Pointers examples in c pdf

I'm not robot

reCAPTCHA

Continue

Pointers are used anywhere in C, so if you want to fully use the C language, you should fully understand the pointer. They must be comfortable for you. This section and several of the following objectives will help you fully understand how pointers and C use pointers. For most people, it takes time and practice to fully familiar with pointers, but once you've mastered them, you're a full-fledged C programmer. C uses pointers in three different ways: the provision C uses pointers to create dynamic data structures. C uses a pointer to handle the variable parameters passed to the function. The C pointer provides another way to access the information stored in the array. Pointer techniques are especially useful when working with strings. There is a close link between the uses of C and the pointer. C programmers may use pointers to make their code a little more efficient. All you find is that once you're completely accustomed to pointers, you tend to use them all the time. This section begins with a basic introduction to pointers and the concepts surrounding them, and then move on to the three techniques described above. Especially in this article, you'll want to read it twice. You can learn all the concepts for the first time. The second time, you can work on the whole integrated in your mind by integrating concepts. After you make your way through the material for the second time, it makes a lot of sense. In C, you can create pointers to almost any type, including user-defined types. It is very common to create a pointer to a structure. Here's an example: the typedef structure, char name[21]; char city[21]; char state[3]; Reck;type-deflector-reck-pointer;r-looking-pointer-(wreck-pointer)-mallock(size of(wreck));pointer r is a pointer to a structure. Note the fact that r is a pointer, so it take 4 bytes of memory just like any other pointer. However, the malloc statement allocates 45 bytes of memory from the heap. It has the same structure as any other structure of type Rec. The following code shows the general use of pointer variables. strcpy(.r.City, Raleigh);strcpy(r.State, NC). Print (%s, ... Free(r);.r treats it like a normal structural variable, but you need to be aware of the precedence of the C operator. Enclose the parentheses. The operator is not compiled because it has a higher precedence than the <a0> </a0> operator. Because it is troublesome to enter so many parentheses when dealing with pointers to structures, C uses strcpy(r->name)r->The notation is exactly the same as ,r), but it is 2 characters less. Pointer to array You can also create an array pointer to an array as follows: int i;p , (int) malloc (size of(int[10])) (i_0; i <10; i++) free(p);&><<<> <10; free(p);note-that-when-when-you-create-a-pointer-to-an-integer-array,-you-simple-create-a-normal-pointer-to-int.-the-call-malloc-alloc <a0></a0><a1></a1> <a2></a2> <a3></a3><a4></a1><a2></a2> <a3></a3><a4></a4><a5></a5><a5></a5></a5></a5><a5></a5></a1><a2></a2><a4></a4></a1><a2></a2> <a3></a3><a4></a1><a2></a2> <a3></a3></a1> <a2></a2> <a3></a3><a4></a5><a5></a5></a1><a2></a2> <a3></a4> <a4></a5></a1> <a2></a2> <a3>< using-pointer-arithmetic.-c-sees-both-forms-as-equivalent.-this-partial--technologie-is-extreme-weeful-when-working-with ly-hold-a-string-of-a-partial-size.-arrays-of-points-sometimes-a-great-deal-deal-of-space-can-be-saved,-or-by-declering-an-array-of-pointers.-in-the-example-code-below,-an-array-of-10-points-to-structures-is-destroyed,--the-structures-had-been-created-instead,-243--- -- minimal-space-until-the-simultaneous-records-are-allocated-with-mayloc-statements.-the-code-below-simple-allocates-one-record,-places-a-value-in-it,--- def_struct-s1[81];char-s2[81];;char-s3[81];;;-rec;-rec-a[10];a[0];(Rec)malloc(sizeof(rec);strcpy(a[0]-->s1, hello); free (a[0]);Structures containing pointer structures can contain pointers, as follows: Addr; Addr s;Chalcom[100]; get (s.name, 20). Get (s.city,20); get (s.phone, 20); get (communication, 100); s.comment . strcpy(s.comment, communication); If there are no comments in the record, the comment field consists of only a pointer (4 bytes). Records with comments are allocated enough space to hold the comment string based on the length of the string entered by the user. Arrays and pointers are closely linked in C. To use arrays effectively, you need to know how to use pointers with arrays. A full understanding of the relationship between the two will probably require several days of research and experiments, but it's well worth the effort. Let's start with a simple example of an array of C: advertisements , define MAX 10 int main() , int a[MAX]; int b[MAX]; <MAX; i++) a[i]-i; b-a; Return 0; Enter this code and try to compile it. You will find that C will not compile i++) b-a; return, 0;, enter, this, code, and, try, to, compile, it. i++) a[i]-i; b-a; Return 0; Enter this code and try to compile it. You Will Find That C Will Not Compile > int i; for(i_0; i</10;>If you copy a to b, you must enter something like this instead of (i_0; i<MAX; i++) b[i]-a[i]: Or, more simply, (i_0;i<MAX;b[i]-a[i], i++) Better yet, use the memppy utility in string.h. The array of C is unusual because the variables a and b are technically not the array itself. Instead, it is a persistent pointer to an array. a and b permanently point to the first element of each array and hold the addresses a[0] and b[0], respectively. Because these are persistent pointers, you cannot change the address. The statement a-b; therefore does not work. Because a and b are pointers, you can use pointers and arrays to do interesting things. For example, the following code #define> max 10 void main() int a[MAX] ; int i; int .p; p-a; for(i_0; i<MAX; i++) a[i]-i; printf(%d, Technically, it refers to the address of the 0th element of the actual array. Since this element is an integer, a is a pointer to a single integer. Therefore, declare p as a pointer to an integer and set it equal to the work. Another way to say exactly the same thing is to replace p along with p&a[0];; Because a contains the address of a[0], and a &a[0] have the same meaning. Now that p points to the 0th element of a, you can do pretty weird things with it. The a variable is a persistent pointer and cannot be changed, but p is not subject to such restrictions. C is actually recommended to be moved using pointer arithmetic. For example, p++; The compiler recognizes that p points to an integer, so this statement increments the appropriate number of bytes and moves to the next element in the array. If p points to an array of 100-byte-length structures, move p++;p by 100 bytes. C processes the details of the element size. You can also use a pointer to copy array a to b. The following code can replace (i_0;i<MAX;a[i]-b[i], i++): : p-a;q-b; i<MAX; i++) <a0></a0> <a1></a1><a2></a2><a3></a3><a5></a5><a5></a5><a5></a q_b;(i i<MAX; i++) ,q++ , p++; what if it exceeds the end of array a or b using the pointer p or q(p-a,q_b,i_0; i<MAX; C doesn't care -- it increments p and q brightly and abandons and copies it on top of other variables. Because C assumes that you understand what you are doing, you should be careful when indexing arrays in C. There are two ways to pass arrays such as a and b to a function: Imagine a function dump that accepts an array of integers as a parameter and prints the contents of the array to stdout. There are two ways to code a dump: void dump (int a[a], int ia) (i_0; i<ia; i++) printf (%d,a[i]); or: void dump (int . . . . (i_0; i<ia; i++)If you know the size of the array, you number_in_array a near (optional) variable. Note that only pointers to arrays are passed to the function, not the contents of the array. The C function can also accept a variable-size array as a parameter. Ivan Kuten is a co-owner and CTO of Promwad Electronics Design House Some of you may already be tired of this topic, but our software engineers have chosen seven examples and tried to explain their behavior using the standard (the latest draft at the time of writing): structure A int data_mem;void non_static_mem_fn static_mem_fn />/2>/foo(p, 5);)//>>>>data_mem;//int data_mem b non_static_mem_fn One obvious point is that p initialized with a static_mem_fn null pointer cannot point to a valid object. Example 1 is an expression statement that is a discard value expression that still needs to be evaluated (stmt.expr In the definition (expr.unary.op It is clear what semantics are, but it is not clear whether there is a prerequisite that an object must exist. The null pointer has never been mentioned. Since basic.stc's is an indirection through an invalid pointer value, it is possible to try to draw a conclusion from the fact that it performs an indirection. has undefined behavior. However, the paragraph contains the definition of an invalid pointer value and refers to basic.compound There is also a note that the only way to create such a reference is to bind to an object obtained indirectly through a null pointer, resulting in undefined behavior, but it is not clear which part of the last clause refers to. For binding, binding to an object that does not exist is undefined and follows the standard text of the paragraph. The standard leaves room for interpretation rather than clarifying this particular topic, so let's look at the list of core language issues where the core working group details the standard language. For topics where the CWG has reached an informal consensus (i.e., how the draft status is defined), p is essentially not an error. Converting an astrid value to a left-hand value gives undefined behavior. If you don't hear enough unofficial consensus, there's another problem that specializes in Example 7.Should be allowed for that exact reason. This consensus should be taken into account in the following: If future standards prohibit indirection through null pointers (N2176, 6.5.3.2, footnote 104) as in C, all examples will be rendered with undefined behavior. Example 2 In order to call foo(), the parameter must be initialized, which leads to the evaluation of the operator's comma. Operands are evaluated from left to right and are all discarded value expressions (expr.comma Therefore, this example is well-shaped. Example 3 You must select an implicit copy constructor, initialize it, and initialize it with a valid object to call const A&. But in our case there is no such object. Example 4 The expression in this expression statement is converted to .data_mem(p)) for each expr.ref2 that specifies the corresponding member sub-object of the object specified in the first expression (expr.ref It is not clear once again whether there are any prerequisites for the object to exist. If you look at the reference and specify used in the same sense in basic.lookup.qual I would say this example is well-shaped for that, but some compilers disagree. For more information, see Checking Constant Expressions at the end of this article. Example 5 In the previous example, instead of discarding int, initialize it with the result of an expression. Expressions in this category must be converted to prvalue to initialize the object (basic.lval Because the target type is int, the result of the expression is accessed because the condition of basic.lval Example 6class.mfct.non-static.1 states, A non-static member function may be called on an object of that class type or on an object of a class type derived from that class type. Therefore, because there are no objects, the behavior is undefined. Example 7 As described in Example 1, CWG explains that this example is valid code. The only thing to add is to perform indirect execution through the expression to the left of ->, even if you don't need the result (footnote 59). Constant expressions checked using constant expressions cannot rely on undefined behavior (expr.const Diagnosis is not ideal, but at least sometimes it is correct. Edited a few examples to fit a constant evaluation, supplied them to three popular compilers, and commented out examples that were considered bad for GCC and MSVC diagnostic messagesMany of these specific examples are desired. The test itself can be found in Godbold, and a summary of the results is shown in the table below. The result raises a little question about the conclusion of Example 6 and even the conclusion of Example 4. But it's also interesting to see all of us share the same opinion about important examples 1.C++ Thank you for being with me to follow the null pointer adventure!-) We usually share fragments of code from current firmware development projects, but this time the examples were synthesized because software engineers were genuinely interested in philosophical questions. If you want to share our love contradictions in C++, feel free to share your code and comments. Previously published at get a daily round-up of the top tech story!