COMPSCI 1XC3 - Computer Science Practice and Experience: Development Basics

Topic 8 - Pointers in C

NCC Moore

McMaster University

Fall 2020

Adapted from C: How to Program 8th ed., Deitel & Deitel

Working with Pointers

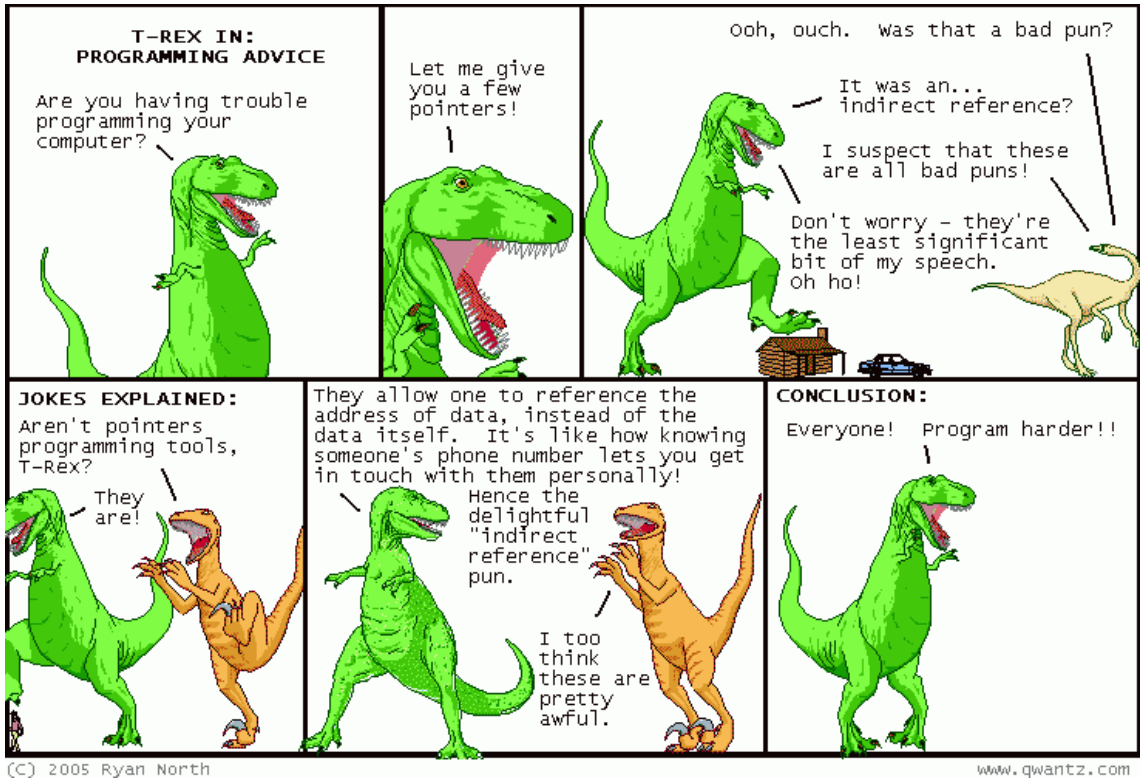
Pointer Operations!

The Wonderful World of Passing by Reference

Pointer Expressions and Arithmetic

Dynamic Memory Allocation

Pointers in C



"When a wise man points at the moon
the imbecile examines the finger"
– A person a lot of quotes get attributed to –

So what the heck's a pointer and why should I care?

Pointers are one of the most *powerful* constructs in C.

- A **pointer** is a variable whose value is a **memory address**.
- Where most variables directly reference the value stored at some position in memory, pointers are **indirect references**.
- In this set of slides we're going to talk about:
  - Pointer Data Types
  - Pointer Operations
  - Applications of Pointers

Pointerization!

A pointer is not declared as a new datatype, but as a modifier to existing data types.

```
1 int* ptr;
```

The * character indicates that ptr is an **integer pointer**.

- This is to say, ptr points to a segment of memory the size of an int.

The * character may be applied to either the data type or the identifier itself:

```
1 int *countPtr, count;
```

- * only pointerizes countPtr in this example.
- If * were applied to int, countPtr would still be the only variable that was pointerized.

In Blue Block-o-vision...

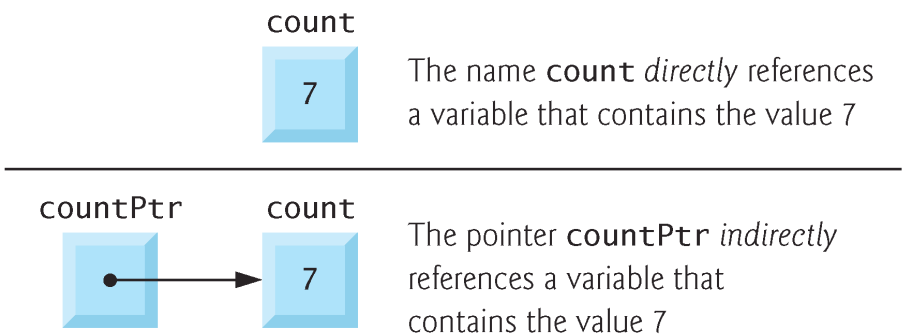


Fig. 7.1 | Directly and indirectly referencing a variable.

Pointer Facts

- You can create a pointer out of *any* datatype, including custom ones.
- * means something different when you're not declaring a pointer... *it's not part of the identifier!*
- Since pointers are a more direct manipulation of memory, you can squeeze out some efficiencies by squeezing in some pointers!

Stylistic Points

- To avoid the confusion on the previous slide, it's better to declare pointers and direct variables on separate lines.
- Putting some indication that a variable is a pointer in the identifier is a good way to be able to tell which variables are pointers later on.

Initial Pointing

Like a lot of things in C, uninitialized pointers contain junk data.

- A pointer will initially point to a random memory cell!
- Pointers should be initialized to either 0, `NULL` or a value that makes sense.
  - `NULL` is a **symbolic constant**, which is defined in a number of header files (such as `stdio.h` and `stddef.h`)
  - `NULL` is the same thing as zero, but it's preferred for stylistic reasons.

```
1 int* ptr = NULL;
2 if (ptr != NULL) {
3   // do stuff
4 }
```

We've done the nouns, here come the verbs...

Now we know how to store a memory address, but what good is it if we have no addresses to store?

- Memory Adresses are accessed using &, the "address of" operator.
- Applied to any identifier, it returns the physical memory address of that identifier.
- This includes pointers!

```
1 int y = 5;
2 int* yPtr;
3 // store the address of y in yPtr
4 yPtr = &y;
```

Address of Operator in Box-O-Vision

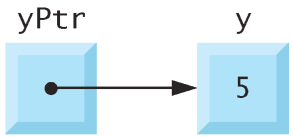


Fig. 7.2 | Graphical representation of a pointer pointing to an integer variable in memory.

Fig. 7.3 | Representation of `y` and `yPtr` in memory.

Dereferencing for Fun and Profit

The inverse operation of & is **pointer dereferencing**

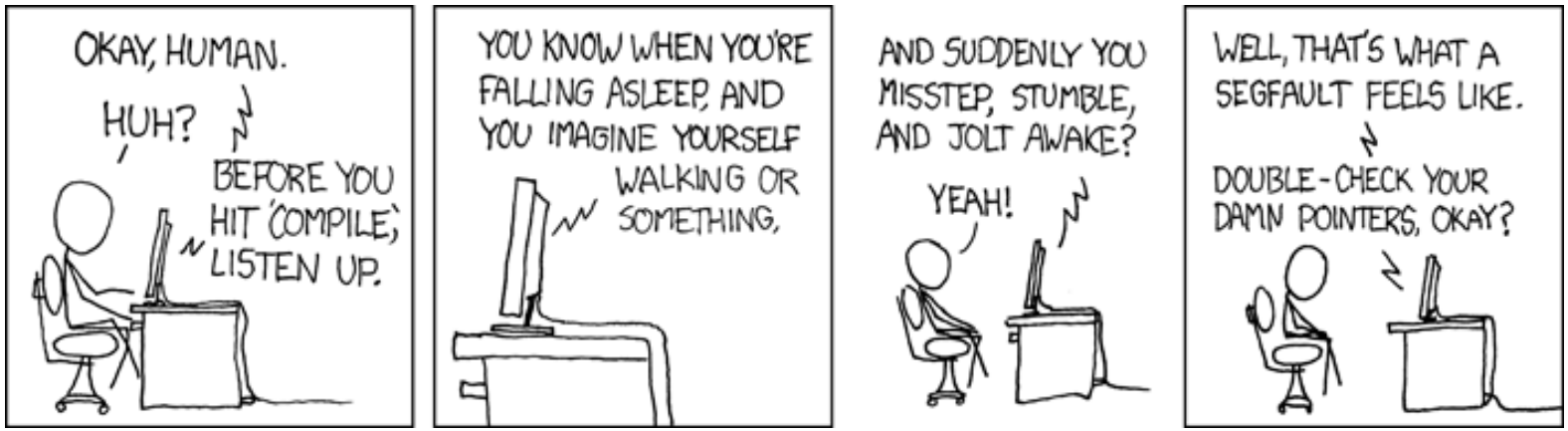
- This is a little confusing, but it's also *.
- Additionally, the formatter for pointers is %p.

```
1 int y = 5;
2 int* yPtr;
3 // store the address of y in yPtr
4 yPtr = &y;
5 printf("The pointer's value is %p\n", yPtr);
6 printf("The value pointed to is %d", *yPtr);
```

```
The pointer's value is 0x7ffe0fba22fc
The value pointed to is 5
```

The Deadly and Dreaded SEGFAULT

Dereferencing a pointer that points to memory outside your program's allocated memory space causes a fatal runtime error called a **segmentation fault**.



This is most common with `NULL` pointers, but can also happen if you mess up your pointer arithmetic.

& and * are Inverse Operations

```c
#include <stdio.h>
int main(void) {
  int a = 7;
  int *aPtr = &a;
  printf("a = %d\n", a);
  printf("&a = %p\n", &a);
  printf("aPtr = %p\n", aPtr);
  printf("*aPtr = %d\n", *aPtr);
  printf("Generating a memory address and dereferencing
     it\nleaves you where you started.\n");
  printf("*&a = %d\n", *&a);
  printf("The other way around causes an actual
    compiler error...\n");
//   printf("&*a = %d", &*a);
}
```

& and * are Inverse Operations (Output)

```
a = 7
&a = 0x7ffe08ed2edc
aPtr = 0x7ffe08ed2edc
*aPtr = 7
Generating a memory address and then dereferencing it
leaves you where you started.
*&a = 7
The other way around causes an actual compiler error...
```

Arrays of Pointers!

It can be easy to think of pointers entirely symbolically, and forget that they too have concrete values in memory.

- Pointers may be collected and organized in arrays, just like other data types.

```
1 const char *suit[4]
2     = {"Hearts", "Diamonds", "Clubs", "Spades"};
```

- Each element in the array is a pointer to a character array.
- The fact that our array contains pointers, instead of the character arrays themselves, means the character arrays' memory is managed separately from the array of pointers.
- Whereas a 2D array must be rectangular, each character array pointed to by the array of pointers may have a unique length!

Arrays of Pointers in Block-O-Vision

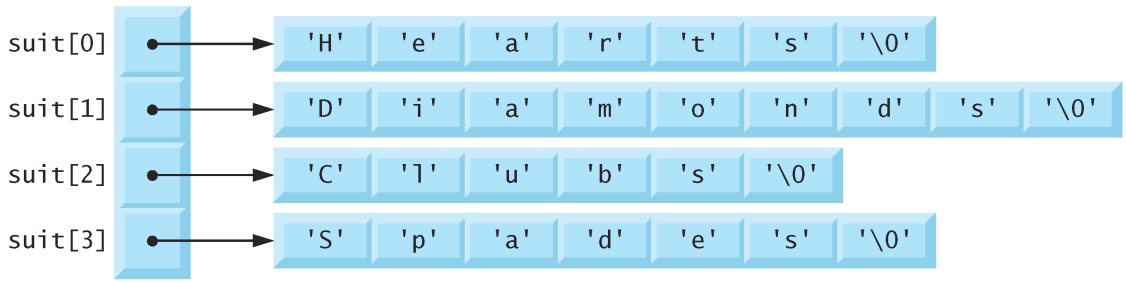


Fig. 7.22 | Graphical representation of the `suit` array.

Proud and Practical Pointer Passing

- All arguments are passed by value by C... Unless the value you pass is a memory address!
- Passing by reference allows functions to modify the referred data in the calling function, which can be useful!

```
1 int foo;
2 int bar[];
3 myFunc(&foo, bar);
```

- We use `&` to pass the address of `foo`, but `bar` is already a memory address.
- In C, `someArray` ≡ `&someArray[0]`.
- Inside the function definition, `foo` will need to be dereferenced, but `bar` will not.

Example: Cube By Reference

```c
#include <stdio.h>

void cubeByReference(int *nPtr);

int main(void) {
   int num = 5;
   printf("num = %d\n", num);
   cubeByReference(&num);
   printf("num = %d\n", num);
}

void cubeByReference(int *nPtr){
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

In Block-O-Vision...

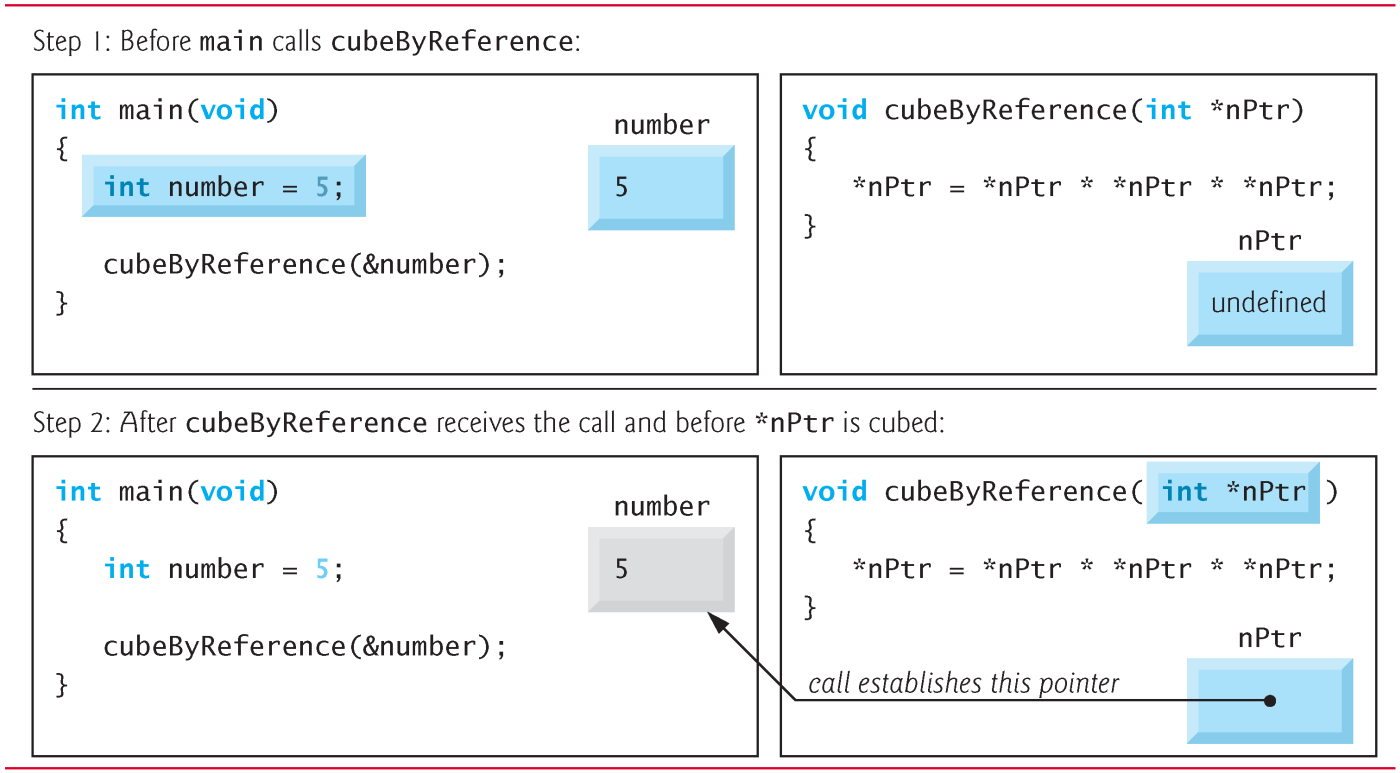


Fig. 7.9 | Analysis of a typical pass-by-reference with a pointer argument. (Part 1 of 2.)

In Block-O-Vision...

Step 3: After *nPtr is cubed and before program control returns to main:

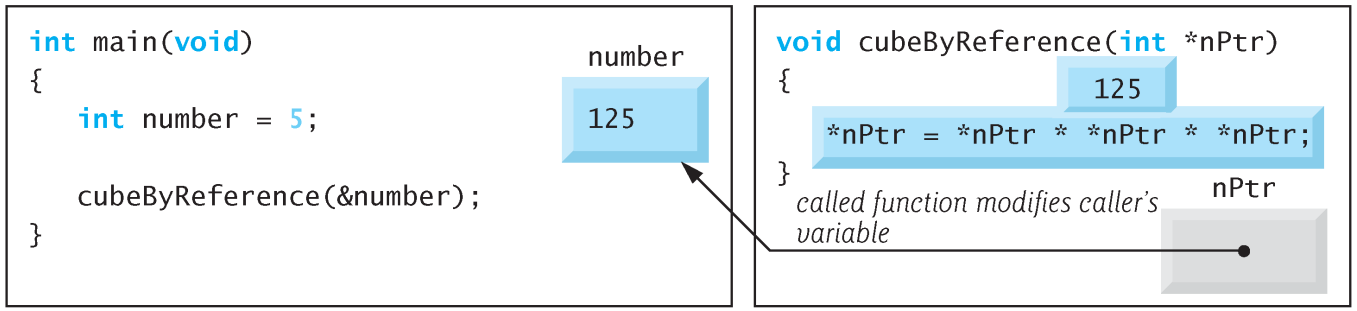


Fig. 7.9 | Analysis of a typical pass-by-reference with a pointer argument. (Part 2 of 2.)

Example: Cube By Reference

- In order to accept a memory address as an argument, the fact must be specified in the argument's type information.
- In this example, we replace the following operations:
  - Pass `num` by value to `cubeByReference`
  - Compute the cube and return it to `main`
  - Assign the return value of `cubeByReference` to `num`
- With this:
  - Pass the address of `num` to `cubeByReference`
  - Compute the cube and store it in the memory address directly
- Stylistically, pass by value is preferred unless the situation explicitly calls for pass by reference.
  - Passing by reference in this way violates the **principle of least privilege**.

Arrays Are Pointers!

The syntax for passing an array as an argument to a function is the same as passing a variable by reference.

- This is because the compiler does not differentiate between pointers and one-dimensional arrays.
  - Like so many things in C, that means it's your job!
  - It's up to you to write your functions so that they are using their arguments as intended!
  - Documentation becomes critical!
- An array is actually a pointer to it's own first element.
- We can perform arithmetic to traverse arrays without the indexing operator!

const and the Art of Mental Stillness

Problem: Whenever we work with pointers, there's a possibility of pointer misuse resulting in a segfault!

- We can prevent arguments from being modified by using the `const` qualifier.

```
1 void myFunc (const int *foo, float *bar);
```

- Trying to modify a `const` argument will result in the following compiler error using gcc:

```
error: assignment of read-only location
```

- Using `const` to restrict functions from modifying things they shouldn't modify is *good software design!*
- In general, a program should have only enough data access to accomplish it's task, and not one smidgeon more!

Pointer Arithmetic

No clever quote this time, though I did learn the internet seems to think "indirect" means "passive aggressive."

– The Prof –

A Program Illustrating Array Addressing

```
1  #include<stdio.h>
2
3  int main(void){
4    int foo[] = {0,1,2,3,4};
5    short int bar[] = {0.0,1.0,2.0,3.0,4.0};
6    printf("----------------------\n");
7    for (int i = 0; i < 5; i++) {
8      printf("&foo[%d] = %p\n", i, &foo[i]);
9    }
10   printf("----------------------\n");
11   for (int i = 0; i < 5; i++) {
12     printf("&bar[%d] = %p\n", i, &bar[i]);
13   }
14 }
```

Output of a Program Illustrating Array Addressing

```
-----------------
&foo[0] = 0x7fff2f07b190
&foo[1] = 0x7fff2f07b194
&foo[2] = 0x7fff2f07b198
&foo[3] = 0x7fff2f07b19c
&foo[4] = 0x7fff2f07b1a0
-----------------
&bar[0] = 0x7fff2f07b186
&bar[1] = 0x7fff2f07b188
&bar[2] = 0x7fff2f07b18a
&bar[3] = 0x7fff2f07b18c
&bar[4] = 0x7fff2f07b18e
```

In Block-O-Vision

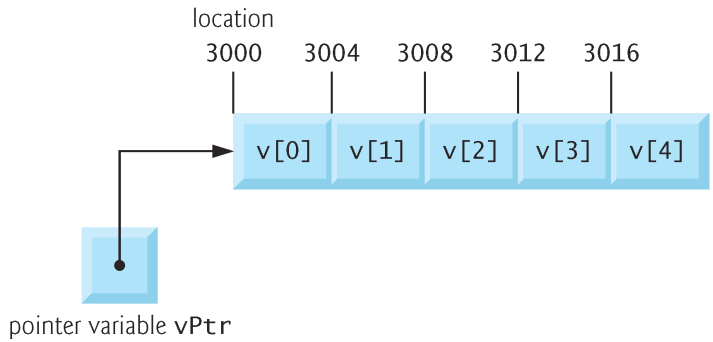


Fig. 7.18 | Array v and a pointer variable vPtr that points to v.

Our Old Friend ++

- Notice in the previous slide how `foo`'s memory addresses are 4 bytes apart, and `bar`'s are 2 bytes apart.
- The array is continuous memory, and each element is allocated the size of the base data type of the array.
- If we wish to perform pointer arithmetic to traverse an array, the compiler needs to how big the steps are!

Operators can mean different things when applied to arguments of different types.

- Float vs Int division, for example.
- `++`, when applied to a pointer, will automatically take the size of the data type its operating on!

Traversing an Array with Pointer Arithmetic

```c
#include<stdio.h>
int main(void){
  char foo[] = "Dated Reference";
  char *fooPtr = foo;
  while (*fooPtr != '\0') {
    printf("(%c)", *fooPtr++);
  }
  printf("\n");
}
```

Output:

```
(D)(a)(t)(e)(d)( )(R)(e)(f)(e)(r)(e)(n)(c)(e)
```

Other Pointer Operations

The following are valid operations on pointers:

- `++`, `--`, `+`, `-`, `+=`, `-=`

In each of these cases, the number you are adding/subtracting to/from the pointer is *implicitly multiplied by the byte-width of the data type*.

- For example, if `ptr` points to an `int`, then `ptr += 4` would move the pointer by 16 bytes, since the bit-width of an `int` is 4 bytes.

Pointers may also be subtracted from one another, but only meaningfully if they point to the same array.

- `ptrA - ptrB` yields the number of *array elements* difference between the two pointers, *not* the number of bytes difference.

A Word to the Wise...

- In the previous example tracing a character array, we used our knowledge that strings are null terminated to set a stopping condition for our loop.
- Pointer arithmetic can easily place a pointer outside the bounds of its original data structure.
- There is no in-built protection against out-of-bounds pointers, so we can easily use them to assign to memory outside the array bounds.
- This could overwrite other variables, cause segmentation faults, and many other troubles!
- Back in the day, this was a common exploit used to *hack the government!*.

Wildcard Pointers

Normally, pointers require compatible types to be assigned to each other.

- The exception to this is a **void pointer**

```
1 void *wildcard;
```

- This is a generic pointer that can point to any data type.
- A void pointer is compatible with all data types, and may be used in assignment operations freely
- The catch is a void pointer may not be dereferenced.
- This is because the dereferencing operation uses the byte width of the pointer's data type to select the area of memory to return.

Operator Miscellany

Equality and relational operators work on pointers!

- Relation operators are only meaningful if the pointers refer to the same data structure!
- Equality comparison with `NULL` is common.

Array indexing is actually syntactic sugar for pointer arithmetic!

- `foo[3]` $\equiv$ `*(foo + 3)`
- Therefore, it is also possible to index pointers in the same way as arrays!
- One of the few differences between pointers and array identifiers is that a array identifiers may not be assigned to.
  - We can think of them as *read only pointers*.

Dynamic Memory Allocation!

Up to now, our array sizes have been hard-coded (that is, set in the program code itself, not at runtime).

- In this section, we will learn how to allocate memory dynamically, allowing us to expand or contract arrays as needed at runtime.
  - When we use static arrays, these memory operations are *implicit*.
  - The general proceedure is to declare a pointer, invoke a memory allocation operation, and store the resultant memory address in the declared pointer.

We will learn about the following functions, contained in `stdlib.h`

- `malloc()`
- `free()`
- `calloc()`
- `realloc()`

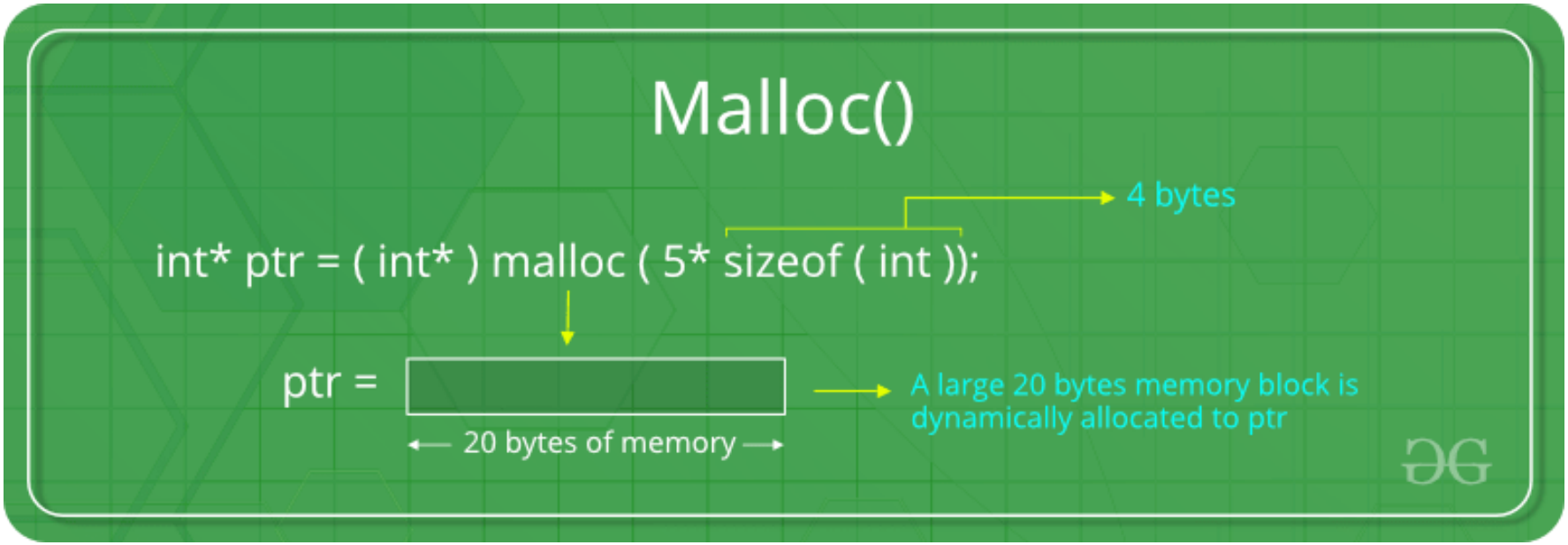
malloc() - Memory Allocation

Dynamic allocation of a single, large block of memory, given a specified size.

```
1 int *ptr = (int*) malloc(100*sizeof(int));
```

- malloc() accepts as an argument the number of bytes to allocate, expressed as an integer.
- It produces a void pointer, which may be type-cast to the type of the pointer you wish to store the address in.
- malloc() may fail, if the requested memory is larger than the available memory.
  - In this case, a NULL pointer will be returned.
  - You should always check a pointer returned from malloc() for null status before using it.

malloc() Broken Down

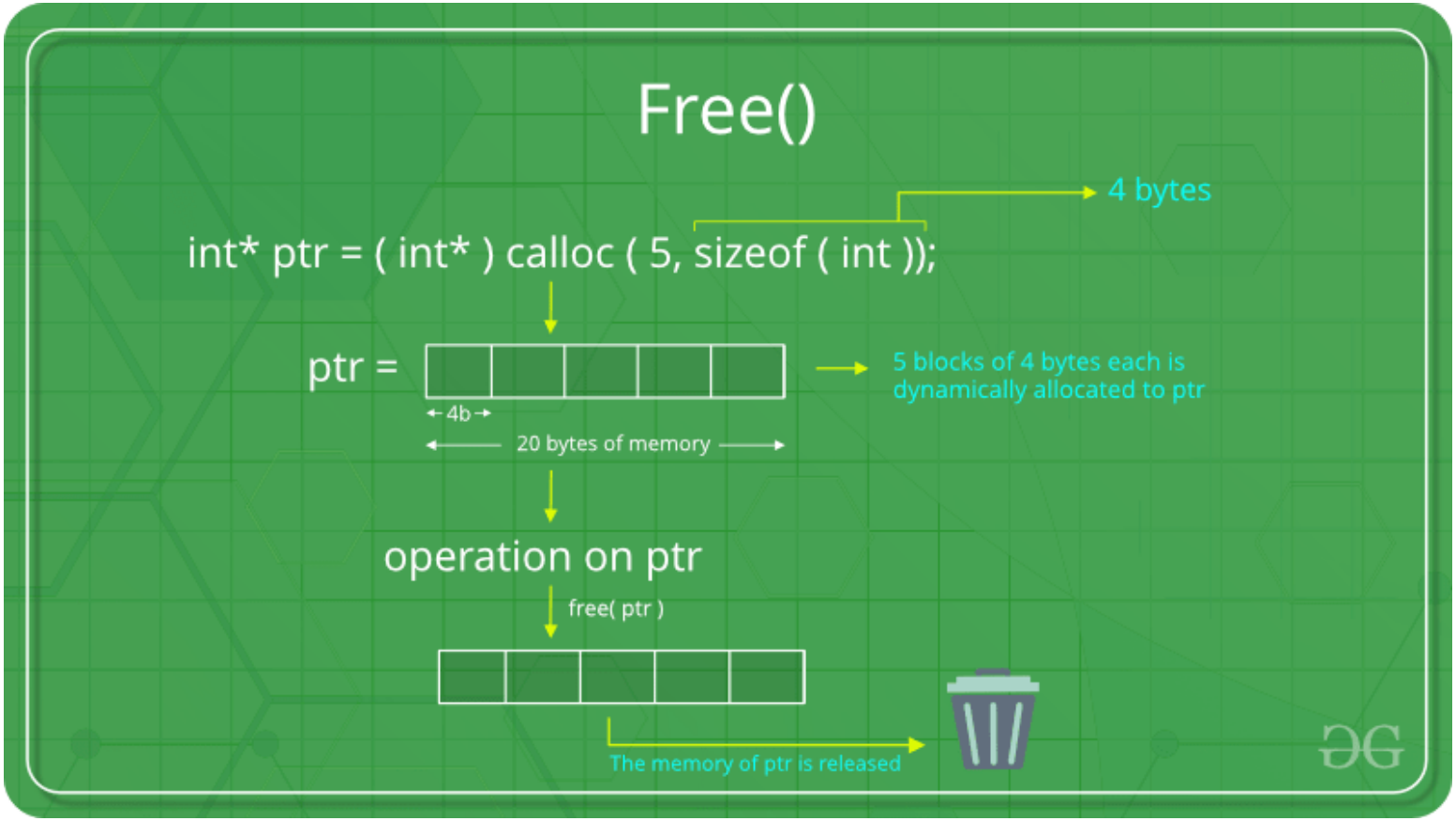


No `free()` Lunches

We now know how to allocate memory, so let's talk about how to deallocate memory.

- Memory manually allocated using `malloc()` *must* be deallocated manually as well.
  - To **deallocate** is to take memory allocated to a program, and return control of that memory to the operating system.
  - If no program deallocated the memory it used, you would have to restart your computer *very frequently!*
- The `free()` function accepts a pointer as an argument, and deallocates the memory pointed to.
- While it is not enforced by the compiler, freeing dynamic memory is the same as closing filestreams: Very good practice.

No `free()` Lunches (cont.)



An example using malloc() and free()

```
1  void mallocDemo (int n) {
2    int* ptr = malloc(n*sizeof(int));
3    if (ptr == NULL) {
4      printf("Runtime Error!");
5      return;
6    }
7    printf("The memory location allocated is %p\n", ptr);
8    for (int i = 0; i < n ; i ++) {
9      ptr[i] = i;
10   }
11   printf("The allocated array is : ");
12   printArray(ptr, n);
13   free(ptr); return;
14 }
```

`calloc()` : Memory Allocation For the Hygiene Obsessed

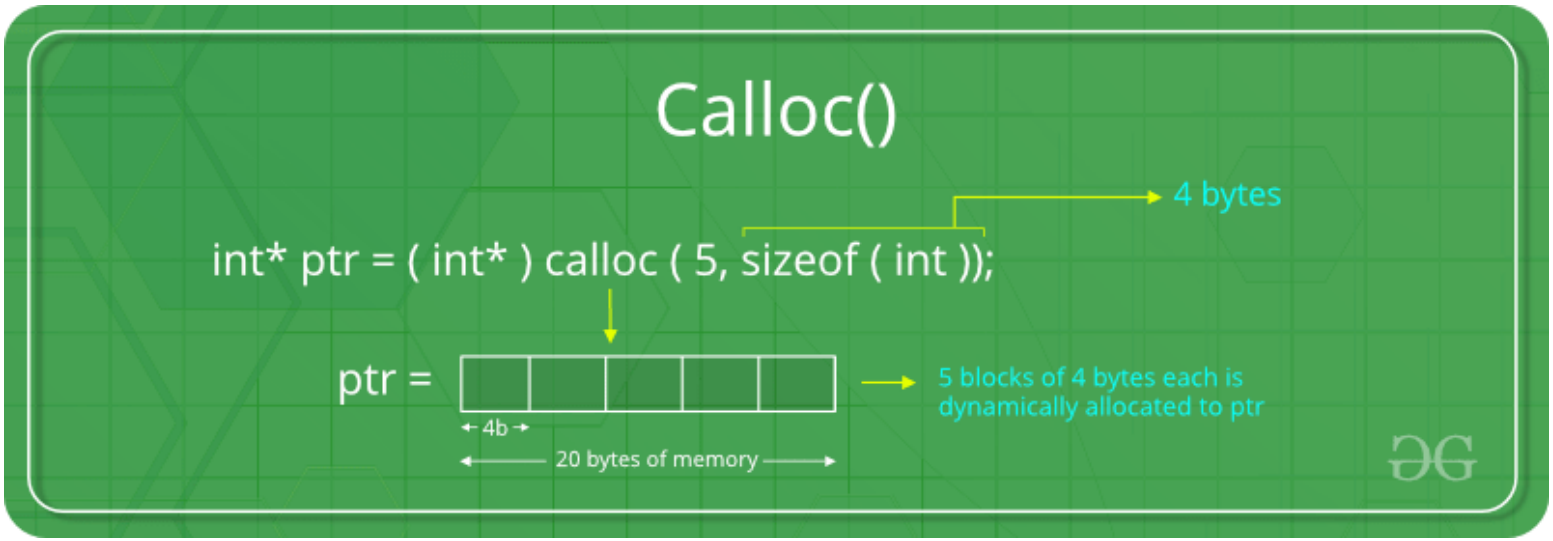
An alternative to `malloc()` is `calloc()`

- `calloc()` accepts two arguments:
  - The first is the number of chunks of contiguous memory to allocate
  - The second is the size of these chunks of memory in bytes.
- Additionally, `calloc()` wipes all allocated memory (i.e., writes 0 to each chunk).
- Aside from this, usage is exactly the same as `malloc()`.

```
1 int n = 100;
2 int *ptr = calloc(n, sizeof(float));
```

- `calloc()` is slower than `malloc()`, however, as overwriting the allocated memory takes time.

`calloc()` : Memory Allocation For the Hygiene Obsessed



realloc() : Change It! Because I Said So!

So what if we need to dynamically change memory that's been dynamically allocated?

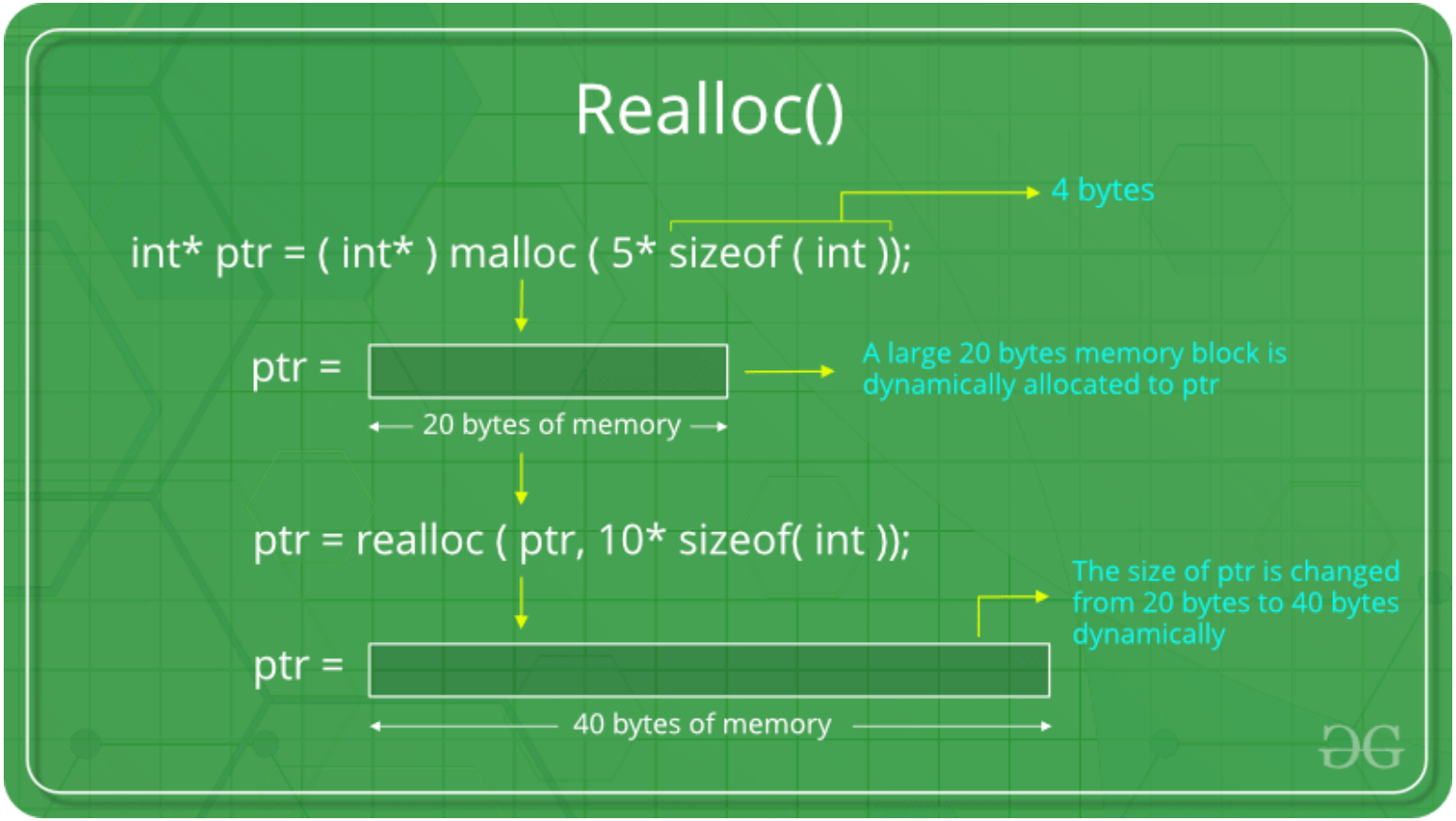
- realloc() allows us to change the amount of memory allocated to a pointer, while keeping the data already in it!

```
1 int *ptr = calloc(40, sizeof(float));
2 ptr = realloc(ptr, 100*sizeof(float));
```

  - The first argument is the pointer to be resized.
  - The second argument is the new size, following malloc() rules.

- realloc() returns a pointer to the newly allocated memeory
- New chunks of memory will be filled with smelly garbage!

textttrealloc() : Change It! Because I Said So!



The Last Slide Comic