Shares

How to fix 20 most frequent C pointer mistakes

Top 20 C pointer mistakes and how to fix them

May 1, 2018 // by Deb Haldar (https://www.acodersjourney.com/author/debohaldaryahoo-com/)

After I graduated college with a BS in Electrical Engineering, I thought that was the last time I was going to program in "C". I could not have been more wrong. Throughout various points in my career, I've encountered and wrangled with a decent amount of "C" code either due to legacy or portability reasons.

Pointers are the most complicated and fundamental part of the C Programming language. Most of the mistakes I've made in school assignments and production code is in handling pointers. So here is my attempt to catalog some of the common and not so common mistakes – something I ca refer back to the next time I have to write production code in C. Hope it helps you as well.

Mistake # 1: Omitting the pointer "*" character when declaring multiple pointers in same declaration

Consider the following declaration:

Consider the following declaration:

```
int* p1, p2;
```

Shares

It declares an *integer pointer p1* and an *integer p2*. More often than not, the intent is to declare two integer pointers.

In the test code below, the last line will result in a compile error "Error C2440 '=': cannot convert from 'int *' to 'int' "

```
int main()
{
  int* p1, p2;

  int n = 30;

  p1 = &n;

  p2 = &n; // error
}
```

This is a pretty basic mistake that most modern compilers will catch.

Recommended Fix:

Use the following declaration to declare two pointers of the same type:

```
int *p1, *p2;
```

Alternatively, use a typedef – for example,

```
typedef int* Pint;
```

and then, use this type when declaraing pointers:

```
Pint p1, p2; // yay - no funny * business !
```

Mistake # 2: Using uninitialized pointers

The usage of an uninitialized pointer typically results in program crashes if the pointer accesses memory it is not allowed to.

Consider the code below:

```
int main()
{
  int* p1; // p1 can point to any location in memory

  int n = *p1; // Error on debug builds

  printf("%d", n); // access violation on release builds
  return 0;
}
```

On debug builds in Visual Studio, you'll first get the following error:

```
Run-Time Check Failure #3 - The variable 'p1' is being used without being initialized.
```

followed by:

```
"Exception thrown: read access violation.

p1 was 0xCCCCCCCC."
```

```
p1 was 0xcccccccc.
```

0xcc is microsoft's debug mode marker for uninitialized stack memory.

On release builds, you'll encounter a runtime crash on the line :printf("%d", n);

```
"Unhandled exception thrown: read access violation. p1 was nullptr."
```

Recommended Fix:
Always initialize pointers to a valid value.

```
int main()
{
  int* p1; // p1 can point to any location in memory

  int m = 10;
  p1 = &m; // initialize pointer with a valid value

  int n = *p1; // No error on Debug

  printf("%d", n); // no access violation on release builds
  return 0;
}
```

Mistake # 3: Assigning a pointer to an uninitialized variable

This is more dangerous, IMHO, than an uninitialized pointer.In this case, unlike an uninitialized pointer, you won't get a crash. Instead it can lead to serious logic errors in your code.

Consider the code below:

```
int main()
{
  int* p1; // p1 can point to any location in memory

  int m;
  p1 = &m; // initialize pointer with an uninitialized variable

  int n = *p1;

  printf("%d", n); // huge negative number in debug and 0 in release on VC++
  return 0;
}
```

On debug builds, it'll result in a large negative number like "-858993460". In VC++, the result will be 0 but that is not guaranteed by the C standard (http://c0x.coding-guidelines.com/6.7.8.html). More specifically item 1652 in the referenced doc states that If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.

Recommended Fix:

Deceptively simple – do not assign pointers to uninitialized variables.

Mistake # 4: Assigning value to pointer variables

Another one of the novice errors where the IDE/compiler will most likely bail you out. Consider the code:

```
int main()
{
  int* p1; // p1 can point to any location in memory

  int m = 100;
  p1 = m; // error

  return 0;
}
```

Shares

The problem is that p1 can contain an address of an int and not the int value itself. You'll get a compiler error:

```
"Error  C2440  '=': cannot convert from 'int' to 'int *' "
```

Recommended Fix:

Assign the address of the integer variable to the pointer .

```
int main()
{
  int* p1; // p1 can point to any location in memory

  int m = 100;
  p1 = &m; // assign address of m to p1

  return 0;
}
```

Mistake # 5: Incorrect syntax for incrementing dereferenced pointer values

If the intent is to increment a variable pointed to by a pointer, the following code fails to achieve that.

```
int main()
{
  int* p1; // create a pointer to an integer
  int m = 100;
  p1 = &m; // assign address of m to p1

  *p1++; // ERROR: we did not increment value of m

  printf("%d\n", *p1);
  printf("%d\n", m);

  return 0;
}
```

In fact, p1 now points to an undefined memory location. When you run this code, you get the following output with the first line corresponding to the value at the address p1 points to.

```
-858993460
100
```

Recommended Fix:
To increment a dereferenced pointer, use :
(*p1)++;

Mistake # 6: Trying to deallocate stack memory using free()

Consider the code below where variable m is allocated on the stack.

```
int main()
{
  int* p1; // create a pointer to an integer
  int m = 100;
  p1 = &m;

  free(p1);//error - trying to free stack memory using free()
  return 0;
}
```

Shares

Attempting to free memory on the stack using the free() function throws an access violation.

```
"Unhandled exception at 0x0F7BFC79 (ucrtbased.dll) in CPointerMistakes.exe: 0xC0000005: Access violation reading location 0x47D2C000."
```

Memory on the stack(non-pointer variables) is done implicitly by the system. It is illegal to get memory from the stack and return it to the heap.

Recommended Fix:

Use free() to deallocate memory that has been previously allocated by malloc() or one of its variants. Always remember where the memory came from – stack or heap 🙂

Mistake # 7: Dereferncing the value of a pointer after it has been freed

Consider the following code – we allocate an integre pointer, use it , free the memory associated with the pointer and then try to use the pointer again. This'll end in undefined behavior – maybe crashes depending on the state of the system/platform.

```
int main()
{
  int* p1;

  if ((p1 = (int*)malloc(sizeof(int))) == NULL)
  {
    return 1;
  }

  *p1 = 99;
  free(p1);

  *p1 = 100; // BAD - undefined behavior

  return 0;
}
```

Fix:

Never use a pointer after it has been freed. A good practice is to set the pointer to NULL after it has been freed such that any attempt to use it again is caught by an access violation.A crash during development is better than undefined behavior after release 🙂

```
free(p1);
p1 = NULL;
```

Mistake # 8 : Double free()

Calling free() on a block of memory twice will lead to heap corruption. For example, the following code results in an unhandled exception indicating heap corruption using MS VC++:

```
int main()
{
  char* str1 = (char*)malloc(strlen("Thunderbird") + 1);
  strcpy_s(str1, strlen("Thunderbird") + 1, "Thunderbird");

  //...
  free(str1);  // first free
         //...
  free(str1); // double free
}
```

OUTPUT:

```
Unhandled exception at 0x77959D71 (ntdll.dll) in CPointerMistakes.exe: 0xC0000374: A heap has been corrupted (parameters:
0x7798D8D0).
```

Shares

This type of issue caused a security vulnerability in zlib which you can read about here (http://seclists.org/cert/2002/7).

Recommended Fix:

Do not free the same block of memory twice! Simply assign NULL to a pointer after it has been freed. Subsequent attempts to free a null pointer will be ignored by most heap managers.

```
char* str1 = (char*)malloc(strlen("Thunderbird") + 1);
strcpy_s(str1, strlen("Thunderbird") + 1, "Thunderbird");

//...
free(str1);  // first free
str1 = NULL;
```

Mistake # 9 : Not using sizeof() operator with malloc

If you're implementing something in C in this day and age, most likely you're doing it with platform portability in mind. The size of data types can vary across different platform architectures. If you write something like malloc(2), you might have trouble porting it across platforms.

Recommended Fix:
Always use sizeof(type) with malloc – for example:

```
malloc(sizeof(int))
```

Mistake # 10 : Using a pointer and sizeof() to determine the size of an array

In the code below, sizeof(arr) will correctly determine the size of the char array but a pointer to the array won't. The type of *cp is const char, which can only have a size of 1, whereas the type of arr is different: array of const char.

```
int main()
{
  const char arr[] = "hello";
  const char *cp = arr;

  printf("Size of arr %lu\n", (int)sizeof(arr));
  printf("Size of *cp %lu\n", (int)sizeof(*cp));

  return 0;
}
```

Recommended Fix:
Never use sizeof on a pointer to an array to determine the size of the array.

Mistake # 11 : Creating garbage objects using C pointers

You need a pointer to a memory location to free / deallocate that memory. If you re-assign a pointer and there is no other pointer pointing to that memory block, you cannot deallocate that previous memory block. This causes a memory leak.

Consider the code below:

```
int main()
{
  int* p = (int*)malloc(sizeof(int)); // Let's call this memory block 1
  *p = 5;

  p = (int*)malloc(sizeof(int)); // Now you have no way to delete memory block 1 !!!

  return 0;
}
```

Shares

“Memory block 1” is not inaccessible because we don’t have a pointer to it. Without having a pointer to a memory block, we cannot call free() on a block and we’ve created a garbage object in that block – in other words, we leaked memory.

Recommended Fix:

In general, it’s not a good idea to recycle pointer variables. Use new pointer variables where possible and remember to set a pointer variable to NULL right after it has been freed.

Mistake # 12 : Not Understanding the difference between shallow copy and deep copy

Given two pointers p and q, the assignment p = q does not copy the block of memory pointed to by q into a block of memory pointed to by p; instead it assigns memory addresses (so that both p and q point to the same memory location; changing the value of that memory location affects both pointers).

Consider the code below:

```
#include "stdafx.h"
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <string.h>

typedef struct {
  char *model;
  int capacity;
}Aircraft;

int main()
{
  Aircraft af1;
  Aircraft af2;
  Aircraft af3;

  // Initialize af1
  af1.model = (char*)malloc(strlen("Thunderbird") + 1);
  strcpy(af1.model, "Thunderbird");
  af1.capacity = 320;

  // Shallow copy, af2.modelNum points to the same int as af1.modelNum
  af2 = af1;

  // Modifying af2 will affect af1
  printf("%s\n", af1.model); // prints ThunderBird
  strcpy(af2.model, "BlackHawk");
  printf("%s\n", af1.model); // prints BlackHawk - when ThunderBird is expected

  // Deep Copy: If the intent is to get a copy of af1, use a deep copy - which basically
  // means a member-wise cloning of values
  af3.model = (char*)malloc(strlen("Thunderbird") + 1);
  strcpy(af3.model, af1.model);
  af3.capacity = af1.capacity;

  // Let's run the same test:
  strcpy(af1.model, "Thunderbird");
  printf("%s\n", af1.model);          // prints ThunderBird

  strcpy(af3.model, "BlackHawk");
  printf("%s\n", af1.model); // prints ThunderBird as expected

  //cleanup the heap allocated strings
  free(af1.model);
  free(af3.model);

  return 0;
}
```

Shares

OUTPUT:

```
Thunderbird
BlackHawk
Thunderbird
Thunderbird
```

So what just happened?

In the shallow copy case, af1 and af2 both points to the same memory location. Any change to the memory location via af2 is reflected when af1 is used.

In the deep copy case, when we modify af3 (which points to an entirely different memory block than af1), the memory block pointed by af1 is not affected.

Mistake # 13 : Freeing a memory block shared by two pointers using one of the pointers and subsequently trying to use the other pointer

In the code below,. str1 and str2 points to the same memory block – so when str1 is freed, essentially the memory block pointed to by str2 is freed. Any attempt to use str2 after str1 has been freed will cause undefined behavior. In the case of the program below – it'll print some garbage value.

```
int main()
{
  char* str1 = (char*)malloc(strlen("Thunderbird") + 1);
  strcpy(str1, "Thunderbird");

  char* str2 = str1;
  printf("%s\n", str1);

  // ... many lines of code
  free(str1);

  // .. many lines of code

  printf("%s\n", str2); // ERROR: memory pointed to by q has been freed via p - you have undefined behavior

  return 0;
}
```

OUTPUT:

```
Thunderbird
αf╓        // some garbage value
```

There's really no good way around this in C except to use static analyzers. If you're in C++, you can use shared_pointers (https://www.acodersjourney.com/2016/05/top-10-dumb-mistakes-avoid-c-11-smart-pointers/) – but use caution as advised in the linked article. . There's also a good discussion on Stackoverflow (https://stackoverflow.com/questions/799825/smart-pointers-safe-memory-management-for-c) on this topic.

Mistake # 14 : Trying to access memory locations not allocated by your code

If you have allocated a block of n objects, do not try to access objects beyond this block (which includes any objects in locations p+n and beyond)

Consider the code below:

Shares

```c
int main()
{
  const int SIZE = 10;
  double *doubleVals;

  if ((doubleVals = (double*)malloc(sizeof(double)*SIZE)) == NULL)
  {
    exit(EXIT_FAILURE);
  }

  doubleVals[SIZE - 1] = 20.21;
  printf("%lf\n", doubleVals[SIZE - 1]);

  doubleVals[SIZE] = 25.99; // Error - we've only allocated blocks through SIZE-1 - you're writing over memory you do not own
  printf("%lf\n", doubleVals[SIZE]);

  return 0;
}
```

The statement *doubleVals[SIZE] = 25.99* is essentially writing over memory it does not own – which can cause undefined behavior in programs.

Recommended Fix:

Always be aware of the bounds of memory allocated by your code and operate within those safe limits.

Mistake # 15 : Off by one errors when operating on C pointers

Given a block of memory of SIZE objects pointed to by p, the last object in the block can be retrieved by using another pointer q and setting it to (p+SIZE-1) instead of (p+SIZE).

Consider the code below:

```c
int main()
{
  const int SIZE = 10;
  double *p;

  if ((p = (double*)malloc(sizeof(double)*SIZE)) == NULL)
  {
    exit(EXIT_FAILURE);
  }

  for (int i = 0; i < SIZE; i++)
  {
    *(p + i) = i;
  }

  double *q = p;

  //Incorrectly Access the last element
  double lastVal = *(q + SIZE); // Error - the last element is at (q + SIZE - 1)
  printf("%lf\n", lastVal);

  // Correctly access the last element
  lastVal = *(q + SIZE - 1);
  printf("%lf\n", lastVal);

  return 0;
}
```

The first print statement incorrectly prints "0" while the last element is "9". The second print statement fixes it by accessing the last element at (q + SIZE – 1)

Recommended Fix:

Carefully apply the “off by one error” rules that you learnt for array access to pointers.

Shares

Mistake # 16 : Mismatching the type of pointer and type of underlying data

Always use the appropriate pointer type for the data. Consider the code below where a pointer to an integer is assigned to a short:

```
int main()
{
  int  num = 2147483647;
  int *pi = &num;
  short *ps = (short*)pi;
  printf("pi: %p  Value(16): %x  Value(10): %d\n", pi, *pi, *pi);
  printf("ps: %p  Value(16): %hx  Value(10): %hd\n", ps, (unsigned short)*ps, (unsigned short)*ps);
}
```

OUTPUT:

```
pi: 00DFFC44  Value(16): 7fffffff  Value(10): 2147483647
ps: 00DFFC44  Value(16): ffff  Value(10): -1
```

Notice that it appears that the first hexadecimal digit stored at address 100 is 7 or f, depending on whether it is displayed as an integer or as a short. This apparent contradiction is an artifact of executing this sequence on a little endian machine.If we treat this as a short number and only use the first two bytes, then we get the short value of –1. If we treat this as an integer and use all four bytes, then we get 2,147,483,647.

Recommended Fix:

Always use the correct pointer type for a specific data type – int* for int , double* for double etc.

Mistake # 17 : Comparing two pointers to determine object equality

Often we want to compare if the contents of two objects are same – for example check if two strings are equal.

In the code below, clearly the intent was to check if both strings are “Thunderbird”. But, we ended up comparing the memory addresses with the statement “str1 == str2”. Here str1 and str2 are essentially pointers to different memory addresses which holds the same string.

```
int main()
{
  char* str1 = (char*)malloc(strlen("Thunderbird") + 1);
  strcpy(str1, "Thunderbird");

  char* str2 = (char*)malloc(strlen("Thunderbird") + 1);
  strcpy(str2, "Thunderbird");

  if (str1 == str2)
  {
    printf("Two strings are equal\n");
  }
  else
  {
    printf("Two strings are NOT equal\n");
  }
}
```

The code can be made to work as intended, i.e., compare string contents by making the following changes:

```
if (strcmp(str1,str2) == 0) // Are the contents of the strings the same
{
  printf("Two strings are equal\n");
}
```

Recommended Fix:

Always remember to compare the contents of the memory location pointed to by pointers instead of comparing the address of pointer themselves.

Shares

Mistake # 18 : Thinking that C arrays are pointers

While C pointers and Arrays can be used interchangeably in most situations, they are not quite the same. Here's an example of where it is a recipe for access violation.

```
// File1.cpp

int global_array[10];


// File2.cpp

extern int *global_array;

int main()
{
  for (int i = 0; i < 10; i++)
  {
    global_array[i] = i; // Access Violation
  }

  return 0;
}
```

In File2.cpp, global_array is declared as an pointer but defined as an array in File1.cpp. At a high level, the compile generates different code for array indexing and access via pointer.

Recommended Fix:

Change the declaration so it does match the definition, like:

```
// File1.cpp

int global_array[10];


// File2.cpp

extern int global_array[];

int main()
{
  for (int i = 0; i < 10; i++)
  {
    global_array[i] = i; // NO Access Violation
  }

  return 0;
}
```

Note: A detailed discussion is beyond the scope of this article. The best explanation of this issue I found was in the section, "Chapter 4. The Shocking Truth: C Arrays and Pointers Are NOT the Same!" in Deep C Secrets (https://www.amazon.com/Expert-Programming-Peter-van-Linden/dp/0131774298/ref=sr_1_1?ie=UTF8&qid=1524791462&sr=8-1&keywords=Deep+C+secrets). It's a fantastic book if you really want to become an expert C programmer – highly recommended.

Mistake # 19 : Not clearing out sensitive heap data managed through pointers

Shares

When an application terminates, most operating systems do not zero out or erase the heap memory that was in use by your application. The memory blocks used by your application can be allocated to another program, which can use the contents of non-zeroed out memory blocks. Just imagine you asked for a security question from the user and stored it in heap memory – it's always a good idea to erase that memory block contents before returning the memory to the Operating System via free().

```
int main()
{
  char* userSecurityQuestion = (char*)malloc(strlen("First Pet?") + 1);
  strcpy_s(userSecurityQuestion, strlen("First Pet?") + 1, "First Pet?");

  //...
  // Done with processing security question - stored in secured db etc.

  // Now set the program memory to zero before returning memory back to OS
  memset(userSecurityQuestion, 0, sizeof(userSecurityQuestion));
  free(userSecurityQuestion);
}
```

Mistake # 20 : Not taking time to understand C function pointers

Functions pointers are used extensively in many large scale production system. It's also critical to understand more advanced concepts like callbacks, events in Win32 or lambdas in standard C++.

Here's an example of function pointer in linux kernel:

```
struct net_device_ops {
int                     (*ndo_init)(struct net_device *dev);
void                    (*ndo_uninit)(struct net_device *dev);
int                     (*ndo_open)(struct net_device *dev);
int                     (*ndo_stop)(struct net_device *dev);
netdev_tx_t             (*ndo_start_xmit) (struct sk_buff *skb,
struct net_device *dev);
```

If code like this makes your head swivel, no sweat – mine did too when i started my career. 🙂

The problem is that most college level C courses seldom does any deep exploration of function pointers, whereas once you're in industry, it's all over the place. Here is a good book that has an in-depth treatment of C function pointers :Understanding and Using C Pointers (https://www.amazon.com/Understanding-Using-Pointers-Techniques-Management/dp/1449344186/ref=sr_1_1?s=books&ie=UTF8&qid=1524791674&sr=1-1&keywords=understanding+c+pointers).

Final Thoughts

C is one of the oldest languages in use today. Pointers forms the heart and soul of C. Pointers are not only useful for writing production quality code but also in school for understanding the concepts behind self referential data structures like linked list and binary trees. Even if you are working in a high level language like Java or C#, an object is essentially a pointer. So, study pointers well because they keep showing up in coding interviews and tech screens – I wouldn't be surprised if you get a question similar to the code snippets in this article and asked "what's wrong with this piece of C code?".

Good luck !

« 35 things I learnt at Game Developer Conference (GDC) 2018

(https://www.acodersjourney.com/gdc-2018-35-key-insights/)

Shares

System Design Interview Concepts – CAP Theorem »

(https://www.acodersjourney.com/system-design-interview-cap-theorem/)

Comments Community 1 Login

Recommend Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Shares

Name

Aaron • 6 months ago
No mention of using double pointers? :)
^ | ˅ • Reply • Share ›

Deb Haldar Mod ➦ Aaron • 6 months ago
I should really do an addendum to this article - double pointers and 2D arrays are the solid candidates.Got any others in mind that i missed ?
^ | ˅ • Reply • Share ›

Aaron ➦ Deb Haldar • 6 months ago
3/4D arrays; arrays of arrays; pointers to arrays; so much fun :D
^ | ˅ • Reply • Share ›

Deb Haldar Mod ➦ Aaron • 6 months ago
Thanks Aaron - will create a supplemental article talking about those :)
^ | ˅ • Reply • Share ›

Neqste • 10 months ago
Hi, thank you very much for useful article.
Just a question, if you have a suggestion about books which cover common pitfalls and problems like in your topic, and can name it - it will be great.
^ | ˅ • Reply • Share ›

Deb Haldar ➦ Neqste • 9 months ago
Thanks Neqste. There are a few books like "C++ Gotchas" and " C++ Coding standards" that you might find useful. The best way to create your own list that you can review and learn from is using old code reviews in your company/team. A lot of the stuff i write about is pulled from either books or old code reviews where i learnt things the "hard way". Hope this helps !
1 ^ | ˅ • Reply • Share ›

Shaikh Ahmed • a year ago
Trying out the cases one by one, in my case using mingw-gcc (x86_64-8.1.0-posix-seh-rt_v6-rev0) as compiler with codeblocks IDE.

My compiler lets go of the first mistake of assigning address to a non pointer variable with just this warning:

Shares