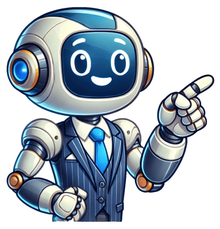
I'm not a bot



Finding the length of an array is a very common and basic task in Java programming. Knowing the size of an array is essential so that we can perform certain operations. In this article, we will discuss multiple ways to find the length or size of an array in Java.In this article, we will learn:How to find the length of an array using the length propertyDifference between length, length() and size() methodsVarious approaches to find the array length, including loops and Stream APIDifferent Ways to Find the Length or Size of an Array1. Using the length Property (Best and Easiest Way)The length property is the most common way to find the size of an array in Java. It gives the exact number of elements present in the array.Example: This example demonstrates how to find the length (size) of an array using the length property. Java // Finding the length of an Arraypublic class Geeks { public static void main(String[] args) { // Here arr is the // array name of int type int[] arr = new int[4]; System.out.println("The size of the Array is " + arr.length); }}OutputThe size of the Array is 4Time complexity:O(1) constant timeNote:length is a property not a method.length property works for all array types (int[] , String[], double[]) etc.2. Finding Array Length Using a For Loop (Manual Counting)When iterating over an array, we can also determine its size manually.Example: This example, demonstrates how to use for each loop to calculate the length of an array. Java // Java program to demonstrate for loop// to calculate length of all type of Arraysimport java.util.*; public class Geeks { public static void main(String[] arg) { int[] arr = { 1, 2, 3, 4, 5, 6, 7 }; int c = 0; for (int i : arr) c++; System.out.println("The Size of the array is " + c); }}OutputThe Size of the array is 7Time Complexity: O(n) slower than lengthNote: This approach is not recommended in real world code.3. Finding Array Length Using Java 8 Stream APIStream API was introduced in Java 8 and because of it we can perform various operations on arrays using functional programming. The Stream class provide count() method which is used to count the number of elements in an array.Example: This example, demonstrates using Arrays.stream() with count() to calculate the length of an array. Java // Java program to demonstrate Stream.count()// method to calculate length of an arrayimport java.util.*; public class Geeks { public static void main(String[] argv) { // Creating Array and Populating them int[] arr = { 1, 2, 3, 4, 5 }; // calculating the length of the arrays long e = Arrays.stream(arr).count(); // print the length of the arrays System.out.println("The size of the array is " + e); }}OutputThe size of the array is 5Time Complexity: O(n)Note: Avoid Arrays.stream(arr).count() for arrays because it forces unnecessary iteration.length vs length() vs size()The difference between length, length() and size() is listed below:FeatureUsed ForExamplelengthIt is used for arrays to calculate the number of elementsarr.lengthlength()It is used for strings to calculate the number of characters in the string s.length()size()It is used for collections like ArrayList or List to calculate the number of elementslist.size() I would advise to never use sizeof (even if it can be used) to get any of the two different sizes of an array, either in number of elements or in bytes, which are the last two cases I show here. For each of the two sizes, the macros shown below can be used to make it safer. The reason is to make obvious the intention of the code to maintainers, and difference sizeof(ptr) from sizeof(arr) at first glance (which written this way isn't obvious), so that bugs are then obvious for everyone reading the code. TL;DR:#define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0]) + must_be_array(arr))#define ARRAY_BYTES(arr) (sizeof(arr) + must_be_array(arr))#define is_same_type(a, b) __builtin_types_compatible_p(a, b)#define is_same_typeof(a, b) is_same_type(typeof(a), typeof(b))#define decay(a) (&*(a))#define is_array(arr) (!is_same_typeof(arr, decay(arr)))#define must_be(e) \(\ 0 * (int)sizeof(\ struct { \ static_assert(e); \ char ISO_C_forbids_a_struct_with_no_members__; \ } \ ) \)#define must_be_array(arr) must_be(is_array(arr)) There have been important bugs regarding this topic: disagree with the solution that Linus provides, which is to never use array notation for parameters of functions.I like array notation as documentation that a pointer is being used as an array. But that means that a fool-proof solution needs to be applied so that it is impossible to write buggy code.From an array we have three sizes which we might want to know:The size of the elements of the arrayThe number of elements in the arrayThe size in bytes that the array uses in memory The size of the elements of the arrayThe first one is very simple, and it doesn't matter if we are dealing with an array or a pointer, because it's done the same way.Example of usage:void foo(size_t nmemb, int arr[nmemb]){ qsort(arr, nmemb, sizeof(arr[0]), cmp);}qsort() needs this value as its third argument. For the other two sizes, which are the topic of the question, we want to make sure that we're dealing with an array, and break the compilation if not, because if we're dealing with a pointer, we will get wrong values. When the compilation is broken, we will be able to easily see that we weren't dealing with an array, but with a pointer instead, and we will just have to write the code with a variable or a macro that stores the size of the array behind the pointer. The number of elements in the arrayThis one is the most common, and many answers have provided you with the typical macro COUNTOF:#define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0]))Recent versions of compilers, such as GCC 8, will warn you when you apply this macro to a pointer, so it is safe (there are other methods to make it safe with older compilers).It works by dividing the size in bytes of the whole array by the size of each element.Examples of usage:void foo(size_t nmemb){ char buf[nmemb]; fgets(buf, COUNTOF(buf), stdin);}void bar(size_t nmemb){ int arr[nmemb]; for (size_t i = 0; i < COUNTOF(arr); i++) arr[i] = i;}If these functions didn't use arrays, but got them as parameters instead, the former code would not compile, so it would be impossible to have a bug (given that a recent compiler version is used, or that some other trick is used), and we need to replace the macro call by the value:void foo(size_t nmemb, char buf[nmemb]){ fgets(buf, nmemb, stdin);}void bar(size_t nmemb, int arr[nmemb]){ for (size_t i = nmemb - 1; i < nmemb; i--) arr[i] = i;} The size in bytes that the array uses in memoryCOUNTOF is commonly used as a solution to the previous case, but this case is rarely written safely, maybe because it's less common.The common way to get this value is to use sizeof(arr). The problem: the same as with the previous one; if you have a pointer instead of an array, your program will go nuts.The solution to the problem involves using the same macro as before, which we know to be safe (it breaks compilation if it is applied to a pointer):#define ARRAY_BYTES(arr) (sizeof((arr)[0]) * COUNTOF(arr))How it works is very simple: it undoes the division that COUNTOF does, so after mathematical cancellations you end up with just one sizeof(arr), but with the added safety of the COUNTOF construction.Example of usage:void foo(size_t nmemb){ int arr[nmemb]; memset(arr, 0, ARRAY_BYTES(arr));}memset() needs this value as its third argument.As before, if the array is received as a parameter (a pointer), it won't compile, and we will have to replace the macro call by the value:void foo(size_t nmemb, int arr[nmemb]){ memset(arr, 0, sizeof(arr[0]) * nmemb);} Update (23/apr/2020): -Wsizeof-pointer-div doesn't warn from library code:Today I found out that the new warning in GCC only works if the macro is defined in a header that is not a system header. If you define the macro in a header that is installed in your system (usually /usr/local/include/ or /usr/include/) (#include), the compiler will NOT emit a warning (I tried GCC 9.3.0).So we have #define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0])) and want to make it safe. We will need C2X static_assert() and some GCC extensions: Statements and Declarations in Expressions, __builtin_types_compatible_p:#include #define is_same_type(a, b) __builtin_types_compatible_p(a, b)#define is_same_typeof(a, b) is_same_type(typeof(a), typeof(b))#define decay(a) (&*(a))#define is_array(arr) (!is_same_typeof(arr, decay(arr)))#define Static_assert_array(arr) static_assert(is_array(arr))#define COUNTOF(arr) \({ \ Static_assert_array(arr); \ sizeof(arr) / sizeof((arr)[0]); \})Now COUNTOF() is completely safe, and therefore all its derivatives will be safe. Update: Allow the macro to be used at file scope:Unfortunately, the ({}) gcc extension cannot be used at file scope.To be able to use the macro at file scope, the static assertion must beinside sizeof(struct {}). Then, multiply it by 0 to not affectthe result. A cast to (int) might be good to simulate a functionthat returns (int)0 (in this case it is not necessary, but then itis reusable for other things).Additionally, the definition of ARRAY_BYTES() can be simplified a bit.#include #include #include #include #define is_same_type(a, b) __builtin_types_compatible_p(a, b)#define is_same_typeof(a, b) is_same_type(typeof(a), typeof(b))#define decay(a) (&*(a))#define is_array(arr) (!is_same_typeof(arr, decay(arr)))#define must_be(e) \(\ 0 * (int)sizeof(\ struct { \ static_assert(e); \ char ISO_C_forbids_a_struct_with_no_members__; \ } \ ) \)#define must_be_array(arr) must_be(is_array(arr))#define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0]) + must_be_array(arr))#define ARRAY_BYTES(arr) (sizeof(arr) + must_be_array(arr)) This code makes use of the following extensions, which are completely necessary, and their presence is absolutely necessary to achieve safety. If your compiler doesn't have them, or some similar ones, then you can't achieve this level of safety.__builtin_types_compatible_p()typeof()I also make use of the following C2X feature. However, its absence by using an older standard can be overcome using some dirty tricks (see for example: What is :-!! in C code?) (in C11 you also have static_assert(), but it requires a message). I would advise to never use sizeof (even if it can be used) to get any of the two different sizes of an array, either in number of elements or in bytes, which are the last two cases I show here. For each of the two sizes, the macros shown below can be used to make it safer. The reason is to make obvious the intention of the code to maintainers, and difference sizeof(ptr) from sizeof(arr) at first glance (which written this way isn't obvious), so that bugs are then obvious for everyone reading the code. TL;DR:#define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0]) + must_be_array(arr))#define ARRAY_BYTES(arr) (sizeof(arr) + must_be_array(arr))#define is_same_type(a, b) __builtin_types_compatible_p(a, b)#define is_same_typeof(a, b) is_same_type(typeof(a), typeof(b))#define decay(a) (&*(a))#define is_array(arr) (!is_same_typeof(arr, decay(arr)))#define must_be(e) \(\ 0 * (int)sizeof(\ struct { \ static_assert(e); \ char ISO_C_forbids_a_struct_with_no_members__; \ } \ ) \)#define must_be_array(arr) must_be(is_array(arr)) There have been important bugs regarding this topic: disagree with the solution that Linus provides, which is to never use array notation for parameters of functions.I like array notation as documentation that a pointer is being used as an array. But that means that a fool-proof solution needs to be applied so that it is impossible to write buggy code.From an array we have three sizes which we might want to know:The size of the elements of the arrayThe number of elements in the arrayThe size in bytes that the array uses in memory The size of the elements of the arrayThe first one is very simple, and it doesn't matter if we are dealing with an array or a pointer, because it's done the same way.Example of usage:void foo(size_t nmemb, int arr[nmemb]){ qsort(arr, nmemb, sizeof(arr[0]), cmp);}qsort() needs this value as its third argument. For the other two sizes, which are the topic of the question, we want to make sure that we're dealing with an array, and break the compilation if not, because if we're dealing with a pointer, we will get wrong values. When the compilation is broken, we will be able to easily see that we weren't dealing with an array, but with a pointer instead, and we will just have to write the code with a variable or a macro that stores the size of the array behind the pointer. The number of elements in the arrayThis one is the most common, and many answers have provided you with the typical macro COUNTOF:#define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0]))Recent versions of compilers, such as GCC 8, will warn you when you apply this macro to a pointer, so it is safe (there are other methods to make it safe with older compilers).It works by dividing the size in bytes of the whole array by the size of each element.Examples of usage:void foo(size_t nmemb){ char buf[nmemb]; fgets(buf, COUNTOF(buf), stdin);}void bar(size_t nmemb){ int arr[nmemb]; for (size_t i = 0; i < COUNTOF(arr); i++) arr[i] = i;}If these functions didn't use arrays, but got them as parameters instead, the former code would not compile, so it would be impossible to have a bug (given that a recent compiler version is used, or that some other trick is used), and we need to replace the macro call by the value:void foo(size_t nmemb, char buf[nmemb]){ fgets(buf, nmemb, stdin);}void bar(size_t nmemb, int arr[nmemb]){ for (size_t i = nmemb - 1; i < nmemb; i--) arr[i] = i;} The size in bytes that the array uses in memoryCOUNTOF is commonly used as a solution to the previous case, but this case is rarely written safely, maybe because it's less common.The common way to get this value is to use sizeof(arr). The problem: the same as with the previous one; if you have a pointer instead of an array, your program will go nuts.The solution to the problem involves using the same macro as before, which we know to be safe (it breaks compilation if it is applied to a pointer):#define ARRAY_BYTES(arr) (sizeof((arr)[0]) * COUNTOF(arr))How it works is very simple: it undoes the division that COUNTOF does, so after mathematical cancellations you end up with just one sizeof(arr), but with the added safety of the COUNTOF construction.Example of usage:void foo(size_t nmemb){ int arr[nmemb]; memset(arr, 0, ARRAY_BYTES(arr));}memset() needs this value as its third argument.As before, if the array is received as a parameter (a pointer), it won't compile, and we will have to replace the macro call by the value:void foo(size_t nmemb, int arr[nmemb]){ memset(arr, 0, sizeof(arr[0]) * nmemb);} Update (23/apr/2020): -Wsizeof-pointer-div doesn't warn from library code:Today I found out that the new warning in GCC only works if the macro is defined in a header that is not a system header. If you define the macro in a header that is installed in your system (usually /usr/local/include/ or /usr/include/) (#include), the compiler will NOT emit a warning (I tried GCC 9.3.0).So we have #define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0])) and want to make it safe. We will need C2X static_assert() and some GCC extensions: Statements and Declarations in Expressions, __builtin_types_compatible_p:#include #define is_same_type(a, b) __builtin_types_compatible_p(a, b)#define is_same_typeof(a, b) is_same_type(typeof(a), typeof(b))#define decay(a) (&*(a))#define is_array(arr) (!is_same_typeof(arr, decay(arr)))#define Static_assert_array(arr) static_assert(is_array(arr))#define COUNTOF(arr) \({ \ Static_assert_array(arr); \ sizeof(arr) / sizeof((arr)[0]); \})Now COUNTOF() is completely safe, and therefore all its derivatives will be safe. Update: Allow the macro to be used at file scope:Unfortunately, the ({}) gcc extension cannot be used at file scope.To be able to use the macro at file scope, the static assertion must beinside sizeof(struct {}). Then, multiply it by 0 to not affectthe result. A cast to (int) might be good to simulate a functionthat returns (int)0 (in this case it is not necessary, but then itis reusable for other things).Additionally, the definition of ARRAY_BYTES() can be simplified a bit.#include #include #include #include #define is_same_type(a, b) __builtin_types_compatible_p(a, b)#define is_same_typeof(a, b) is_same_type(typeof(a), typeof(b))#define decay(a) (&*(a))#define is_array(arr) (!is_same_typeof(arr, decay(arr)))#define must_be(e) \(\ 0 * (int)sizeof(\ struct { \ static_assert(e); \ char ISO_C_forbids_a_struct_with_no_members__; \ } \ ) \)#define must_be_array(arr) must_be(is_array(arr))#define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0]) + must_be_array(arr))#define ARRAY_BYTES(arr) (sizeof(arr) + must_be_array(arr)) This code makes use of the following extensions, which are completely necessary, and their presence is absolutely necessary to achieve safety. If your compiler doesn't have them, or some similar ones, then you can't achieve this level of safety.__builtin_types_compatible_p()typeof()I also make use of the following C2X feature. However, its absence by using an older standard can be overcome using some dirty tricks (see for example: What is :-!! in C code?) (in C11 you also have static_assert(), but it requires a message). This post provides an overview of some of the available alternatives to find the size of an array in C.1. sizeof operatorThe standard way is to use the sizeof operator to find the size of a C-style array. The sizeof operator on an array returns the total memory occupied by the array in bytes. To determine the total number of the array elements, the trick is to divide the total memory occupied by the array by the size of each element. This is demonstrated below in C: int arr[] = { 1, 2, 3, 4, 5 };size_t n = sizeof(arr)/sizeof(arr[0]);printf("The size of the array is %d", n); DownloadRun CodeFor simplicity and to improve code readability, it is suggested to construct a MACRO out of it, as shown below: #define SIZEOF(a) sizeof(a)/sizeof(*a)int arr[] = { 1, 2, 3, 4, 5 };printf("The size of the array is %d", n); DownloadRun CodeWe know that an array decays into a pointer when passed to a function as an argument regardless of whether the parameter is declared as int[] or not. So, above approach works only with the static arrays but fails on the dynamically allocated arrays and function parameters as they both involve pointers. // array decays into a pointersize_t n = sizeof(arr)/sizeof(arr[0]);int *a = malloc(N * sizeof(int));printf("%zu", sizeof(a)/sizeof(a[0]));// this won't workint b[] = { 1, 2, 3, 4, 5 };printf("%zu", getSize(b));// this won't work DownloadRun Code2. Using pointer arithmeticThe trick is to use the expression (&arr)[1] - arr to get the array arr size. Both arr and &arr points to the same memory location, but they both have different types.arr has the type int* and decays into a pointer to the first element of the array. Hence, any knowledge about the size of the array is gone.&arr results in a pointer of type int (*)[n], i.e., a pointer to an array of n ints. So, &arr points to the entire array and *(&arr + 1) (or &arr)[1]) points to the next byte after the array.This works because of the way pointer arithmetic works in C. We know that a pointer to int is advanced by sizeof(int) when incrementing by 1. Similarly, a pointer to int[n] is advanced by sizeof(int[n]), which is the size of the entire array. int arr[] = { 1, 2, 3, 4, 5 };size_t n = (&arr)[1] - arr;printf("The size of the array is %d", n); DownloadRun CodeThats all about finding the size of an array in C using sizeof operator and pointer arithmetic.Read More:Find the length of an array in C++ The size of an array is generally considered to be the number of elements in the array or the size of the simplest method to find the size of an array in C.The simplest method to find the size of an array in C is by using sizeof operator. First determine the total size of the array in bytes and divide it by the size of one element to get the number of elements. C #include int main() { int arr[] = {10, 20, 30, 40, 50}; // Total size divided by size of one element int n = sizeof(arr) / sizeof(arr[0]); printf("%d", n); return 0;}There is also one more method is C that can be used to find the size of an array in C.Using Pointer ArithmeticThe idea is to use pointer arithmetic to find the difference between the memory addresses of the first element of the array and the beyond the last which will be then automatically scaled by the compiler according to the type of array. C #include int main() { int arr[] = {10, 20, 30, 40, 50}; // Calculate the size int n = *(&arr + 1) - arr; printf("%d", n); return 0;}Note: All these methods only work as long as the array has not decayed into a pointer. Once it decays, it is treated as a pointer, and the size of the array can no longer be determined. I would advise to never use sizeof (even if it can be used) to get any of the two different sizes of an array, either in number of elements or in bytes, which are the last two cases I show here. For each of the two sizes, the macros shown below can be used to make it safer. The reason is to make obvious the intention of the code to maintainers, and difference sizeof(ptr) from sizeof(arr) at first glance (which written this way isn't obvious), so that bugs are then obvious for everyone reading the code. TL;DR:#define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0]) + must_be_array(arr))#define ARRAY_BYTES(arr) (sizeof(arr) + must_be_array(arr))#define is_same_type(a, b) __builtin_types_compatible_p(a, b)#define is_same_typeof(a, b) is_same_type(typeof(a), typeof(b))#define decay(a) (&*(a))#define is_array(arr) (!is_same_typeof(arr, decay(arr)))#define must_be(e) \(\ 0 * (int)sizeof(\ struct { \ static_assert(e); \ char ISO_C_forbids_a_struct_with_no_members__; \ } \ ) \)#define must_be_array(arr) must_be(is_array(arr)) There have been important bugs regarding this topic: disagree with the solution that Linus provides, which is to never use array notation for parameters of functions.I like array notation as documentation that a pointer is being used as an array. But that means that a fool-proof solution needs to be applied so that it is impossible to write buggy code.From an array we have three sizes which we might want to know:The size of the elements of the arrayThe number of elements in the arrayThe size in bytes that the array uses in memory The size of the elements of the arrayThe first one is very simple, and it doesn't matter if we are dealing with an array or a pointer, because it's done the same way.Example of usage:void foo(size_t nmemb, int arr[nmemb]){ qsort(arr, nmemb, sizeof(arr[0]), cmp);}qsort() needs this value as its third argument. For the other two sizes, which are the topic of the question, we want to make sure that we're dealing with an array, and break the compilation if not, because if we're dealing with a pointer, we will get wrong values. When the compilation is broken, we will be able to easily see that we weren't dealing with an array, but with a pointer instead, and we will just have to write the code with a variable or a macro that stores the size of the array behind the pointer. The number of elements in the arrayThis one is the most common, and many answers have provided you with the typical macro COUNTOF:#define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0]))Recent versions of compilers, such as GCC 8, will warn you when you apply this macro to a pointer, so it is safe (there are other methods to make it safe with older compilers).It works by dividing the size in bytes of the whole array by the size of each element.Examples of usage:void foo(size_t nmemb){ char buf[nmemb]; fgets(buf, COUNTOF(buf), stdin);}void bar(size_t nmemb){ int arr[nmemb]; for (size_t i = 0; i < COUNTOF(arr); i++) arr[i] = i;}If these functions didn't use arrays, but got them as parameters instead, the former code would not compile, so it would be impossible to have a bug (given that a recent compiler version is used, or that some other trick is used), and we need to replace the macro call by the value:void foo(size_t nmemb, char buf[nmemb]){ fgets(buf, nmemb, stdin);}void bar(size_t nmemb, int arr[nmemb]){ for (size_t i = nmemb - 1; i < nmemb; i--) arr[i] = i;} The size in bytes that the array uses in memoryCOUNTOF is commonly used as a solution to the previous case, but this case is rarely written safely, maybe because it's less common.The common way to get this value is to use sizeof(arr). The problem: the same as with the previous one; if you have a pointer instead of an array, your program will go nuts.The solution to the problem involves using the same macro as before, which we know to be safe (it breaks compilation if it is applied to a pointer):#define ARRAY_BYTES(arr) (sizeof((arr)[0]) * COUNTOF(arr))How it works is very simple: it undoes the division that COUNTOF does, so after mathematical cancellations you end up with just one sizeof(arr), but with the added safety of the COUNTOF construction.Example of usage:void foo(size_t nmemb){ int arr[nmemb]; memset(arr, 0, ARRAY_BYTES(arr));}memset() needs this value as its third argument.As before, if the array is received as a parameter (a pointer), it won't compile, and we will have to replace the macro call by the value:void foo(size_t nmemb, int arr[nmemb]){ memset(arr, 0, sizeof(arr[0]) * nmemb);} Update (23/apr/2020): -Wsizeof-pointer-div doesn't warn from library code:Today I found out that the new warning in GCC only works if the macro is defined in a header that is not a system header. If you define the macro in a header that is installed in your system (usually /usr/local/include/ or /usr/include/) (#include), the compiler will NOT emit a warning (I tried GCC 9.3.0).So we have #define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0])) and want to make it safe. We will need C2X static_assert() and some GCC extensions: Statements and Declarations in Expressions, __builtin_types_compatible_p:#include #define is_same_type(a, b) __builtin_types_compatible_p(a, b)#define is_same_typeof(a, b) is_same_type(typeof(a), typeof(b))#define decay(a) (&*(a))#define is_array(arr) (!is_same_typeof(arr, decay(arr)))#define Static_assert_array(arr) static_assert(is_array(arr))#define COUNTOF(arr) \({ \ Static_assert_array(arr); \ sizeof(arr) / sizeof((arr)[0]); \})Now COUNTOF() is completely safe, and therefore all its derivatives will be safe. Update: Allow the macro to be used at file scope:Unfortunately, the ({}) gcc extension cannot be used at file scope.To be able to use the macro at file scope, the static assertion must beinside sizeof(struct {}). Then, multiply it by 0 to not affectthe result. A cast to (int) might be good to simulate a functionthat returns (int)0 (in this case it is not necessary, but then itis reusable for other things).Additionally, the definition of ARRAY_BYTES() can be simplified a bit.#include #include #include #include #define is_same_type(a, b) __builtin_types_compatible_p(a, b)#define is_same_typeof(a, b) is_same_type(typeof(a), typeof(b))#define decay(a) (&*(a))#define is_array(arr) (!is_same_typeof(arr, decay(arr)))#define must_be(e) \(\ 0 * (int)sizeof(\ struct { \ static_assert(e); \ char ISO_C_forbids_a_struct_with_no_members__; \ } \ ) \)#define must_be_array(arr) must_be(is_array(arr))#define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0]) + must_be_array(arr))#define ARRAY_BYTES(arr) (sizeof(arr) + must_be_array(arr)) This code makes use of the following extensions, which are completely necessary, and their presence is absolutely necessary to achieve safety. If your compiler doesn't have them, or some similar ones, then you can't achieve this level of safety.__builtin_types_compatible_p()typeof()I also make use of the following C2X feature. However, its absence by using an older standard can be overcome using some dirty tricks (see for example: What is :-!! in C code?) (in C11 you also have static_assert(), but it requires a message). The Length of an array in C refers to the maximum number of elements that an array can hold. It must be specified at the time of declaration. It is also known as the size of an array that is used to determine the memory required to store all of its elements.In C language, we don't have any pre-defined function to find the length of the array instead, you must calculate the length manually using techniques based on how the array is declared and used.The simplest method to find the length of an array is by using sizeof() operator to get the total size of the array and then divide it by the size of one element. Take a look at the below example: C #include int main() { int arr[5] = { 1, 2, 3, 4, 5 };// Find the size of the array arr int n = sizeof(arr) / sizeof(arr[0]); printf("%d", n); return 0;}Explanation: In this program, the total size of the array arr (20 bytes) is divided by the size of a single element in the array (4 bytes). This gives the number of elements in the array, which is 20/4 = 5Using Pointer Arithmetic TrickWe can also calculate the length of an array in C using pointer arithmetic. This solution of using a pointer is just a hack that is used to find the number of elements in an array. C #include int main(){ int arr[5] = { 1, 2, 3, 4, 5 }; // Find the size of array arr int n = *(&arr + 1) - arr; printf("%d", n); return 0;}Explanation: In the above code, in the expression: *(&arr + 1) - arr; is the main logic. Here is how it works,&arr gives the address of the entire array.&arr + 1 moves the pointer to the address just after the end of the array arr.*(&arr + 1) dereferences this pointer to get the memory location just after the array.Subtracting the base address of the array (arr) from this value gives the total number of elements in the array, as pointer arithmetic works based on the size of the elements (e.g., int in this case).Note: Please note that these methods only works when the array is declared in the same scope. These methods will fail if we try them on an array which is passed as a pointer. This happens due to Array Decay. So, it is recommended to keep a variable that tracks the size of the array.For more insights into array handling and other key data structures in C, our C programming course offers comprehensive lessons on arrays, pointers, and memory management. When coding in the C programming language, there may be times when you need to know the size of an array.For example, when you want to loop through all the elements stored in the array to determine whether a specific value is present.In this article, you will learn how to find the size of an array using the sizeof() operator.Let's dive in! Arrays let you store multiple values under the same variable name.An array in the C programming language is a collection of items of the same data type. This means you can create an array of only integer values or an array of chars and so on.To create an array in C, you first need to specify the data type of the values the array will store.Then, you give the array a name followed by a pair of square brackets, [].Inside the square brackets, you can specify the size of the array.So, here is how you would create an array of type int called faveNumbers that will hold 5 integers:int faveNumbers[5];To insert values inside the array during its declaration, use the assignment operator, =, and a pair of curly braces, {}.Inside the curly braces, enter the items and separate each one with a comma:int faveNumbers[5] = {7, 33, 13, 9, 29};The code above creates an array with the name faveNumbers that holds 5 integers, 7, 33, 13, 9, 29.You could also write the code above as follows:int faveNumbers[] = {7, 33, 13, 9, 29};In the example above, I didn't specify the size of the array. However, the compiler can tell that the size is 5 since I initialized it with 5 elements.Something to note here is that you cannot change the size and type of the array once you declare it since they have a fixed length. C does not provide a built-in way to get the size of an array. With that said, it does have the built-in sizeof operator, which you can use to determine the size.The general syntax for using the sizeof operator is the following:datatype size = sizeof(array_name) / sizeof(array_name[index]);Let's break it down:size is the variable name that stores the size of the array, and datatype specifies the type of data value stored in size.sizeof(array_name) calculates the size of the array in bytes.sizeof(array_name[index]) calculates the size of one element in the array.Now, let's see this operation in action and break it down into individual steps to see how it works.Firstly, the sizeof operator returns the total amount of memory allocated to the array in bytes. int main() { int faveNumbers[] = {7, 33, 13, 9, 29}; size_t size = sizeof(faveNumbers); printf("The size is %d bytes ", size);}However, the code above doesn't calculate the size of the array directly. You will need some extra programming logic, which will look something like this:array_length = (total size of the array) / (size of the first element in the array)To find the length of the array, you need to divide the total amount of memory by the size of one element - this method works because the array stores items of the same type. So, you can divide the total number of bytes by the size of the first element in the array.To access the first element in an array, specify the name and, in square brackets, include 0. In programming and Computer Science in general, indexing always starts at 0, so the first element in an array will always have an index of 0. int main() { int faveNumbers[] = {7, 33, 13, 9, 29}; size_t size = sizeof(faveNumbers) / sizeof(faveNumbers[0]); printf("The length of the array is %d ", size);}Something to note here is the size of data types will vary from platform to platform. Hopefully, this article helped you understand how to find the size of an array in the C programming language using the built-in sizeof operator.Thank you for reading, and happy coding! I would advise to never use sizeof (even if it can be used) to get any of the two different sizes of an array, either in number of elements or in bytes, which are the last two cases I show here. For each of the two sizes, the macros shown below can be used to make it safer. The reason is to make obvious the intention of the code to maintainers, and difference sizeof(ptr) from sizeof(arr) at first glance (which written this way isn't obvious), so that bugs are then obvious for everyone reading the code. TL;DR:#define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0]) + must_be_array(arr))#define ARRAY_BYTES(arr) (sizeof(arr) + must_be_array(arr))#define is_same_type(a, b) __builtin_types_compatible_p(a, b)#define is_same_typeof(a, b) is_same_type(typeof(a), typeof(b))#define decay(a) (&*(a))#define is_array(arr) (!is_same_typeof(arr, decay(arr)))#define must_be(e) \(\ 0 * (int)sizeof(\ struct { \ static_assert(e); \ char ISO_C_forbids_a_struct_with_no_members__; \ } \ ) \)#define must_be_array(arr) must_be(is_array(arr))#define COUNTOF(arr) (sizeof(arr) / sizeof((arr)[0]) + must_be_array(arr))#define ARRAY_BYTES(arr) (sizeof(arr) + must_be_array(arr)) This code makes use of the following extensions, which are completely necessary, and their presence is absolutely necessary to achieve safety. If your compiler doesn't have them, or some similar ones, then you can't achieve this level of safety.__builtin_types_compatible_p()typeof()I also make use of the following C2X feature. However, its absence by using an older standard can be overcome using some dirty tricks (see for example: What is :-!! in C code?) (in C11 you also have static_assert(), but it requires a message).

Size of the array. Can you declare an array without array size.