Click to prove you're human

In C programming, a struct (or structure) is a collection of variables (can be of different types) under a single name. Define Structures Before you can create structure variables, you need to define its data type. To define a struct, the struct keyword is used. Syntax of struct struct structureName { dataType member1; dataType member2; ... }; For example, struct Person { char name[50]; int citNo; float salary; }; Here, a derived type struct Person is defined. Now, you can create variables of this type. Create struct Variables When a struct type is declared, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables. Here's how we create structure variables: struct Person { // code }; int main() { struct Person person1, person2, p[20]; return 0; } Another way of creating a struct variable is: struct Person { // code } person1, person2, p[20]; In both cases, person1 and person2 are struct Person variables p[] is a struct Person array of size 20. Access Members of a Structure There are two types of operators used for accessing members of a structure. . - Member operator -> - Structure pointer operator (will be discussed in the next tutorial) Suppose, you want to access the salary of person2. Here's how you can do it. person2.salary Example 1: C structs #include #include // create struct with person1 variable struct Person { char name[50]; int citNo; float salary; } person1; int main() { // assign value to name of person1 strcpy(person1.name, "George Orwell"); // assign values to other person1 variables person1.citNo = 1984; person1. salary = 2500; // print struct variables printf("Name: %s", person1.name); printf("Citizenship No.: %d", person1.citNo); printf("Salary: %.2f", person1.salary); return 0; } Output Name: George Orwell Citizenship No.: 1984 Salary: 2500.00 In this program, we have created a struct named Person. We have also created a variable of Person named person1. In main(), we have assigned values to the variables defined in Person for the person1 object. strcpy(person1.name, "George Orwell"); person1.citNo = 1984; person1. salary = 2500; Notice that we have used strcpy() function to assign the value to person1.name. This is because name is a char array (C-string) and we cannot use the assignment operator = with it after we have declared the string. Finally, we printed the data of person1. Keyword typedef We use the typedef keyword to create an alias name for data types. It is commonly used with structures to simplify the syntax of declaring variables. For example, let us look at the following code: struct Distance{ int feet; float inch; }; int main() { struct Distance d1, d2; } We can use typedef to write an equivalent code with a simplified syntax: typedef struct Distance { int feet; float inch; } distances; int main() { distances d1, d2; } Example 2: C typedef #include #include // struct with typedef person typedef struct Person { char name[50]; int citNo; float salary; } person; int main() { // create Person variable person p1; // assign value to name of p1 strcpy(p1.name, "George Orwell"); // assign values to other p1 variables p1.citNo = 1984; p1. salary = 2500; // print struct variables printf("Name: %s", p1.name); printf("Citizenship No.: %d", p1.citNo); printf("Salary: %.2f", p1.salary); return 0; } Output Name: George Orwell Citizenship No.: 1984 Salary: 2500.00 Here, we have used typedef with the Person structure to create an alias person. // struct with typedef person typedef struct Person { // code } person; Now, we can simply declare a Person variable using the person alias: // equivalent to struct Person p1 person p1; Nested Structures You can create structures within a structure in C programming. For example, struct complex { int imag; float real; }; struct number { struct complex comp; int integers; } num1, num2; Suppose, you want to set imag of num2 variable to 11. Here's how you can do it: num2.comp.imag = 11; Example 3: C Nested Structures #include struct complex { int imag; float real; }; struct number { struct complex comp; int integer; } num1; int main() { // initialize complex variables num1.comp.imag = 11; num1.comp.real = 5.25; // initialize number variable num1.integer = 6; // print struct variables printf("Imaginary Part: %d", num1.comp.imag); printf("Real Part: %.2f", num1.comp.real); printf("Integer: %d", num1.integer); return 0; } Output Imaginary Part: 11 Real Part: 5.25 Integer: 6 Why structs in C? Suppose you want to store information about a person: his/her name, citizenship number, and salary. You can create different variables name, citNo and salary to store this information. What if you need to store information of more than one person? Now, you need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2, etc. A better approach would be to have a collection of all related information under a single name Person structure and use it for every person. More on struct Structures and pointers Passing structures to a function Share — copy and redistribute the material in any medium or format for any purpose, even commercially. Adapt — remix, transform, and build upon the material for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow the license terms. Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits. You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation . No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material. Sign in to track your progress You have already completed these exercises! Do you want to take them again? × Close the exercise You completed the C Structures Exercises from W3Schools.com Share on: In C programming, both structures and unions are used to group different types of data under a single name, but they behave in different ways. The main difference lies in how they store data.The below table lists the primary differences between the C structures and unions:ParameterStructureUnionDefinitionA structure is a user-defined data type that groups different data types into a single entity.A union is a user-defined data type that allows storing different data types at the same memory location.KeywordThe keyword struct is used to define a structureThe keyword union is used to define a unionSizeThe size is the sum of the sizes of all members, with padding if necessary.The size is equal to the size of the largest member, with possible padding.Memory AllocationEach member within a structure is allocated unique storage area of location.Memory allocated is shared by individual members of union.Data OverlapNo data overlap as members are independent.Full data overlap as members shares the same memory.Accessing MembersIndividual member can be accessed at a time.Only one member can be accessed at a time.StructuresA structure in C is a collection of variables, possibly of different types, under a single name. Each member of the structure is allocated its own memory space, and the size of the structure is the sum of the sizes of all its members.Syntaxstruct name { member1 definition; member2 definition; ... memberN definition;};Example: C #include struct Student { char name[50]; int age; float grade; }; int main() { // Create a structure variable struct Student s1 = {"Geek", 20, 85.5}; // Access structure members printf("%s", s1.name); printf("%d", s1.age); printf("%.2f", s1.grade); printf("Size: %d bytes", sizeof(s1)); return 0; } OutputGeek 20 85.50 Size: 60 bytesExplanation: In this example, we create a structure Student to store a student's name, age, and grade. Each of the members (name, age, grade) is stored in its own separate memory location, and we access them individually. The size is also the size of all members combined plus structure padding.UnionsA union in C is similar to a structure, but with a key difference: all members of a union share the same memory location. This means only one member of the union can store a value at any given time. The size of a union is determined by the size of its largest member.Syntax:union name { member1 definition; member2 definition; ... memberN definition;};Example: C //Driver Code Starts #include //Driver Code Ends union Data { int i; double d; char c; }; int main() { // Create a union variable union Data data; // Store an integer in the union data.i = 100; printf("%d ", data.i); // Store a double in the union (this will // overwrite the integer value) data.d = 99.99; printf("%.2f ", data.d); // Store a character in the union (this will // overwrite the double value) data.c = 'A'; printf("%c ", data.c); printf("Size: %d", sizeof(data)); //Driver Code Starts return 0; } //Driver Code Ends Output100 99.99 A Size: 8Similarities Between Structure and UnionStructures and unions are also similar in some aspects listed below:Both are user-defined data types used to store data of different types as a single unit.Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.'.' operator or selection operator, which has one of the highest precedences, is used for accessing member variables inside both the user-defined datatypes. A data structure is a way to store data. We structure data in different ways depending on what data we have, and what we want to do with it. Family tree First, let's consider an example without computers in mind, just to get the idea. If we want to store data about people we are related to, we use a family tree as the data structure. We choose a family tree as the data structure because we have information about people we are related to and how they are related, and we want an overview so that we can easily find a specific family member, several generations back. With such a family tree data structure visually in front of you, it is easy to see, for example, who my mother's mother is—it is 'Emma,' right? But without the links from child to parents that this data structure provides, it would be difficult to determine how the individuals are related. Data structures give us the possibility to manage large amounts of data efficiently for uses such as large databases and internet indexing services. Data structures are essential ingredients in creating fast and powerful algorithms. They help in managing and organizing data, reduce complexity, and increase efficiency. In Computer Science there are two different kinds of data structures. Primitive Data Structures are basic data structures provided by programming languages to represent single values, such as integers, floating-point numbers, characters, and booleans. Abstract Data Structures are higher-level data structures that are built using primitive data types and provide more complex and specialized operations. Some common examples of abstract data structures include arrays, linked lists, stacks, queues, trees, and graphs. What are Algorithms? An algorithm is a set of step-by-step instructions to solve a given problem or achieve a specific goal. Pommes Frites Recipe A cooking recipe written on a piece of paper is an example of an algorithm, where the goal is to make a certain dinner. The steps needed to make a specific dinner are described exactly. When we talk about algorithms in Computer Science, the step-by-step instructions are written in a programming language, and instead of food ingredients, an algorithm uses data structures. Algorithms are fundamental to computer programming as they provide step-by-step instructions for executing tasks. An efficient algorithm can help us to find the solution we are looking for, and to transform a slow program into a faster one. By studying algorithms, developers can write better programs. Algorithm examples: Finding the fastest route in a GPS navigation system Navigating an airplane or a car (cruise control) Finding what users search for (search engine) Sorting, for example sorting movies by rating The algorithms we will look at in this tutorial are designed to solve specific problems, and are often made to work on specific data structures. For example, the 'Bubble Sort' algorithm is designed to sort values, and is made to work on arrays. Data Structures together with Algorithms Data structures and algorithms (DSA) go hand in hand. A data structure is not worth much if you cannot search through it or manipulate it efficiently using algorithms, and the algorithms in this tutorial are not worth much without a data structure to work on. DSA is about finding efficient ways to store and retrieve data, to perform operations on data, and to solve specific problems. By understanding DSA, you can: Decide which data structure or algorithm is best for a given situation. Make programs that run faster or use less memory. Understand how to approach complex problems and solve them in a systematic way. Where is Data Structures and Algorithms Needed? Data Structures and Algorithms (DSA) are used in virtually every software system, from operating systems to web applications: For managing large amounts of data, such as in a social network or a search engine. For scheduling tasks, to decide which task a computer should do first. For planning routes, like in a GPS system to find the shortest path from A to B. For optimizing processes, such as arranging tasks so they can be completed as quickly as possible. For solving complex problems: From finding the best way to pack a truck to making a computer 'learn' from data. DSA is fundamental in nearly every part of the software world: Operating Systems Database Systems Web Applications Machine Learning Video Games Cryptographic Systems Data Analysis Search Engines As we go along in this tutorial, new theoretical concepts and terminology (new words) will be needed so that we can better understand the data structures and algorithms we will be working on. These new words and concepts will be introduced and explained properly when they are needed, but here is a list of some key terms, just to get an overview of what is coming: Term Description Algorithm A set of step-by-step instructions to solve a specific problem. Data Structure A way of organizing data so it can be used efficiently. Common data structures include arrays, linked lists, and binary trees. Time Complexity A measure of the amount of time an algorithm takes to run, depending on the amount of data the algorithm is working on. Space Complexity A measure of the amount of memory an algorithm uses, depending on the amount of data the algorithm is working on. Big O Notation A mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. Used in this tutorial to describe the time complexity of an algorithm. Recursion A programming technique where a function calls itself. Divide and Conquer A method of solving complex problems by breaking them into smaller, more manageable sub-problems, solving the sub-problems, and combining the solutions. Recursion is often used when using this method in an algorithm. Brute Force A simple and straight forward way an algorithm can work by simply trying all possible solutions and then choosing the best one. Where to Start? In this tutorial, you will first learn about a data structure with matching algorithms, before moving on to the next data structure. Further into the tutorial the concepts become more complex, and it is therefore a good idea to learn DSA by doing the tutorial step-by-step from the start. And as mentioned on the previous page, you should be comfortable in at least one of the most common programming languages, like for example JavaScript, C or Python, before doing this tutorial. On the next page we will look at two different algorithms that prints out the first 100 Fibonacci numbers using only primitive data structures (two integer variables). One algorithm uses a loop, and one algorithm uses something called recursion. Click the 'Next' button to continue. Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a member of the structure. Unlike an array, a structure can contain many different data types (int, float, char, etc.). Create a Structure You can create a structure by using the struct keyword and declare each of its members inside curly braces: struct MyStructure { // Structure declaration int myNum; // Member (int variable) char myLetter; // Member (char variable)}; // End the structure with a semicolon To access the structure, you must create a variable of it. Use the struct keyword inside the main() method, followed by the name of the structure and then the name of the structure variable: Create a struct variable with the name "s1": struct myStructure { int myNum; char myLetter;};int main() { struct myStructure s1; return 0;} Access Structure Members To access members of a structure, use the dot syntax (.): // Create a structure called myStructurestruct myStructure { int myNum; char myLetter;};int main() { // Create a structure variable of myStructure called s1 struct myStructure s1; // Assign values to members of s1 s1.myNum = 13; s1.myLetter = 'B'; // Print values printf("My number: %d", s1.myNum); printf("My letter: %c", s1.myLetter); return 0;} Try it Yourself » Now you can easily create multiple structure variables with different values, using just one structure: // Create different struct variablesstruct myStructure s1;struct myStructure s2; // Assign values to different struct variabless1.myNum = 13;s1.myLetter = 'B'; s2.myNum = 20;s2.myLetter = 'C'; Try it Yourself » Remember that strings in C are actually an array of characters, and unfortunately, you can't assign a value to an array like this: struct myStructure { int myNum; char myLetter; char myString[30]; // String};int main() { struct myStructure s1; // Trying to assign a value to the string s1.myString = "Some text"; // Trying to print the value printf("My string: %s", s1.myString); return 0;} An error will occur: prog.c:12:15: error: assignment to expression with array type Try it Yourself » However, there is a solution for this! You can use the strcpy() function and assign the value to s1.myString, like this: struct myStructure { int myNum; char myLetter; char myString[30]; // String};int main() { struct myStructure s1; // Assign a value to the string using the strcpy function strcpy(s1.myString, "Some text"); // Print the value printf("My string: %s", s1.myString); return 0;} Result: My string: Some text Try it Yourself » Simpler Syntax You can also assign values to members of a structure variable at declaration time, in a single line. Just insert the values in a comma-separated list inside curly braces {}. Note that you don't have to use the strcpy() function for string values with this technique: // Create a structurestruct myStructure { int myNum; char myLetter; char myString[30];};int main() { // Create a structure variable and assign values to it struct myStructure s1 = {13, 'B', "Some text"}; // Print values printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString); return 0;} Try it Yourself » Note: The order of the inserted values must match the order of the variable types declared in the structure (13 for int, 'B' for char, etc). Copy Structures You can also assign one structure to another. In the following example, the values of s1 are copied to s2: struct myStructure s1 = {13, 'B', "Some text"};struct myStructure s2; s2 = s1; Try it Yourself » Modify Values If you want to change/modify a value, you can use the dot syntax (.). And to modify a string value, the strcpy() function is useful again: struct myStructure { int myNum; char myLetter; char myString[30];};int main() { // Create a structure variable and assign values to it struct myStructure s1 = {13, 'B', "Some text"}; // Modify values s1.myNum = 30; s1.myLetter = 'C'; strcpy(s1.myString, "Something else"); // Print values printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString); return 0;} Try it Yourself » Modifying values are especially useful when you copy structure values: // Create a structure variable and assign values to itstruct myStructure s1 = {13, 'B', "Some text"};// Create another structure variablestruct myStructure s2;// Copy s1 values to s2s2 = s1;// Change s2 valuess2.myNum = 30;s2.myLetter = 'C';strcpy(s2.myString, "Something else");// Print valuesprintf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);printf("%d %c %s", s2.myNum, s2.myLetter, s2.myString); Try it Yourself » Imagine you have to write a program to store different information about Cars, such as brand, model, and year. What's great about structures is that you can create a single "Car template" and use it for every cars you make. See below for a real life example. Real-Life Example Use a structure to store different information about Cars: struct Car { char brand[50]; char model[50]; int year;};int main() { struct Car car1 = {"BMW", "X5", 1999}; struct Car car2 = {"Ford", "Mustang", 1969}; struct Car car3 = {"Toyota", "Corolla", 2011}; printf("%s %s %d", car1.brand, car1.model, car1.year); printf("%s %s %d", car2.brand, car2.model, car2.year); printf("%s %s %d", car3.brand, car3.model, car3.year); return 0;} Try it Yourself » Try it on GfG Practice In C, a structure is a user-defined data type that can be used to group items of possibly different types into a single type. The struct keyword is used to define a structure. The items in the structure are called its member and they can be of any valid data type.Example: C #include // Defining a structure struct A { int x; }; int main() { // Creating a structure variable struct A a; // Initializing member a.x = 11; printf("%d", a.x); return 0; } Explanation: In this example, a structure A is defined to hold an integer member x. A variable a of type struct A is created and its member x is initialized to 11 by accessing it using dot operator. The value of a.x is then printed to the console.Structures are used when you want to store a collection of different data types, such as integers, floats, or even other structures under a single name. To understand how structures are foundational to building complex data structures, the C Programming Course Online with Data Structures provides practical applications and detailed explanations.Syntax of StructureThere are two steps of creating a structure in C:Structure DefinitionCreating Structure VariablesStructure DefinitionA structure is defined using the struct keyword followed by the structure name and its members. It is also called a structure template or structure prototype, and no memory is allocated to the structure in the declaration.struct structure_name { data_type1 member1; data_type2 member2; ...};structure_name: Name of the structure.member1, member2, ...: Name of the members.data_type1, data_type2, ...: Type of the members.Be careful not to forget the semicolon at the end.Creating Structure VariableAfter structure definition, we have to create variable of that structure to use it. It is similar to the any other type of variable declaration:struct strcuture_name var;We can also declare structure variables with structure definition.struct structure_name { ...}var1, var2....;Basic Operations of StructureFollowing are the basic operations commonly used on structures:1. Access Structure MembersTo access or modify members of a structure, we use the (.) dot operator. This is applicable when we are using structure variables directly.structure_name . member1;strcuture_name . member2;In the case where we have a pointer to the structure, we can also use the arrow operator to access the members.structure_ptr -> member1structure_ptr -> member22. Initialize Structure MembersStructure members cannot be initialized with the declaration. For example, the following C program fails in the compilation.struct structure_name { data_type1 member1 = value1; // COMPILER ERROR: cannot initialize members here data_type2 member2 = value2; // COMPILER ERROR: cannot initialize members here ...};The reason for the above error is simple. When a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created. So there is no space to store the value assigned.We can initialize structure members in 4 ways which are as follows:Default InitializationBy default, structure members are not automatically initialized to 0 or NULL. Uninitialized structure members will contain garbage values. However, when a structure variable is declared with an initializer, all members not explicitly initialized are zero-initialized.struct structure_name = {0}; // Both x and y are initialized to 0Initialization using Assignment Operatorstruct structure_name str;str.member1 = value1;....Note: We cannot initialize the arrays or strings using assignment operator after variable declaration.Initialization using Initializer Liststruct structure_name str = {value1, value2, value3 ....};In this type of initialization, the values are assigned in sequential order as they are declared in the structure template.Initialization using Designated Initializer ListDesignated Initialization allows structure members to be initialized in any order. This feature has been added in the C99 standard.struct structure_name str = { .member1 = value1, .member2 = value2, .member3 = value3 };The Designated Initialization is only supported in C but not in C++. C #include // Defining a structure to represent a student struct Student { char name[50]; int age; float grade; }; int main() { // Declaring and initializing a structure // variable struct Student s1 = {"Rahul",20, 18.5}; // Designated Initializing another structure struct Student s2 = {.age = 18, .name = "Vikas", .grade = 22}; // Accessing structure members printf("%s\t%d\t%.2f", s1.name, s1.age, s1.grade); printf("%s\t%d\t%.2f", s2.name, s2.age, s2.grade); return 0; } OutputRahul 20 18.50 Vikas 18 22.00 3. Copy StructureCopying structure is simple as copying any other variables. For example, s1 is copied into s2 using assignment operator.s2 = s1;But this method only creates a shallow copy of s1 i.e. if the structure s1 have some dynamic resources allocated by malloc, and it contains pointer to that resource, then only the pointer will be copied to s2. If the dynamic resource is also needed, then it has to be copied manually (deep copy). C #include #include struct Student { int id; char grade; }; int main() { struct Student s1 = {1, 'A'}; // Create a copy of student s1 struct Student s1c = s1; printf("Student 1 ID: %d", s1c.id); printf("Student 1 Grade: %c", s1c.grade); return 0; } OutputStudent 1 ID: 1 Student 1 Grade: A4. Passing Structure to FunctionsStructure can be passed to a function in the same way as normal variables. Though, it is recommended to pass it as a pointer to avoid copying a large amount of data. C #include // Structure definition struct A { int x; }; // Function to increment values void increment(struct A a, struct A* b) { a.x++; b->x++; } int main() { struct A a = { 10 }; struct A b = { 10 }; // Passing a by value and b by pointer increment(a, &b); printf("a.x: %d \tb.x: %d", a.x, b.x); return 0; } 5. typedef for StructuresThe typedef keyword is used to define an alias for the already existing datatype. In structures, we have to use the struct keyword along with the structure name to define the variables. Sometimes, this increases the length and complexity of the code. We can use the typedef to define some new shorter name for the structure. C #include // Defining structure typedef struct { int a; } str1; // Another way of using typedef with structures typedef struct { int x; } str2; int main() { // Creating structure variables using new names str1 var1 = { 20 }; str2 var2 = { 314 }; printf("var1.a = %d", var1.a); printf("var2.x = %d", var2.x); return 0; } Outputvar1.a = 20 var2.x = 314Explanation: In this code, str1 and str2 are defined as aliases for the unnamed structures, allowing the creation of structure variables (var1 and var2) using these new names. This simplifies the syntax when declaring variables of the structure.Size of StructuresTechnically, the size of the structure in C should be the sum of the sizes of its members. But it may not be true for most cases. The reason for this is Structure Padding.Structure padding is the concept of adding multiple empty bytes in the structure to naturally align the data members in the memory. It is done to minimize the CPU read cycles to retrieve different data members in the structure.There are some situations where we need to pack the structure tightly by removing the empty bytes. In such cases, we use Structure Packing. C language provides two ways for structure packing:Using #pragma pack(1)Using __attribute((packed))__To know more about structure padding and packing, refer to this article - Structure Member Alignment, Padding and Data Packing.Nested StructuresIn C, a nested structure refers to a structure that contains another structure as one of its members. This allows you to create more complex data types by grouping multiple structures together, which is useful when dealing with related data that needs to be grouped within a larger structure.There are two ways in which we can nest one structure into another:Embedded Structure Nesting: The structure being nested is also declared inside the parent structure.Separate Structure Nesting: Two structures are declared separately and then the member structure is nested inside the parent structure.Accessing Nested MembersWe can access nested Members by using the same (.) dot operator two times as shown:str_parent . str_child . member;Example C #include // Child structure declaration struct child { int x; char c; }; // Parent structure declaration struct parent { int a; struct child b; }; int main() { struct parent p = { 25, 195, 'A' }; // Accessing and printing nested members printf("p.a = %d", p.a); printf("p.b.x = %d", p.b.x); printf("p.b.c = %c", p.b.c); return 0; } Outputp.a = 25 p.b.x = 195 p.b.c = AExplanation: In this code, the structure parent contains another structure child as a member. The parent structure is then initialized with values, including the values for the child structure's members.Structure PointerA pointer to a structure allows us to access structure members using the (->) arrow operator instead of the dot operator. C #include // Structure declaration struct Point { int x, y; }; int main() { struct Point p = { 1, 2 }; // ptr is a pointer to structure p struct Point* ptr = &p; // Accessing structure members using structure pointer printf("%d %d", ptr->x, ptr->y); return 0; } Explanation: In this example, ptr is a pointer to the structure Point. It holds the address of the structure variable p. The structure members x and y are accessed using the -> operator, which is used to dereference the pointer and access the members of the structure.Self-Referential StructuresThe self-referential structures are those structures that contain references to the same type as themselves i.e. they contain a member of the type pointer pointing to the same structure type.Example:struct str { int mem1; // Reference to the same type struct str* next;};Such kinds of structures are used in different data structures such as to define the nodes of linked lists, trees, etc.Bit FieldsBit Fields are used to specify the length of the structure members in bits. When we know the maximum length of the member, we can use bit fields to specify the size and reduce memory consumption.Syntaxstruct structure_name { data_type member_name: width_of_bit-field;};Uses of Structure in CC structures are used for the following:The structure can be used to define the custom data types that can be used to create some complex data types such as dates, time, complex numbers, etc. which are not present in the language.It can also be used in data organization where a large amount of data can be stored in different fields.Structures are used to create data structures such as trees, linked lists, etc.They can also be used for returning multiple values from a function. Hello there, future coding wizards! Today, we're going to embark on an exciting journey into the world of structures and functions in C. As your friendly neighborhood computer science teacher, I'm here to guide you through this adventure with plenty of examples and explanations. So, grab your virtual backpacks, and let's dive in! What Are Structures? Before we start passing structures around like hot potatoes, let's understand what they are. In C, a structure is like a container that can hold different types of data. Imagine you're packing for a trip - you might have a suitcase (the structure) that contains your clothes, toiletries, and maybe a good book (the different data types). Here's how we define a simple structure: struct Student { char name[50]; int age; float gpa; }; This Student structure can hold a name (as a string), an age (as an integer), and a GPA (as a float). Pretty neat, right? How to Pass Struct Elements Now, let's look at how we can pass individual elements of a structure to a function. It's like asking your friend to bring just your toothbrush from your suitcase, not the whole thing. #include struct Student { char name[50]; int age; float gpa; }; void printAge(int age) { printf("The student's age is: %d", age); } int main() { struct Student john = {"John Doe", 20, 3.8}; printAge(john.age); return 0; } In this example, we're only passing the age element of our Student structure to the printAge function. It's simple and straightforward! How to Pass a Struct Variable But what if we want to pass the whole "suitcase"? We can do that too! Here's how we pass an entire structure to a function: #include struct Student { char name[50]; int age; float gpa; }; void printStudent(struct Student s) { printf("Name: %s", s.name); printf("Age: %d", s.age); printf("GPA: %.2f", s.gpa); } int main() { struct Student john = {"John Doe", 20, 3.8}; printStudent(john); return 0; } Here, we're passing the entire john structure to the printStudent function. It's like handing over your whole suitcase to your friend. How to Return Struct from a Function Now, let's get fancy. What if we want a function to create and return a whole structure? It's like asking your friend to pack a suitcase for you and bring it back. Here's how we do it: #include #include struct Student { char name[50]; int age; float gpa; }; struct Student createStudent(char *name, int age, float gpa) { struct Student s; strcpy(s.name, name); s.age = age; s.gpa = gpa; return s; } int main() { struct Student newStudent = createStudent("Jane Doe", 22, 3.9); printf("New student created: %s, %d years old, GPA: %.2f", newStudent.name, newStudent.age, newStudent.gpa); return 0; } In this example, our createStudent function is like a student-creating machine. You give it the details, and it hands you back a fully packed "student suitcase"! How to Pass a Struct by Reference Sometimes, we want to modify the original structure inside a function. It's like asking your friend to add something to your suitcase without bringing the whole thing back. For this, we use pointers: #include struct Student { char name[50]; int age; float gpa; }; void updateAge(struct Student *s, int newAge) { s->age = newAge; } int main() { struct Student john = {"John Doe", 20, 3.8}; printf("John's age before: %d", john.age); updateAge(&john, 21); printf("John's age after: %d", john.age); return 0; } Here, we're passing the address of our john structure to the updateAge function. The function then uses the -> operator to access and modify the age field directly. How to Return a Struct Pointer Lastly, let's look at how we can return a pointer to a structure. This is useful when we're dealing with large structures or when we want to create structures that persist after the function ends. #include #include #include struct Student { char name[50]; int age; float gpa; }; struct Student* createStudentPointer(char *name, int age, float gpa) { struct Student *s = malloc(sizeof(struct Student)); strcpy(s->name, name); s->age = age; s->gpa = gpa; return s; } int main() { struct Student *newStudent = createStudentPointer("Bob Smith", 19, 3.7); printf("New student created: %s, %d years old, GPA: %.2f", newStudent->name, newStudent->age, newStudent->gpa); free(newStudent); // Don't forget to free the allocated memory! return 0; } In this example, our createStudentPointer function is like a valet service. It not only packs the suitcase for you but also remembers where it put it and gives you the location (the pointer). Conclusion And there you have it, folks! We've packed, unpacked, modified, and created structures in various ways. Remember, practice makes perfect, so don't be afraid to experiment with these concepts. Who knows? You might just structure your way to becoming the next big thing in programming! Here's a quick reference table of the methods we've covered: Method Description Passing Struct Elements Pass individual fields of a structure to a function Passing Struct Variable Pass an entire structure to a function Returning Struct Create and return a structure from a function Passing Struct by Reference Modify a structure inside a function using pointers Returning Struct Pointer Create a structure on the heap and return its pointer Happy coding, and may your structures always be well-organized! Credits: Image by storyset Diagnostics library Memory management library Metaprogramming library (C++11) General utilities library Containers library Iterators library Ranges library (C++20) Algorithms library Strings library Text processing library Numerics library Date and time library Calendar (C++20) − Time zone (C++20) Input/output library Concurrency support library (C++11) Execution support library (C++26) Feature test macros (C++20) Page 2 From cppreference.com < c ISO/IEC 9899/AMD1:1995, a.k.a. C95, is a previous revision of the C standard. [edit] New language features Digraphs __STDC_VERSION__ [edit] New library features [edit] From cppreference.com A struct is a type consisting of a sequence of members whose storage is allocated in an ordered sequence (as opposed to union, which is a type consisting of a sequence of members whose storage overlaps). The type specifier for a struct is identical to the union type specifier except for the keyword used: [edit] Syntax struct attr-spec-seq(optional) name(optional) { struct-declaration-list } (1) struct attr-spec-seq(optional) name (2) 1) Struct definition: introduces the new type struct name and defines its meaning 2) If used on a line of its own, as in struct name ;, declares but doesn't define the struct name (see forward declaration below). In other contexts, names the previously-declared struct, and attr-spec-seq is not allowed. name - the name of the struct that's being defined struct-declaration-list - any number of variable declarations, bit-field declarations, and static assert declarations. Members of incomplete type and members of function type are not allowed (except for the flexible array member described below) attr-spec-seq - (C23)optional list of attributes, applied to the struct type Within a struct object, addresses of its elements (and the addresses of the bit-field allocation units) increase in order in which the members were defined. A pointer to a struct can be cast to a pointer to its first member (or, if the member is a bit-field, to its allocation unit). Likewise, a pointer to the first member of a struct can be cast to a pointer to the enclosing struct. There may be unnamed padding between any two members of a struct or after the last member, but not before the first member. The size of a struct is at least as large as the sum of the sizes of its members. If a struct defines at least one named member, it is allowed to additionally declare its last member with incomplete array type. When an element of the flexible array member is accessed (in an expression that uses operator . or -> with the flexible array member's name as the right-hand-side operand), then the struct behaves as if the array member had the longest size fitting in the memory allocated for this object. If no additional storage was allocated, it behaves as if an array with 1 element, except that the behavior is undefined if that element is accessed or a pointer one past that element is produced. Initialization and the assignment operator ignore the flexible array member. sizeof omits it, but may have more trailing padding than the omission would imply. Structures with flexible array members (or unions who have a recursive-possibly structure member with flexible array member) cannot appear as array elements or as members of other structures. struct s { int n; double d[]; }; // s.d is a flexible array member struct s t1 = { 0 }; // OK, d is as if double d[1], but UB to access struct s t2 = { 1, { 4.2 } }; // error: initialization ignores flexible array // if sizeof (double) == 8 struct s *s1 = malloc(sizeof (struct s) + 64); // as if d was double d[8] struct s *s2 = malloc(sizeof (struct s) + 40); // as if d was double d[5] s1 = malloc(sizeof (struct s) + 10); // now as if d was double d[1]. Two bytes excess. double *dp = &(s1->d[0]); // OK *dp = 42; // OK s1->d[1]++; // Undefined behavior. 2 excess bytes can't be accessed // as double. s2 = malloc(sizeof (struct s) + 6); // same, but UB to access because 2 bytes are // missing to complete 1 double dp = &(s2->d[0]); // OK, can take address just fine *dp = 42; // undefined behavior *s1 = *s2; // only copies s.n, not any element of s.d // except those caught in sizeof (struct s) (since C99) Similar to union, an unnamed member of a struct whose type is a struct without name is known as anonymous struct. Every member of an anonymous struct is considered to be a member of the enclosing struct or union, keeping their structure layout. This applies recursively if the enclosing struct or union is also anonymous. struct v { union // anonymous union { struct { int i, j; }; // anonymous structure struct { long k, l; } w; }; int m; } v1; v1.i = 2; // valid v1.k = 3; // invalid: inner structure is not anonymous v1.w.k = 5; // valid Similar to union, the behavior of the program is undefined if struct is defined without any named members (including those obtained via anonymous nested structs or unions). (since C11) [edit] Forward declaration A declaration of the following form struct attr-spec-seq(optional) name ; hides any previously declared meaning for the name name in the tag name space and declares name as a new struct name in current scope, which will be defined later. Until the definition appears, this struct name has incomplete type. This allows structs that refer to each other: struct y; struct x { struct y *p; /* ... */ }; struct y { struct x *q; /* ... */ }; Note that a new struct name may also be introduced just by using a struct tag within another declaration, but if a previously declared struct with the same name exists in the tag name space, the tag would refer to that name struct s* p = NULL; // tag naming an unknown struct declares it struct s { int a; }; // definition for the struct pointed to by p void g(void) { struct s; // forward declaration of a new, local struct s // this hides global struct s until the end of this block struct s *p; // pointer to local struct s // without the forward declaration above, // this would point at the file-scope s struct s { char* p; }; // definitions of the local struct s } [edit] Keywords struct [edit] Notes See struct initialization for the rules regarding the initializers for structs. Because members of incomplete type are not allowed, and a struct type is not complete until the end of the definition, a struct cannot have a member of its own type. A pointer to its own type is allowed, and is commonly used to implement nodes in linked lists or trees. Because a struct declaration does not establish scope, nested types, enumerations and enumerators introduced by declarations within struct-declaration-list are visible in the surrounding scope where the struct is defined. [edit] Example #include #include int main(void) { // Declare the struct type. struct car { char* make; int year; }; // Declare and initialize an object of a previously-declared struct type. struct car c = {.year = 1923, .make = "Nash"}; printf("1) Car: %d %s", c.year, c.make); // Declare a struct type, an object of that type, and a pointer to it. struct spaceship { char* model; int max_speed; } ship = {"T-65 X-wing starfighter", 1050}, *pship = &ship; printf("2) Spaceship: %s. Max speed: %d km/h", ship.model, ship.max_speed); // Address increase in order of definition. Padding may be inserted. struct A { char a; double b; char c; }; printf("3) Offset of char a = %zu" "4) Offset of double b = %zu" "5) Offset of char c = %zu" "6) Size of struct A = %zu", offsetof(struct A, a), offsetof(struct A, b), offsetof(struct A, c), sizeof(struct A)); struct B { char a; char b; double c; }; printf("7) Offset of char a = %zu" "8) Offset of char b = %zu" "9) Offset of double c = %zu" "A) Size of struct B = %zu", offsetof(struct B, a), offsetof(struct B, b), offsetof(struct B, c), sizeof(struct B)); // A pointer to a struct can be cast to a pointer // to its first member and vice versa. char** pmodel = (char **)pship; printf("B) %s", *pmodel); pship = (struct spaceship *)pmodel; } Possible output: 1) Car: 1923 Nash 2) Spaceship: T-65 X-wing starfighter. Max speed: 1050 km/h 3) Offset of char a = 0 4) Offset of double b = 8 5) Offset of char c = 16 6) Size of struct A = 24 7) Offset of char a = 0 8) Offset of char b = 1 9) Offset of double c = 8 A) Size of struct B = 16 B) T-65 X-wing starfighter [edit] Defect reports The following behavior-changing defect reports were applied retroactively to previously published C standards. DR Applied to Behavior as published Correct behavior DR 499 C11 members of anonymous structs/unions were considered members of the enclosing struct/union they retain their memory layout [edit] References C23 standard (ISO/IEC 9899:2024): 6.7.2.1 Structure and union specifiers (p: TBD) C17 standard (ISO/IEC 9899:2018): 6.7.2.1 Structure and union specifiers (p: 81-84) C11 standard (ISO/IEC 9899:2011): 6.7.2.1 Structure and union specifiers (p: 112-117) C99 standard (ISO/IEC 9899:1999): 6.7.2.1 Structure and union specifiers (p: 101-104) C89/C90 standard (ISO/IEC 9899:1990): 3.5.2.1 Structure and union specifiers [edit] See also struct and union member access bit-field struct initialization C++ documentation for Class declaration MASTERCLASSALUMNI Hello, aspiring programmers! Today, we're going to dive into the fascinating world of structures in C. Don't worry if you've never written a line of code before – I'll guide you through this journey step by step, just as I've done for countless students over my years of teaching. So, let's embark on this adventure together! What are Structures in C? Imagine you're building a digital address book. For each person, you'd want to store their name, phone number, and email address. In C, a structure allows you to group these related pieces of information together under a single name. It's like creating a custom data type that can hold multiple pieces of data of different types. Here's a simple analogy: think of a structure as a backpack. Just as a backpack can hold different items like books, pens, and a water bottle, a structure can hold different types of data. Declare (Create) a Structure Let's start by creating our first structure. We'll use the keyword struct to do this. struct Person { char name[50]; int age; float height; }; In this example, we've created a structure called Person. It can hold a name (up to 50 characters), an age (an integer), and a height (a floating-point number). Structure Variable Declaration Now that we've defined our structure, let's create a variable of this type. struct Person john; This line creates a variable named john of type struct Person. It's like saying, "Hey C, give me a backpack called 'john' that can hold a name, age, and height." Structure Initialization Let's put some data into our structure: struct Person john = {"John Doe", 30, 5.9}; Here, we're filling our john backpack with a name ("John Doe"), an age (30), and a height (5.9). Accessing the Structure Members To access or modify the data in a structure, we use the dot (.) operator: printf("Name: %s", john.name); printf("Age: %d", john.age); printf("Height: %.1f", john.height); john.age = 31; // Happy birthday, John! This code prints John's details and then updates his age. It's like reaching into specific pockets of our backpack. Copying Structures In C, you can copy one structure to another of the same type: struct Person jane = john; This creates a new Person called jane with the same data as john. It's like making an exact copy of John's backpack for Jane. Structures as Function Arguments You can pass structures to functions, just like any other data type: void printPerson(struct Person p) { printf("Name: %s, Age: %d, Height: %.1f", p.name, p.age, p.height); } // Usage printPerson(john); This function takes a Person structure and prints its contents. It's like handing your backpack to someone so they can tell you what's inside. Pointers to Structures Sometimes, it's more efficient to use pointers to structures, especially when dealing with large structures: struct Person *pJohn = &john; printf("Name: %s", (*pJohn).name); // Or, more commonly: printf("Age: %d", pJohn->age); The arrow operator (->) is a shorthand for dereferencing a pointer and accessing a member. It's like having a map that points to where your backpack is, rather than carrying the backpack itself. Bit Fields Bit fields allow you to specify the number of bits to be used for structure members: struct PackedData { unsigned int flag : 1; unsigned int data : 31; }; This structure uses only 32 bits of memory: 1 bit for flag and 31 bits for data. It's like having a tiny, specialized compartment in your backpack for very small items. Methods Table Here's a quick reference table of the methods we've covered: Method Description Example Structure Declaration Define a new structure struct Person { ... }; Variable Declaration Create a variable of a structure type struct Person john; Initialization Set initial values for a structure struct Person john = {"John", 30, 5.9}; Member Access Access or modify structure members john.age = 31; Structure Copying Copy one structure to another struct Person jane = john; Function Arguments Pass structures to functions void printPerson(struct Person p) { ... } Pointers to Structures Use pointers for efficient handling struct Person *pJohn = &john; Bit Fields Specify bit sizes for members struct PackedData { unsigned int flag : 1; }; And there you have it! We've unpacked the basics of structures in C. Remember, like learning to organize your backpack efficiently, mastering structures takes practice. Don't be discouraged if it doesn't click immediately – keep coding, keep experimenting, and soon you'll be structuring your data like a pro! Happy coding, future C wizards! Credits: Image by storyset C allows programmers to create user-defined data types by grouping data of different types together using struct keywords, such data types are called structures. Like any other data type in C, variables of user-defined structure occupy addresses in a memory block, and pointers can be used to point them. A pointer pointing to a structure is called structure pointer. Structures and pointers in C together help in accessing structure members efficiently.Structure pointer declaration is similar to declaring a structure variable using the struct keyword followed by the type of structure it will point to. A structure pointer can only hold the address of the structure used at the time of its declaration. Structures and pointers in C together make accessing structure value and passing to functions easier. C allows programmers to create their data type by grouping different types together into one using structures. For example, if we want to store information about our classmates, each student variable should hold information about the student's name, roll number, and grades. No pre-defined data type in C can alone store all this information.For such cases where we want to store information that no data type can hold, we create our data types using structure to hold the required information.Different components of a structure are called members for example, in the above case, student name and roll number are members of the structure. Like every other data type, structure variables are stored in memory, and we can use pointers to store their addresses.Structure pointer points to the address of the structure variable in the memory block to which it points. This pointer can be used to access and change the value of structure members. This way, structures and pointers in C can be used to create and access user-defined data types conveniently.Before understanding how to use structures and pointers in C together, let us understand how structures are defined and accessed using the variable name. C struct keyword is used to create a new data type, followed by the structure name. We define different members of the structure inside parenthesis. Once a structure is defined, its name structure_name can be used to declare variables asTo access the value of members of a structure, the dot (.) operator and the structure variable name followed by the member's name are used. For example, if we want the value of member_variable_1 from a structure variable structure_variable syntax will bestructure_variable is the structure variable, and member_variable_1 is one of its members. Note:The structure members do not occupy space in memory until they are associated with a structure variable. Now that we know how structures are declared and accessed let us create a structure User that holds information about user name, his role and his age. Here name, role, age are members of the structure User.OutputExplanationHere, we have created a user-defined data type, User using the struct keyword, this structure has three members that are name (string), age (int), and role (string). To store information of two users, we have declared two structure variables user_1 and user_2 of type User and later initialized and accessed their value in the main() function using the variable name and dot (.) operator. As shown in the above figure, a structure pointer stores the memory address of a structure variable. This is the reason why in the figure ptr stores the location 3000 inside it, which is the address of the variable student1.We now know how structures are defined and used in a C code let us see how we can use structures with pointers to access structure variables and their members. Declaration of structure pointer is similar to the declaration of structure variables, and the only difference is that the pointer name is prefixed with an asterisk * symbol.SyntaxStructure pointer in C is declared using the keyword struct followed by structure name to which the pointer will point to followed by pointer name. A structure pointer can only hold the address of a variable of the same structure type used in its declaration.This way structures and pointers in C are used together to create a pointer pointing to the structure. After a structure pointer is declared, we need to initialize it to a variable before using it. To initialize a variable, we need to provide the address of the structure variable using the & operator.Also, the structure pointer can be initialized during the time of declaration. There are two ways to access the values of structure members using pointers -1. Using asterisk (*) and dot (.) operator with the structure pointer. 2. Using membership or arrow (->) operator.ExamplesLet us see some examples to understand how we can access structure members using two different approaches.Example 1: Accessing structure members using the dot operatorOutputExplanationHere, cp is a pointer that points to the structure variable first_point. This means dereferencing the pointer gives us the content of first_point. Hence, *cp and first_point are functionally identical. To access members of the structure dot operator can be used followed by the member name.For example, in the example above: (*cp).x refers to member x of first_point. (*cp).y refers to member y of first_point. Note: Parentheses around the pointer is important because the precedence of dot operator is greater than indirection (*) operator. Example 2: Accessing structure members using the arrow operatorAnother way to access structure members in C is using the (->) operator. Using this way, we don't need an asterisk and dot operator with the pointer. To access members of the structure using (->) operator we write pointer name with -> followed by the name of the member that isLet us see an example to understand how we can use an arrow operator to access structure members using structures and pointers in C.OutputAccessing members of the structure using the membership operator on structure pointer makes code more readable when compared to the other approach.Example 3: Structure pointer in function argumentsOutputHere, we have defined function arguments as structure pointers and when we are creating function calls instead of passing structure variables, we are passing reference of them to function. Because reference of variable is passed to the function any changes made on structure members inside function body will persist outside the function scope. Structures in C allow programmers to create user-defined data types by grouping different defined data types into one. Different individual components in the structure are called members. To create a new structure struct keyword is used and similarly, when a structure variable is created struct keyword is used followed by structure type and variable name. Pointer pointing to a structure variable is called a structure pointer, and structures and pointers in C together can be used to access and change the values of members of the structure they are pointing. Declaring a structure pointer is similar to the declaration of a structure variable. To declare a structure pointer struct keyword is used followed by the structure name and pointer name with an asterisk * symbol. Members of a structure can be accessed from pointers using two ways that are. Using dot and asterisk operator on a pointer. Using arrow operator (->) on a pointer.