wiki/Primitive_data_type). The alignment of particular fields in the struct (with respect to [word](https://en.wikipedia.org/wiki/Word_(computer_architecture)) boundaries) is implementation-specific and may include padding, although modern compilers typically support the #pragma pack directive, which changes the size in bytes used for alignment.[[1]](https://en.wikipedia.org/wiki/Struct_(C_programming_language)#cite_note-1)  In the [C++](https://en.wikipedia.org/wiki/C%2B%2B) language, a struct is identical to a [C++ class](https://en.wikipedia.org/wiki/C%2B%2B_classes) but has a different default visibility: class members are private by default, whereas struct members are public by default.  Declaration  The general syntax for a struct declaration in C is:  **struct** tag\_name {  type member1;  type member2;  };  **Unions :**  A **union** is a special data type available in **C** that allows to store different data types in the same memory location. You can define a **union** with many members, but only one member can contain a value at any given time. **Unions** provide an efficient way of using the same memory location for multiple-purpose.  We use the union keyword to define unions. Here's an example:  union car  {  char name[50];  int price;  };  The above code defines a derived type union car.  **Create union variables.**  union car  {  char name[50];  int price;  };  int main()  {  union car car1, car2, \*car3;  return 0;  }  Another way of creating union variables is:  union car  {  char name[50];  int price;  } car1, car2, \*car3;  **Difference between unions and structures**  Let's take an example to demonstrate the difference between unions and structures:  #include <stdio.h>  union unionJob  {  //defining a union  char name[32];  float salary;  int workerNo;  } uJob;  struct structJob  {  char name[32];  float salary;  int workerNo;  } sJob;  int main()  {  printf("size of union = %d bytes", sizeof(uJob));  printf("\nsize of structure = %d bytes", sizeof(sJob));  return 0;  }  **Output**  size of union = 32  size of structure = 40  **How are unions stored in memory?**  The size of a **union** is sufficient to contain the largest of its members. The value of at most one of the members can be **stored** in a **union** object at any time. ... The bold statement actually says that each member of the **union** has the same address, so they all "begin" at the same address.  **Memory Management :**  **C** dynamic **memory** allocation refers to performing manual **memory management** for dynamic **memory** allocation in the **C** programming language via a group of functions in the **C** standard library, namely malloc, realloc, calloc and free.  The [C++](https://en.wikipedia.org/wiki/C%2B%2B) programming language includes these functions; however, the operators [new and delete](https://en.wikipedia.org/wiki/New_and_delete_(C%2B%2B)) provide similar functionality and are recommended by that language's authors.Still, there are several situations in which using new/delete is not applicable, such as garbage collection code or performance-sensitive code, and a combination of malloc and placement new may be required instead of the higher-level new operator.  Many different implementations of the actual memory allocation mechanism, used by malloc, are available. Their performance varies in both execution time and required memory.  **Files and Error Handling :**  The **C** programming language provides perror () and strerror() **functions** which can be used to display the text message associated with errno. The perror() **function** displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.  **Functions of error handler :**  The tasks of the **Error Handling** process are to detect each **error**, report it to the user, and then make some recover strategy and implement them to handle **error**. During this whole process processing time of program should not be slow. An **Error** is the blank entries in the symbol table.  **The Preprocessor :**  In [computer science](https://en.wikipedia.org/wiki/Computer_science), a **pre-processor** is a [program](https://en.wikipedia.org/wiki/Computer_program) that processes its input data to produce output that is used as input to another program. The output is said to be a **pre-processed** form of the input data, which is often used by some subsequent programs like [compilers](https://en.wikipedia.org/wiki/Compiler). The amount and kind of processing done depends on the nature of the pre-processor; some pre-processors are only capable of performing relatively simple textual substitutions and [macro](https://en.wikipedia.org/wiki/Macro_(computer_science)) expansions, while others have the power of full-fledged [programming languages](https://en.wikipedia.org/wiki/Programming_language).  A common example from [computer programming](https://en.wikipedia.org/wiki/Computer_programming) is the processing performed on [source code](https://en.wikipedia.org/wiki/Source_code) before the next step of compilation. In some [computer languages](https://en.wikipedia.org/wiki/Computer_language) (e.g., [C](https://en.wikipedia.org/wiki/C_(programming_language)) and [PL/I](https://en.wikipedia.org/wiki/PL/I_(programming_language))) there is a phase of [translation](https://en.wikipedia.org/wiki/Compiler) known as *pre-processing*. It can also include macro processing, file inclusion and language extensions.  The most common example of this is the [C preprocessor](https://en.wikipedia.org/wiki/C_preprocessor), which takes lines beginning with '#' as [directives](https://en.wikipedia.org/wiki/Directive_(programming)). Because it knows nothing about the underlying language, its use has been criticized and many of its features built directly into other languages. For example, macros replaced with aggressive in lining and templates, includes with compile-time imports (this requires the preservation of type information in the object code, making this feature impossible to retrofit into a language); [conditional compilation](https://en.wikipedia.org/wiki/Conditional_compilation) is effectively accomplished with if-then-else and dead code elimination in some languages. However, a key point to remember is that all preprocessor directives should start on a new line.  The most common use of the **preprocessor** is to include header files. In C and C++, all symbols must be declared in a file before they can used. They don't always **need** to be defined\*, but the compiler needs to know they exist somewhere. ... The most common use of the **preprocessor** is to include header files. |
|  |

|  |
| --- |
|  |
|  |
|  |