Why use double pointer? or Why use pointers to pointers?

When should a double pointer be used in C? Can anyone explain with a example?

What I know is that a double pointer is a pointer to a pointer. Why would I need a pointer to a pointer?

c pointers

edited Jul 18 '14 at 15:30
Arash Milani
4,105 1 24 39

asked Apr 7 '11 at 12:08
manju
568 2 5 4

5 Be careful; the phrase "double pointer" also refers to the type `double*` . – Keith Thompson Oct 19 '16 at 2:02

15 Answers

If you want to have a list of characters (a word), you can use `char *word`

If you want a list of words (a sentence), you can use `char **sentence`

If you want a list of sentences (a monologue), you can use `char ***monologue`

If you want a list of monologues (a biography), you can use `char ****biography`

If you want a list of biographies (a bio-library), you can use `char *****biolibrary`

If you want a list of bio-libraries (a ??lol), you can use `char ******lol`

... ...

yes, I know these might not be the best data structures

answered Apr 7 '11 at 12:23
pmg
72k 7 86 146

4 lol............ – amanuel2 Jun 29 '16 at 20:41

21 Perhaps the most brain friendly,down-to-earth,real-world, and humorous explanation of pointer of all time in the history of pointers. – SittingBull Jul 16 '16 at 14:15

8 I wish all Stack Overflow was this readable – Definity Aug 17 '16 at 12:25

4 One my of similar (I was trying to be humouros) answers was deleted saying "StackOverflow doesn't like fun." :/ – Dilip Raj Baral Aug 17 '16 at 18:08

@pmg Can you please help me on this related query? Am not satisfied with your answer. – overexchange

Oct 19 '16 at 2:53

One reason is you want to change the value of the pointer passed to a function as the function argument, to do this you require pointer to a pointer.

In simple words, **Use `**` when you want to preserve (OR retain change in) the Memory-Allocation or Assignment even outside of a function call.** (So, Pass such function with double pointer arg.)

This may not be a very good example, but will show you the basic use:

```
void allocate(int** p)
{
  *p = (int*)malloc(sizeof(int));
}

int main()
{
  int* p = NULL;
  allocate(&p);
  *p = 42;
  free(p);
}
```

edited Jun 28 '13 at 20:29
bharat.chandak100
38 1 10

answered Apr 7 '11 at 12:11
Asha
7,337 1 29 50

4 what would be different if allocate were `void allocate(int *p)` and you called it as `allocate(p)` ? – アレックス Sep 7 '14 at 20:03

@AlexanderSupertramp Yes. The code will segfault. Please see Silviu's Answer. – Abhishek Sep 30 '14 at 5:37

Here is a SIMPLE answer!!!!

- lets say you have a pointer that its value is an address.
- but now you want to change that address.
- you could, by doing pointer1 = pointer2, and pointer1 would now have the address of pointer2.
- BUT! if you want a function to do that for you, and you want the result to persist after the function is done, you need do some extra work, you need a new pointer3 just to point to pointer1, and pass pointer3 to the function.
- here is a fun example (take a look at the output bellow first, to understand!):

```
#include <stdio.h>

int main()
{

    int c = 1;
    int d = 2;
    int e = 3;
    int * a = &c;
    int * b = &d;
    int * f = &e;
    int ** pp = &a;  // pointer to pointer 'a'

    printf("\n a's value: %x \n", a);
    printf("\n b's value: %x \n", b);
    printf("\n f's value: %x \n", f);
    printf("\n can we change a?, lets see \n");
    printf("\n a = b \n");
    a = b;
    printf("\n a's value is now: %x, same as 'b'... it seems we can, but can we do it in a
function? lets see... \n", a);
    printf("\n cant_change(a, f); \n");
    cant_change(a, f);
    printf("\n a's value is now: %x, Doh! same as 'b'...  that function tricked us. \n",
a);

    printf("\n NOW! lets see if a pointer to a pointer solution can help us... remember
that 'pp' point to 'a' \n");
     printf("\n change(pp, f); \n");
    change(pp, f);
    printf("\n a's value is now: %x, YEAH! same as 'f'...  that function ROCKS!!!. \n",
a);
    return 0;
}
```

```
void cant_change(int * x, int * z){
    x = z;
    printf("\n ----> value of 'a' is: %x inside function, same as 'f', BUT will it be the
same outside of this function? lets see\n", x);
}

void change(int ** x, int * z){
    *x = z;
    printf("\n ----> value of 'a' is: %x inside function, same as 'f', BUT will it be the
same outside of this function? lets see\n", *x);
}
```

- and here is the output:

```
 a's value: bf94c204

 b's value: bf94c208

 f's value: bf94c20c

 can we change a?, lets see

 a = b

 a's value is now: bf94c208, same as 'b'... it seems we can, but can we do it in a
function? lets see...

 cant_change(a, f);

 ----> value of 'a' is: bf94c20c inside function, same as 'f', BUT will it be the same
outside of this function? lets see

 a's value is now: bf94c208, Doh! same as 'b'...  that function tricked us.

 NOW! lets see if a pointer to a pointer solution can help us... remember that 'pp' point
to 'a'

 change(pp, f);

 ----> value of 'a' is: bf94c20c inside function, same as 'f', BUT will it be the same
outside of this function? lets see

 a's value is now: bf94c20c, YEAH! same as 'f'...  that function ROCKS!!!.
```

edited Feb 24 '16 at 20:15
Guy Hughes
441 2 15

answered Jun 8 '15 at 20:00
Brian Joseph Spinos
549 1 6 15

I saw a very good example today, from this blog post, as I summarize below.

Imagine you have a structure for nodes in a linked list, which probably is

```
typedef struct node
{
    struct node * next;
    ....
} node;
```

Now you want to implement a `remove_if` function, which accepts a removal criterion `rm` as one of the arguments and traverses the linked list: if an entry satisfies the criterion (something like `rm(entry)==true`), its node will be removed from the list. In the end, `remove_if` returns the head (which may be different from the original head) of the linked list.

You may write

```
for (node * prev = NULL, * curr = head; curr != NULL; )
{
    node * const next = curr->next;
    if (rm(curr))
    {
        if (prev)  // the node to be removed is not the head
            prev->next = next;
        else       // remove the head
            head = next;
        free(curr);
    }
    else
        prev = curr;
    curr = next;
}
```

as your `for` loop. The message is, **without double pointers, you have to maintain a `prev` variable to re-organize the pointers**, and handle the two different cases.

But with double pointers, you can actually write

```
// now head is a double pointer
for (node** curr = head; *curr; )
{
    node * entry = *curr;
    if (rm(entry))
    {
        *curr = entry->next;
        free(entry);
    }
    else
        curr = &entry->next;
}
```

You don't need a `prev` now because **you can directly modify what `prev->next` pointed to**.

To make things clearer, let's follow the code a little bit. During the removal:

1. if `entry == *head` : it will be `*head (==*curr) = *head->next` -- `head` now points to the pointer of the new heading node. You do this by directly changing `head` 's content to a new pointer.
2. if `entry != *head` : similarly, `*curr` is what `prev->next` pointed to, and now points to `entry->next` .

No matter in which case, you can re-organize the pointers in a unified way with double pointers.

edited Aug 28 '14 at 13:00 answered Aug 3 '14 at 23:29

ziyuang
1,478 2 19 41

Adding to Asha's response, if you use single pointer to the example bellow (e.g. alloc1()) you will loose the reference to the memory allocated inside the function.

```
void alloc2(int** p) {
   *p = (int*)malloc(sizeof(int));
   **p = 10;
}

void alloc1(int* p) {
   p = (int*)malloc(sizeof(int));
   *p = 10;
}

int main(){
   int *p;
   alloc1(p);
   //printf("%d ",*p);//value is undefined
   alloc2(&p);
   printf("%d ",*p);//will print 10
   free(p);
   return 0;
}
```

answered Sep 23 '14 at 20:33

Silviu
101 1 2

What happens if p is static integer pointer? Getting Segmentation fault. – kapilddit Sep 12 '16 at 18:41

Pointers to pointers also come in handy as "handles" to memory where you want to pass around a "handle" between functions to re-locatable memory. That basically means that the function can change the memory that is being pointed to by the pointer inside the handle variable, and every function or object that is using the handle will properly point to the newly relocated (or allocated) memory. Libraries like to-do this with "opaque" data-types, that is data-types were you don't have to worry about what they're doing with the memory being pointed do, you simply pass around the "handle" between the functions of the library to perform some operations on that memory ... the library functions can be allocating and de-allocating the memory under-the-hood without you having to explicitly worry about the process of memory management or where the handle is pointing.

For instance:

```
#include <stdlib.h>

typedef unsigned char** handle_type;

//some data_structure that the library functions would work with
typedef struct
{
    int data_a;
    int data_b;
    int data_c;
} LIB_OBJECT;

handle_type lib_create_handle()
{
    //initialize the handle with some memory that points to and array of 10 LIB_OBJECTs
    handle_type handle = malloc(sizeof(handle_type));
    *handle = malloc(sizeof(LIB_OBJECT) * 10);

    return handle;
}

void lib_func_a(handle_type handle) { /*does something with array of LIB_OBJECTs*/ }

void lib_func_b(handle_type handle)
{
    //does something that takes input LIB_OBJECTs and makes more of them, so has to
    //reallocate memory for the new objects that will be created

    //first re-allocate the memory somewhere else with more slots, but don't destroy the
    //currently allocated slots
    *handle = realloc(*handle, sizeof(LIB_OBJECT) * 20);

    //...do some operation on the new memory and return
}

void lib_func_c(handle_type handle) { /*does something else to array of LIB_OBJECTs*/ }

void lib_free_handle(handle_type handle)
{
    free(*handle);
    free(handle);
}


int main()
{
    //create a "handle" to some memory that the library functions can use
    handle_type my_handle = lib_create_handle();

    //do something with that memory
    lib_func_a(my_handle);

    //do something else with the handle that will make it point somewhere else
    //but that's invisible to us from the standpoint of the calling the function and
    //working with the handle
    lib_func_b(my_handle);

    //do something with new memory chunk, but you don't have to think about the fact
    //that the memory has moved under the hood ... it's still pointed to by the "handle"
    lib_func_c(my_handle);

    //deallocate the handle
    lib_free_handle(my_handle);

    return 0;
}
```

Hope this helps,

Jason

answered Apr 7 '11 at 14:52
Jason
24.4k 4 30 58

What is the reason for the handle type being unsigned char**? Would void** work just as well? – Hoten May 31 '16 at 5:44

3 `unsigned char` is specifically used because we're storing a pointer to binary data that will be represented as raw bytes. Using `void` will require a cast at some point, and is generally not as readable as to the intent of what is being done. – Jason Jun 7 '16 at 14:15

---

char *ch - Stores a single string..
char **ch - stores an array of Strings..
Here is the code..

```
#include <stdio.h>
#include <conio.h>

void func( char ** ptr)
{
    *ptr = malloc(255); // allocate some memory
    strcpy( *ptr, "Stack Overflow Rocks..!!");
}

main()
```

```
    {
        char *ptr = 0;
        func( &ptr );
        printf("%s\n", ptr);
        free(ptr);
        getch();
        return 0;
    }
```

One simpler example -

```
int main()
{
    char **p;
    p = (char **)malloc(100);
    p[0] = (char *)"Apple";      // or write *p
    p[1] = (char *)"Banana";     // or write *(p+1)

    cout << *p << endl;
    *p++;                        //Increments for the next string
    cout << *p;
}
```

edited Jun 11 '15 at 15:43

answered Jun 18 '13 at 9:47

Bhavuk Mathur
360 4 11

`#include <stdio.h> int main() { char *ptr = 0; ptr = malloc(255); // allocate some memory strcpy( ptr, "Stack Overflow Rocks..!!"); printf("%s\n", ptr); printf("%d\n",strlen(ptr)); free(ptr); return 0; }` But you can do it without using double pointer too. – kumar May 12 '14 at 12:25

Strings are a great example of uses of double pointers. The string itself is a pointer, so any time you need to point to a string, you'll need a double pointer.

answered Apr 7 '11 at 12:13

drysdam
4,846 1 12 20

For example, you might want to make sure that when you free the memory of something you set the pointer to null afterwards.

```
void safeFree(void** memory) {
    if (*memory) {
        free(*memory);
        *memory = NULL;
    }
}
```

When you call this function you'd call it with the address of a pointer

```
void* myMemory = someCrazyFunctionThatAllocatesMemory();
safeFree(&myMemory);
```

Now `myMemory` is set to NULL and any attempt to reuse it will be very obviously wrong.

answered Apr 7 '11 at 12:13

Jeff Foster
29.5k 6 58 88

1 it should be `if(*memory)` and `free(*memory);` – Asha Apr 7 '11 at 12:14

Good point, signal loss between brain and keyboard. I've edited it to make a bit more sense. – Jeff Foster Apr 7 '11 at 12:15

Why can't we do the following ... void safeFree(void* memory) { if (memory) { free(memory); memory = NULL; } } – Peter_pk Apr 27 '15 at 1:41

@Peter_pk Assigning memory to null wouldn't help because you've got passed a pointer by value, not by reference (hence the example of a pointer to a pointer). – Jeff Foster Apr 27 '15 at 17:07

For instance if you want random access to noncontiguous data.

```
p -> [p0, p1, p2, ...]
p0 -> data1
p1 -> data2
```

-- in C

```
T ** p = (T **) malloc(sizeof(T*) * n);
p[0] = (T*) malloc(sizeof(T));
p[1] = (T*) malloc(sizeof(T));
```

You store a pointer `p` that points to an array of pointers. Each pointer points to a piece of data.

If `sizeof(T)` is big it may not be possible to allocate a contiguous block (ie using malloc) of `sizeof(T) * n` bytes.

edited Apr 7 '11 at 12:23

answered Apr 7 '11 at 12:17

log0
7,734 2 15 50

One thing I use them for constantly is when I have an array of objects and I need to perform lookups (binary search) on them by different fields.
I keep the original array...

```
int num_objects;
OBJECT *original_array = malloc(sizeof(OBJECT)*num_objects);
```

Then make an array of sorted pointers to the objects.

```
int compare_object_by_name( const void *v1, const void *v2 ) {
  OBJECT *o1 = *(OBJECT **)v1;
  OBJECT *o2 = *(OBJECT **)v2;
  return (strcmp(o1->name, o2->name);
}

OBJECT **object_ptrs_by_name = malloc(sizeof(OBJECT *)*num_objects);
  int i = 0;
  for( ; i<num_objects; i++)
    object_ptrs_by_name[i] = original_array+i;
  qsort(object_ptrs_by_name, num_objects, sizeof(OBJECT *), compare_object_by_name);
```

You can make as many sorted pointer arrays as you need, then use a binary search on the sorted pointer array to access the object you need by the data you have. The original array of objects can stay unsorted, but each pointer array will be sorted by their specified field.

answered Apr 7 '11 at 13:41

DavidMFrey
1,302 1 10 13

I have used double pointers today while I was programming something for work, so I can answer why we had to use them (it's the first time I actually had to use double pointers). We had to deal with real time encoding of frames contained in buffers which are members of some structures. In the encoder we had to use a pointer to one of those structures. The problem was that our pointer was being changed to point to other structures from another thread. In order to use the current structure in the encoder, I had to use a double pointer, in order to point to the pointer that was being modified in another thread. It wasn't obvious at first, at least for us, that we had to take this approach. A lot of address were printed in the process :)).

You SHOULD use double pointers when you work on pointers that are changed in other places of your application. You might also find double pointers to be a must when you deal with hardware that returns and address to you.

answered Nov 11 '14 at 17:06

Axenie Ionut
11 1

Hopefully the following example will clear some concepts regarding pointers and double pointers , their differences and usage in common scenarios.

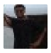
```
    int* setptr(int *x)
    {
        printf("%u\n",&x);
        x=malloc(sizeof(int));
        *x=1;
        return x;
    }

In the above function setptr we can manipulate x either
1. by taking fn arg as int *x , doing  malloc and setting value of x and return x
Or
```

```
    2. By taking arg as int ** and malloc and then set  **x value to some value.
Note: we cant set any general pointer directly without doing  malloc.Pointer indicates
that it is a type of variable which can hold address of any data type.Now either we define
a variable and give reference to it or we declare a pointer(int *x=NULL) and allocate some
memory to it inside the called function where we pass x or a reference to it .. In either
case we need to have address of a memory in the  pointer and in the case pointer initially
points  to NULL or it is defined like int *x where it points  to any random address then
we need to assign a valid memory address to pointer

    1. either we need to allocate memory to it by malloc

    int *x=NULL means its address is 0.
    Now we need to either o following
    1.



    void main()
        {
            int *x;
            x=malloc
            *x=some_val;
        }
        Or
        void main()
        {
            int *x
            Fn(x);
        }

        void Fn(int **x)
        {
            *x=malloc;
            **x=5;
        }
        OR
        int * Fn(int *x)
        {
            x=malloc();
            *x=4;
            Return x;
        }


        2. Or we need to point it to a valid memory like a defined variable inside the
function where pointer is defined.


        OR
        int main()
        {
            int a;
            int *x=&a;
            Fn(x);
            printf("%d",*x);
        }
        void Fn(int *x)
        {
            *x=2;
        }


     in both cases value pointed by x is changed inside fn

    But suppose if we do like


    int main()
    {
        int *x=NULL;
        printf("%u\n",sizeof(x));
        printf("%u\n",&x);
        x=setptr(x);
        //*x=2;
        printf("%d\n",*x);
        return 0;
    }

/* output
4
1
*/

#include<stdio.h>
void setptr(int *x)
{
    printf("inside setptr\n");
    printf("x=%u\n",x);
    printf("&x=%u\n",&x);
    //x=malloc(sizeof(int));
    *x=1;
    //return x;
}
int main()
```

```
{
    int *x=NULL;
    printf("x=%u\n",x);
    printf("&x=%u\n",&x);
    int a;
    x=&a;
    printf("x=%u\n",x);
    printf("&a=%u\n",&a);
    printf("&x=%u\n",&x);
    setptr(x);
    printf("inside main again\n");

    //*x=2;
    printf("x=%u\n",x);
    printf("&x=%u\n",&x);
    printf("*x=%d\n",*x);
    printf("a=%d\n",a);
    return 0;
}
```

edited Apr 27 '15 at 1:51

answered Apr 27 '15 at 1:45



Peter_pk
112 1 8

Why double pointers?

The objective is to change what studentA points to, using a function.

```
#include <stdio.h>
#include <stdlib.h>


typedef struct Person{
    char * name;
} Person;

/**
 * we need a ponter to a pointer, example: &studentA
 */
void change(Person ** x, Person * y){
    *x = y; // since x is a pointer to a pointer, we access its value: a pointer to a
Person struct.
}

void dontChange(Person * x, Person * y){
    x = y;
}

int main()
{

    Person * studentA = (Person *)malloc(sizeof(Person));
    studentA->name = "brian";

    Person * studentB = (Person *)malloc(sizeof(Person));
    studentB->name = "erich";

    /**
     * we could have done the job as simple as this!
     * but we need more work if we want to use a function to do the job!
     */
    // studentA = studentB;

    printf("1. studentA = %s (not changed)\n", studentA->name);

    dontChange(studentA, studentB);
    printf("2. studentA = %s (not changed)\n", studentA->name);

    change(&studentA, studentB);
    printf("3. studentA = %s (changed!)\n", studentA->name);

    return 0;
}

/**
 * OUTPUT:
 * 1. studentA = brian (not changed)
 * 2. studentA = brian (not changed)
 * 3. studentA = erich (changed!)
 */
```

edited Nov 26 '16 at 23:54

answered Nov 26 '16 at 22:59

Brian Joseph Spinos
549 1 6 15

The following is a very simple C++ example that shows that if you want to use a function to set a pointer to point to an object, **you need a pointer to a pointer**. Otherwise, **the pointer will keep reverting to null**.

(A C++ answer, but I believe it's the same in C.)

(Also, for reference: Google("pass by value c++") = "By default, arguments in C++ are passed by value. When an argument is passed by value, the argument's value is copied into the function's parameter.")

So we want to set the pointer `b` equal to the string `a`.

```
#include <iostream>
#include <string>

void Function_1(std::string* a, std::string* b) {
  b = a;
  std::cout << (b == nullptr);  // False
}

void Function_2(std::string* a, std::string** b) {
  *b = a;
  std::cout << (b == nullptr);  // False
}

int main() {
  std::string a("Hello!");
  std::string* b(nullptr);
  std::cout << (b == nullptr);  // True

  Function_1(&a, b);
  std::cout << (b == nullptr);  // True

  Function_2(&a, &b);
  std::cout << (b == nullptr);  // False
}

// Output: 10100
```

What happens at the line `Function_1(&a, b);` ?

- The "value" of `&main::a` (an address) is copied into the parameter `std::string* Function_1::a`. Therefore `Function_1::a` is a pointer to (i.e. the memory address of) the string `main::a`.
- The "value" of `main::b` (an address in memory) is copied into the parameter `std::string* Function_1::b`. Therefore there are now 2 of these addresses in memory, both null pointers. At the line `b = a;`, the local variable `Function_1::b` is then changed to equal `Function_1::a` (= `&main::a`), but the variable `main::b` is unchanged. After the call to `Function_1`, `main::b` is still a null pointer.

What happens at the line `Function_2(&a, &b);` ?

- The treatment of the `a` variable is the same: within the function, `Function_2::a` is the address of the string `main::a`.
- But the variable `b` is now being passed as a pointer to a pointer. The "value" of `&main::b` (the **address of the pointer** `main::b`) is copied into `std::string** Function_2::b`. Therefore within Function_2, dereferencing this as `*Function_2::b` will access and modify `main::b`. So the line `*b = a;` is actually setting `main::b` (an address) equal to `Function_2::a` (= address of `main::a`) which is what we want.

If you want to use a function to modify a thing, be it an object or an address (pointer), you have to pass in a pointer to that thing. The thing that you *actually* pass in cannot be modified (in the calling scope) because a local copy is made.

(An exception is if the parameter is a reference, such as `std::string& a`. But usually these are `const`. Generally, if you call `f(x)`, if `x` is an object you should be able to assume that `f` *won't* modify `x`. But if `x` is a pointer, then you should assume that `f` *might* modify the object pointed to by `x`.)

edited Feb 23 at 1:13

answered Feb 22 at 19:55

jt117
6 3